UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Practical Dynamic Software Updating

Luís Gabriel Ganchinho de Pina

Supervisor: Doctor Luís Manuel Antunes Veiga
Co-Supervisor: Doctor Michael William Hicks

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Michael William Hicks
Doctor Luís Manuel Marques da Costa Caires
Doctor David Manuel Martins de Matos
Doctor Luís Manuel Antunes Veiga
Doctor Paulo Jorge Fernandes Carreira
Doctor Gavin Mark Bierman

2016

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Practical Dynamic Software Updating

Luís Gabriel Ganchinho de Pina

Supervisor: Doctor Luís Manuel Antunes Veiga
Co-Supervisor: Doctor Michael William Hicks

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:

Doctor Michael William Hicks, Professor, University of Maryland, USA
Doctor Luís Manuel Marques da Costa Caires, Professor Catedrático da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
Doctor David Manuel Martins de Matos, Professor Auxiliar do Instituto Superior Técnico da Universidade de Lisboa
Doctor Luís Manuel Antunes Veiga, Professor Auxiliar do Instituto Superior Técnico da Universidade de Lisboa
Doctor Paulo Jorge Fernandes Carreira, Professor Auxiliar do Instituto Superior Técnico da Universidade de Lisboa
Doctor Gavin Mark Bierman, Researcher, Oracle Labs, UK

Funding Institutions

Fundação para a Ciência e a Tecnologia
INESC-ID
European Commission
United States National Science Foundation
Partnership between UMIACS and the Laboratory for Telecommunication Sciences

2016

Resumo

Actualizar um programa é indispensável para corrigir erros, adicionar funcionalidade, ou melhorar o desempenho. No entanto, actualizações interferem com a normal execução do program dado que requerem parar e recomeçar o programa, com perda de disponibilidade e dados como efeito secundário. A capacidade de actualizar um programa sem o parar — *Actualização Dinâmica de Software* — é cada vez mais importante, especialmente quando perda de disponibilidade se traduz directamente em perda de receita.

Existem, claro, sistemas com requisitos de alta disponibilidade que já suportam actualizações dinâmicas. Estes sistemas, no entanto, requerem hardware redundante, já presente para tolerância a faltas, para actualizar incrementalmente algumas máquinas enquanto as restantes continuam a fornecer serviço. Estas abordagens usam algoritmos complexos e especifícos a cada sistema em particular que restrigem a flexibilidade das actualizações. A comunidade científica investigou este problema, tendo produzido uma vasta bibliografia acerca do tema. No entanto, até à data, não existe nenhuma abordagem prática para actualizações dinâmicas de software.

Neste trabalho, proponho a primeira solução *prática* para Actualizações Dinâmicas de Software para linguagens que executam num ambiente gerido, Java em particular. A abordagem que proponho suporta modificações sem restrições entre versões sucessivas do programa e não limita o program de usar qualquer característica da linguagem ou funcionalidade do ambiente de execução. Além disso, não adiciona quaisquer custos de desempenho adicionais e apenas requer uma pausa curta, durante a normal execução do programa, para realizar uma actualização (que não é proporcional ao tamanho do estado do programa).

Proponho que as actualizações sejam suportadas explicitamente como qualquer outra funcionalidade do programa. Portanto, o programador precisa de alterar o programa para suportar actualizações. A solução que proponho minimiza o número de alterações manuais necessárias e gera automaticamente a maior parte do código que descreve cada actualização. Além disso, dado que o programador pode inadvertidamente introduzir erros no programa que apenas se tornam visíveis durante o proceasso de actualização, a abordagem que proponho permite que o programador use testes de sistema já existentes, e escreva novos testes, que garantem que o programa exibe o comportamento esperado após uma actualização.

Abstract

Updating a program is unavoidable to fix bugs, add features, or improve performance. This is, however, a disruptive operation that involves stopping and restarting the running program, with the side-effect of service downtime and data loss. The ability to update a program without stopping it — to perform a *Dynamic Software Update* — is thus increasingly important in a world where service downtime and data loss map directly to loss of revenue.

There are, of course, highly-available systems that simply cannot stop and already support dynamic updates. These systems, however, rely on redundant hardware, already present for fault tolerance, to incrementally update some machines while others keep providing the service. These approaches employ complex and domain-specific algorithms that restrict the flexibility of updates. The research community has focused on this problem and produced a vast body of work. However, to date, there is no practical solution for dynamic software updating.

In this work, I propose the first *practical* solution for Dynamic Software Updating for languages that run in a managed environment, in particular, Java. The approach I propose supports unrestricted changes between successive program versions and does not limit the updatable program from using any language or runtime feature. Moreover, it does not add any steady-state overhead and requires only a short pause in program execution to perform an update (that is not proportional to the size of the program state).

I propose updates to be supported explicitly as program features. Therefore, the developer needs to change their application to support updating it. The solution I propose minimizes the required manual changes and automates most of the code that describes each update. Furthermore, given that the developer may inadvertently introduce errors, only visible during the update process, the approach I propose provides a way for the developer to re-use existing system tests, and write new ones, that ensure that the updated program behaves as expected after an update.

Palavras-Chave
Keywords

Palavras-Chave

Actualizações Dinâmicas de Software
Java
Máquina Virtual Java
Memória Transacional em Software
Testes Sistemáticos
Alta Disponibilidade
Proxies
Re-escrita de Binário
Reciclagem Automática de Memória
Testes de Software

Keywords

Dynamic Software Updates
Java
Java Virtual Machine
Software Transactional Memory
Systematic Testing
High Availability
Proxies
Binary Rewriting
Garbage Collection
Software Testing

Acknowledgments

This dissertation is the final result of six years of research, which involved a fair number of challenges, successes, failures, errors, and achievements. Of course, I was only able to overcome all the obstacles because I was fortunate enough to have the support and help of several people. I would like to thank all of them.

First and foremost, I would like to thank my family. To my mother, Conceição; my grandmother, Idalina; and my uncle, José; I thank the unconditional support they provided ever since I started my academic career. They always believed in my ability to make this dissertation come to fruition and encouraged me to persevere until I reached all the goals I had set for myself.

The work I present in this document was supervised by: João Cachopo, as my original advisor; Luís Veiga, as my final advisor; and Michael Hicks, as my co-advisor. João Cachopo introduced me to academic research. His guidance showed me how to be a methodical scientist that clearly communicates ideas and findings. I thank Luís Veiga for having accepted to be my advisor mid-way through my research. The discussions we had about the current state of my research and the possible avenues for its future were always a source of motivation and inspiration. Michael was always a kind supervisor and a resourceful collaborator. He held me to a high standard and led me to improve myself as a researcher. I thank him for having accepted me as an intern in 2012, thus starting a collaboration that was pivotal to my academic trajectory and, ultimately, made this document possible.

During my research and while writing this document, I was fortunate to have been supervised by Jeff Foster and Cristian Cadar on projects that were not related to this dissertation. I am thankful for the opportunity they gave me to devote some time to this research while we collaborated, and for the supervision in projects that widened my horizons as a researcher, which improved the quality of this dissertation.

I was part of three research groups while working on my dissertation: The Software Engineering Group (ESW), at INESC-ID Lisboa; the Programming Languages Group (PLUM), at University of Maryland; and the Software Reliability Group (SRG), at Imperial College London. I am grateful to the institutions that host these groups for providing me with an healthy and pleasant workspace. I would like to thank all of their members for the insightful discussions about research, which vastly improved my knowledge about Computer Science in general, and the informal conversations, lunches, and social outings, which helped me to relax and regenerated my ability to come up with fresh ideas and pursue them.

I would like to thank the committee that reviewed this document and provided me with feedback that greatly improved the quality of the manuscript. I would also like to thank all the anonymous reviewers that read early versions of my work, in the form of paper drafts submitted for publishing in international conferences, for all the feedback they provided on the quality and direction of my work.

This work was financially supported by: Fundação para a Ciência e Tecnologia (FCT) under grant SFRH / BD / 66249 / 2009 and projects PTDC/EIA-EIA/113613/2009 and PEstOE/EEI/LA0021/2014; INESC-ID through multiannual funding and the PIDDAC program funds; the European commission through the Cloud-TM project (contract number 257784); the National Science Foundation (NSF) through grant CCF-0910530; and the partnership between UMIACS and the Laboratory for Telecommunication Sciences. Without all these entities sponsoring my research through funding, I would not have been able to afford completing my degree.

February 1, 2016
Luís Gabriel Ganchinho de Pina

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomic-Consistent-Isolated-Durable
API	Application Program Interface
CAS	Compare-and-swap
DBMS	Database Management System
DSL	Domain Specific Language
DSU	Dynamic Software Update/Updating
GC	Garbage Collection
IDE	Integrated Development Environment
JIT	Just-in-time
JVM	Java Virtual Machine
LOC	Lines of code
MOP	Meta-object Protocol
OODBMS	Object-Oriented Database Management System
OS	Operating System
OSGi	Open Services Gateway initiative
PC	Program Counter
RFC	Request for comments
SQL	Structured Query Language
STM	Software Transactional Memory
TM	Transactional Memory
VM	Virtual Machine

Chapter 1

Introduction

The ubiquity of the Internet changed dramatically how software is perceived: Security bugs leave systems vulnerable to malicious entities and must be fixed as soon as possible; service providers must ensure high availability and frequently updated features to win and keep their user base. Software updates have the ability to correct bugs, add features, or improve performance. They are unavoidable in the constant struggle for responding quickly to any of these forces.

Updating a program is, however, a disruptive operation. It typically involves stopping the outdated program and starting its new version. Besides reducing availability, the update also leads to the loss of all non-persistent program state, such as the contents of the memory heap, the execution stack and program counter, and any network connection the program was using at the time of the update. The ability to perform *dynamic software updates* (DSU), i.e., to update a running program in place without stopping it, provides the solution to this problem.

Designing and deploying highly available systems is not a new problem. Such systems have been studied since before the internet age and are nowadays successfully deployed on a massive scale. They implement dynamic software updates using techniques such as *rolling updates* or *big flips* [Bre01] that leverage on redundant hardware, already present to tolerate faults, to update some machines separately while others keep providing the service. Besides requiring additional hardware, these techniques have their own shortcomings. In particular, they require complex algorithms to migrate the program state between versions. Such algorithms are hard to reason about, to ensure correctness, and too domain specific to be applicable to other programs as a framework. As a result, these techniques end up restricting the overall flexibility of updates, e.g. by requiring stateless programs/protocols to ensure safety or compatibility between versions.

Over the past 15 years, the way in which software systems are developed and deployed changed to accommodate the vast growth of the Internet and the research on Dynamic Software Updating has taken advantage of the new opportunities this change created. Most notably:

- Operating System (OS) kernels are now more modular and sophisticated, allowing for the development of DSU systems that specifically target OS kernels [BHA+05, CCZ+06, MR07, AK09, GKT13].

- On the user-space, C applications are still used today to run the backbone of the services that the Internet provides. If anything, the focus on their high-availability is now stronger than ever, and the research community acknowledged that focus with a new generation of DSU systems [ABBS05, NHSO06, CYC+07, MB09, HSHF11, HSD+12, PBG13];

- Languages with a managed runtime, such as Java, have become more important. These languages provide developers with memory safety through Garbage-Collection (GC) [JHM11] and performance for expressive high-level language constructs (such as dynamic dispatch) through Just-In-Time (JIT) optimizing compilers. The research community provided several systems that target such languages at their various levels of implementation [RA00, MPG+00, Dmi01, Orab, ORH02, SHM09, WWS10, PGS+11, KV12].

Despite all this research in DSU, state-of-the-art software systems today do not employ most of the knowledge it generated to improve their availability. Few DSU systems ever made it to the industry: KSplice [AK09] and HotSwap [Dmi01, Orab]; providing support for deploying security patches and for implementing stop-edit-continue debugging functionalities, respectively. These systems place restrictive limitations on the flexibility of updates they support. For instance, they only support changing the code that the program executes to a new version that has the same structure, i.e., the same functions / methods / classes with the same name and signature. I thus claim that a practical solution to perform dynamic software updates still eludes researchers and practitioners.

1.1 Challenges of Dynamic Software Updating

A software update is free to, and even expected to, change the semantics of a program. As a consequence, dynamic software updating raises several challenges that require careful reasoning about the meaning of an update taking place while the program is executing.

The ability to perform an update may introduce *steady-state overhead*, making the program run slower when not updating than it would run without the ability to perform DSU. When updating, the system starts by choosing an instant during program execution *when to perform the update*. During the update, it *transforms the program state* to be compatible with the new program code, which may introduce a noticeable *update-induced pause* in execution. *Update correctness* is orthogonal to all challenges, given that an incorrect update may result in a program that crashes, corrupts its data, or otherwise misbehaves. In this section, I explain each challenge in further detail.

1.1.1 When to update

Let us start by considering the instant when the update takes place. Suppose that the program being updated is simply a set of functions that call each other starting from one main function, and the update changes a subset of those functions. Functions are sequences of instructions and the program counter (PC) keeps track of the current instruction being executed. Calling a function pushes a frame to the top of the execution stack, which contains the current PC as the return address and the arguments/local variables of the called function, and changes the PC to the start of the function. The function thus becomes active, until the invocation completes, at which point the frame is popped from the stack and the PC is reset to the return address.

Even in this simple computation model, the ability to perform DSU raises questions. For instance, should we allow updates to active functions? One option is to allow it but keep executing old functions active at the time of the update; only new calls are made to the updated code. Another option is to map all the frames of the old function to equivalent frames of the new function [MB09]. It also has to map the current PC to an equivalent position in the new code. Considering that functions might change arbitrarily between two versions, mapping the frames and PC is a hard problem.

```
1  run() {
2    process();
3
4    cleanup();
5  }
```

Figure 1.1: Example illustrating different versions of the same program interacting due to an update. In one program version, function `cleanup` initializes and uses some state. The following program version moves the initialization code to function `process`. Updating this program at line 3, when all the modified code is not active, results in the old function `process` interacting with the new function `cleanup` and thus crashing the program when function `cleanup` uses the uninitialized state.

The alternative is to not support updates to active functions. This approach is indeed simpler, but has its own problems. For instance, if an updated function never becomes inactive, the program might never get updated. Such long running functions are common on server software that executes commands from clients on a long-running loop. The function that contains the loop never becomes inactive.

Regardless of the alternative, the choice of updating a program at the level of function calls enables the interaction between old code and new code. For instance, consider the example shown in Figure 1.1. Function `run` calls functions `process` and `cleanup` in sequence. An update moves initialization code from function `cleanup` to function `process`. Note that no updated function is active on function `run` at line 3. However, performing the update at this point crashes the program because the new function `cleanup` will access state that was supposed to be initialized by the new function `process` but was not, because the old function `process` ran instead.

1.1.2 Migrate the program state

A running program keeps state in many forms. For instance, in the previous challenge, I mentioned the execution stack and the PC. Both are forms of program state. Besides the stack, programs typically keep their data structures on the heap.

Data structures have a structured format that allows program code to manipulate them. A program update can change the format and update the program code accordingly, so that any program that starts executing with a blank state in the new version uses the new format.

A Dynamic Software Update (DSU), however, starts executing the new code with the old program state. Executing the updated code on the old data structures might result in a crash or in wrong behavior. To avoid this possibility, DSU systems must be able to migrate the data structures when updating a program so that they keep the same data but in the new format and, therefore, are compatible with the new code.

1.1.3 Update induced pause

The first challenge can be reduced to choosing the instant when the update takes place. Suppose that we have an answer to what is an acceptable instant to update a running program. A possible way to enforce updates taking place only at those instants is to wait until the program reaches such an instant after the update is made available.

The second challenge can be reduced to transforming the program state to an equivalent state that is compatible with the updated program. A simple approach is to transform the whole program state at once, pausing the execution of the program while the transformation is taking place.

These are two prudent solutions to each of the previous challenges. However, each one introduces latency between the moment an update is available and the moment the program starts executing the new version. During this time, the program is paused and not providing service. Even worse, the length of the pause grows with the complexity of the control-flow structure of the program and the total size of the program state, respectively.

In the worst case, the pause in program execution that a DSU introduces becomes comparable, if not higher, to the downtime required to stop the program and restart it in the following version. Even though a DSU does not result in the loss of any part of the program state, such a long pause partially defeats the main motivation for supporting DSU.

1.1.4 Steady-state overhead

The ability to update programs dynamically may require modifications to the program or the runtime environment that executes it. For instance, we can implement functions by using a jump table and ensuring that all function calls in the updatable program are made through the jump table. To update the definition of a function, all we need to do is update the respective entry in the table.

Rewriting the program or changing its runtime environment may result in lower performance while the program is not performing an update, i.e., while it is executing in steady-state. In the example we are following, each function call now requires a table lookup.

The fact that performing a DSU induces a pause in the execution of the program is expected. However, besides that expected cost, executing the program in such a way that it has the ability to be updated dynamically might add constant performance overhead, even when not updating.

1.1.5 Update correctness

To fully support DSU, the developer of the updatable program might be required to perform extra tasks besides just writing the new program version. For instance, he might have to write code that migrates the data structures from one version to the following one. Or he might be required to identify, manually, program points where updates can take place.

An error made by the developer in any of these extra tasks might translate to a crash, or other program misbehavior, when performing a DSU. When developing the original program, the developer already has to deal with possible errors. However, he has tools that help him find and fix those bugs before the program is considered ready to be put into production and start providing service.

Throughout this section, I have listed several scenarios that result in an unexpected program crash due to the DSU process itself. In addition, the update may also involve executing code written by the developer. How can the developer trust that the DSU will not result in an immediate program crash? Or that it will not corrupt the program state? Or add a latent bug that will crash the program in an indeterminate amount of time after a DSU? At a higher level: How can the developer be sure that the client-visible behavior is correct, despite potential updates happening while processing client requests?

1.2 Goals

In the previous section, I listed the main challenges that come up when considering DSU. Together with some challenges, I gave examples of possible solutions that are straightforward but not acceptable for some reason. These examples highlight how the problem of DSU is complex and why a good solution requires careful reasoning about subtle details.

I did not, however, describe what a good solution is. In this section, I present a set of goals that a solution for DSU should reach and I argue why these goals matter when designing and using DSU in practice.

1.2.1 Flexibility

In Section 1.1.1, I explored the design space for performing DSU at the function level for a simple language based on functions. Supporting DSU when updated functions are active is challenging because we have to migrate the PC and the stack frames for all updated functions that are active at the time of the update.

Suppose that we place the following restrictions on the supported DSU: Local variables cannot be modified and functions can only replace one instruction by another. These restrictions trivially solve the technical challenge of supporting DSU for active functions: There is a one-to-one mapping between the local variables and the PC in both program versions.

This is, however, a very restrictive form of DSU. It is not clear whether we can use it to perform even simple code changes, such as adjusting a loop guard to avoid an out-of-bounds access to an array. Another way to say this is that the *flexibility* of this type of DSU is very limited.

Goal. *A flexible DSU maximizes the types of program modifications that it supports.*

Flexibility is not a binary choice that states whether a DSU solution reaches it. Flexibility identifies, instead, a continuous dimension in the DSU design space. For instance, a flexible DSU for the example we are following can modify the body of any function present in the outdated program in any way. An even more flexible DSU system can, on top of that, add or remove functions between versions.

A possible way of validating this goal is to use existing software, originally built without support for DSU, and check whether a DSU can update it from one release to the following. A particular design choice for DSU that limits the types of program changes it supports might still be flexible enough to be used in practice.

1.2.2 Correctness

After performing a DSU, it is natural to expect that the program is executing on the new version in a state equivalent to the one in which the old version was stopped executing. We expect that a *correct DSU* does not result in a crash at update time and does not corrupt data in a way that introduces any semantic errors in the new version. What does it mean, however, for a DSU to be correct?

Researchers have studied the topic of DSU correctness and there are several definitions in the literature. Kramer and Magee [KM90] consider updates to be correct if the updated program preserves all observable behaviors of the old program, that is, if updates are *observationally equivalent*. Although intuitive, this definition is too restrictive, as Bloom and Day [BD93] point out: An update that fixes bugs or adds new features is not considered to be correct.

Gupta et. al. [GJB96] address the limitations of strict observational equivalence by proposing *reachability* as the correctness condition for DSU. Their definition takes into account the difference between: (1) Performing a DSU on a program running an old version, and (2) running a program from scratch on the new version. They consider a DSU in case 1 to be correct if it eventually reaches a program state that case 2 would be able to reach.

Reachability improves on observational equivalence because it clearly considers fixing bugs and adding new features as correct updates. Unfortunately, as Hayden et al. [HMH+12] discuss, reachability has serious shortcomings. For instance, consider an update that adds a limit to the maximum number of connected clients to a server program. What happens when this update is performed while the server has more clients connected than the maximum allows? On the one hand, allowing those clients to remain

connected violates reachability because the clients may remain connected for an indefinite amount of time. On the other hand, forcefully terminating client connections at update time does not violate reachability but defeats the purpose of using DSU to avoid losing any part of the program state. Either alternative is unacceptable.

Hayden et. al. [HMH+12] take a different approach to define DSU correctness. They argue against attempts to define correctness in a completely general way and, instead, suggest that it makes more sense for programmers to specify the behavior they expect as a collection of properties called *client-oriented specifications (CO-specs)*. CO-specs are client programs that resemble tests which interact with the updated program, before and after the DSU, to assert if the behavior of the updated program is correct. The authors further classify CO-specs into three major categories: behaviors unaffected by the update (backward-compatible), behaviors specific to the new program version (post-update), and updates that change interfaces but keep core functionality (conformable).

A flexible DSU, as introduced in the previous section, that supports arbitrary changes to how the program represents its state between successive versions, only worsens the problem of asserting the correctness of a DSU. We cannot expect a flexible DSU to be able to migrate the program state between versions automatically. As a consequence, a flexible DSU must allow the developer to write code that migrates the program state. This is yet another moving part that has to be tested to ensure DSU correctness, and that makes the case for CO-specs even stronger.

Goal. *A correct DSU takes some specification of expected behaviors after an update takes place and can either verify exhaustively or test systematically that a DSU conforms to its specifications.*

Ensuring that a DSU is correct thus becomes similar to ensuring that a program conforms to a specification. Verifying that a program written in a general-purpose language conforms to a specification is computationally unfeasible. In practice, verified programs are small and written either in a limited subset of a general-purpose language or in a special language. For larger programs, developers use testing methodologies to ensure correctness up to some bound limited by test coverage. Developers should be to able to do the same for DSU.

1.2.3 Efficiency

The ability to perform DSU has two main advantages: It allows updating a running program without (1) losing any program state, (2) incurring a large execution pause due to the update process, or (3) introducing overhead in steady-state execution just to support future updates. So far, in this section, I only introduced goals for point 1. The efficiency goal addresses points 2 and 3.

Goal. *An efficient DSU minimizes both the performance overhead on steady-state (i.e. when not updating) and the pause in execution required to perform an update.*

A DSU that takes place instantly but requires the program to execute at a fraction of its original performance to support future DSUs cannot be considered efficient. This is exactly what the first part of the efficiency goal states. The execution speed of running an updatable program version when not performing DSU should be as close as possible to the execution speed of running the same program version without the ability to perform DSU.

The pause introduced when performing a DSU should be, ideally, indistinguishable from normal program execution. A DSU needs to migrate the program state so that it is compatible with the new program version. An efficient DSU therefore must minimize the pause in execution required to perform the program state migration.

Pausing the program to perform a DSU for an amount of time proportional to the size of the program state only works for programs that keep a small amount of program state. Otherwise, it is easy to imagine a program execution that builds up an arbitrarily large program state, which in turn introduces an arbitrarily long pause in program execution. Even such programs do not need all their program state all the time: they only require a small portion of it to be readily available to execute without overhead. Such small portion of program state — the *working set* — contains the call stack, the program code, and important program state.

It is reasonable to assume the existence of a working set. In fact, the memory management and paging algorithms in modern operating systems do it [Tan07]. For these types of programs, an acceptable solution for DSU should impose a pause that does not depend linearly on the size of the whole program state. Whilst this may not be possible for all programs, the ideal length of the update pause should be near constant.

1.2.4 Effectiveness

All the goals that this section listed so far can be broadly described as properties of DSU itself. The ability to perform DSU on a running program, however, depends on several factors, such as, for instance, the language and runtime environment of the programming language in which the program undergoing DSU is written.

A possible way to perform DSU is to require programs to be written in a language specifically designed for that purpose. Or, alternatively, to write the updatable program in a popular language but in a way that simplifies the challenges of implementing DSU, according to the goals so far introduced, however cumbersome it might be (e.g. writing C code in Continuation Passing Style [SS75]).

I argue that a solution that does so greatly reduces the utility of DSU, to the point it becomes practically unusable and, ultimately, ineffective.

Goal. *An effective DSU targets popular languages, minimizes any restriction on the style in which updatable programs are written, and allows programmers to use the same development time tools that they would otherwise use.*

An effective solution for DSU maximizes its impact, bringing the advantages of DSU to a potentially large number of existing programs while requiring minimal, if any, effort to adapt those programs to support DSU.

1.3 Past Work

There is already a vast body of literature dedicated to the topic of DSU. The combination of goals that I propose in the previous section, however, is novel as there are no DSU approaches that achieve all of them. In this section, I describe the most relevant approaches and I explain why they still fall short of the intended goals. I leave a more detailed discussion of the state of the art on DSU to Chapter 2.

Some languages have built-in support for DSU. Notable examples are: The CLOS object system for LISP [Ste90], Smalltalk [GR83], and Erlang [AVWW96]. Other languages are designed with the explicit goal of supporting DSU, such as UpgradeJ [BPN08]. The problem with language-based approaches to DSU is that they require programs to be written in a particular language which, unfortunately, is not very popular. Therefore, most of the existing code cannot benefit from the DSU support that these approaches provide. Language-based approaches thus fail the effectiveness goal.

There are some systems that provide support for DSU and that are used in practice. Notable examples are the KSplice system [AK09], which allows a Linux kernel to be updated with security patches without restarting; and the JVM HotSwap mechanism [Orab, Dmi01], which allows JVM bytecode to be redefined to enable break-edit-continue features on Java IDEs and debuggers. The problem with these systems is that they provide very limited flexibility: These approaches support updates that can only change the body of existing functions/methods and that do not change the static structure of the program. Furthermore, neither of these approaches support updates that might require transforming the program state between successive versions.

Some systems are designed around modules: The main program provides only support for module registration and communication, and the behavior of the whole system comes from the interactions between different modules. A notable example is OSGi [OSG14]. In fact, the earliest document that proposes DSU, by Fabry et. al. [Fab76], suggests that such a modular approach eases the task of providing support for DSU because each module can be updated internally in a way that is transparent to the rest of the system. Modular systems, however, are not a panacea for DSU. As Gregersen and Jørgensen [BTW07] point out, a module A can hold an internal reference to another module B that becomes invalid when module B is updated. Migrating the state between successive versions of the same module is also a challenge, as well as updating several inter-dependent modules in a single atomic step. All these problems defeat the flexibility and effectiveness goals.

The most recent line of work that discusses update correctness [HMH+12, HSH+12] is based on Ginseng, which enables DSU for existing C applications and was tested with 6 existing programs [NHSO06, NH09]. It is possible to assert the correctness of a DSU made through Ginseng using systematic testing [HSH+12]. Ginseng is very flexible and supports lazy program state migration. Ginseng, however, introduces measurable performance overhead on steady-state execution. Ginseng supports updates that increase the size of existing structures up to some limit. Once the size of an updated structure hits the limit, Ginseng does not support updates that further increase the size of that structure without restarting the program and limits the programming style to satisfy its conservative static analysis. These three issues, the performance overhead, the limit on the maximum size of updated structures, and the programming style limits, mean that Ginseng fails to achieve the efficiency and effectiveness goals.

Kitsune [HSD+12] is a recent system that supports DSU for C programs. It is very flexible and supports nearly any update made to a C program. Kitsune was tested on 5 existing real-world programs and does not introduce any measurable performance overhead on steady-state execution. Kitsune requires manual changes to the updatable program to support DSU. The developer has to mark *update points*, which are program points where updates can take place. When an update becomes available, Kitsune stops all threads at the next update point each thread reaches. When all threads are stopped, Kitsune traverses the heap and migrates the program state according to state transformation code that the developer can customize. Finally, Kitsune restarts all threads from the top. The developer has to add *control-flow migration* code that guides the restarting thread to the update point at which it was stopped, avoiding any code that would re-initialize state that Kitsune already migrated. This approach comes very close to reach all the goals I propose, but it has two problems: (1) It only supports immediate program-state migration, which means that an arbitrarily large program state would impose an arbitrarily large pause to migrate it between versions; and (2) it does not support any framework to test whether the update points, control-flow migration, and program-state migration are correct and will not crash the program when performing a DSU. These two issues mean that Kitsune fails the efficiency and correctness goal, respectively.

There are several approaches for supporting DSU for Java. The vast majority adds non-trivial performance overhead to steady-state execution [RA00, MPG+00, ORH02] and thus fail the efficiency goal. JRebel [KV12] is an extension of the HotSwap mechanism that adds support to updating the set of fields and methods that a class defines. However, it does not support migrating the program state between versions (new fields are initialized with their default values) or changes to the class hierarchy. Therefore, JavaRebel fails the flexibility goal.

Both JVolve [SHM09] and the DCE-VM [WWS10] are custom JVMs that support DSU without imposing any noticeable performance overhead. The DCE-VM stops threads at VM safe-points[1] to perform DSU. Updates may fail because update safe-points are only a subset of VM safe-points. JVolve also stops threads at VM safe-points but then checks if all threads are stopped at an update safe-point. If JVolve fails to stop all threads at update safe-points after a certain amount of time, the update fails and the program keeps executing in the old version. JVolve allows the developer to blacklist methods during which the update cannot happen, but there are still some situations in which an update with JVolve might cause the program to crash if the developer fails to blacklist some methods that cannot be interrupted to perform a DSU. Neither the DCE-VM nor JVolve has any mechanism to ensure that bad timings cannot happen or to test exhaustively all timings. Therefore, in both approaches, it is very hard to reason about the correctness of a DSU because a particular bad timing might cause an update to crash the program. None of these approaches support updates that change the class hierarchy. Also, both approaches migrate the program state between versions by executing a full garbage-collection cycle, whose duration is proportional to the size of the program state and thus may pause the program for an arbitrarily long amount of time when performing a DSU. Therefore, both these approaches fail the correctness, flexibility, and efficiency goals.

1.4 Thesis Statement

In Section 1.2, I defined a set of goals that a good solution for DSU should reach. Then, in Section 1.3, I briefly discussed why the most relevant systems that support DSU only reach some, but not all, of the goals I defined.

The main contribution of this document is the description of the design, implementation, and evaluation of a system that supports performing DSU and that reaches all the goals that I defined. In short, I shall show that:

*It is possible to design and implement a system that can perform **D**ynamic **S**oftware **U**pdating on a running program in an **effective** and **efficient** way, that maximizes **flexibility**, and that allows the developer to reason about and to test the **correctness** of the program after performing a DSU.*

1.5 Contributions

This document makes the following contributions:

1. It proposes a semantics for performing DSU on programs running on a transactional memory (TM) system that naturally supports lazy program state transformation;

2. It describes the design and evaluation of DuSTM, a prototype system that implements the semantics mentioned in the previous point for programs written in the Java programming language, and that does not require a custom JVM;

[1] A VM safe-point is a point in the execution of a thread where it is safe to perform garbage collection and to reschedule threads.

3. It presents two algorithms for performing program-state transformation when updating running Java programs that either: (1) Eagerly use multiple threads to minimize the update-induced pause; or (2) lazily transform the program state as the new program code reaches it for the first time after the update, amortizing the cost of program-state transformation over the execution of the new program version and thus minimizing the update-induced pause in execution;

4. It describes the design and evaluation of Rubah, a prototype system that implements the algorithms mentioned in the previous point for programs written in the Java programming language, that does not require a custom JVM, that enjoys good steady-state performance before and after the update and low update-induced pauses, and that can be readily applied to existing Java programs with little developer effort;

5. It introduces testing techniques that use existing system tests to assess the correctness of programs updated through DSU, and that uses systematic testing to explore all possible timing opportunities at which the update could be applied to the running program;

6. It describes the design and implementation of Tedsuto, a framework for testing updates performed with Rubah that uses the techniques described in the previous point; that is effective at finding update errors, having discovered several unknown problems; and that can be implemented for any DSU system with simple modifications to how they work.

The contributions that I present is this document have been validated by the research community [PC12, PVH13, PVH14, PH16]. In particular, the work on Rubah [PVH14] was published on a top tier conference (A* in the CORE ranking).

1.6 Structure of this document

The rest of this document is structured as follows:

Chapter 2 describes the existing solutions for DSU and why they do not reach all the goals that I introduced earlier in this section;

Chapter 3 describes *DuSTM*, which is a DSU system for a rich programming model (Java + STM), that provides simple update semantics and allows any part of the program to change between versions, thus reaching the goal of flexibility. DuSTM also transforms the program state lazily between versions and does not require a custom JVM, thus paving the way to an efficient and effective DSU system.

Chapter 4 describes *Rubah*, which is a DSU system that is: effective, relaxing the programming model of DuSTM so that it is applicable to existing Java software; efficient, introducing no steady-state overhead and transforming the program state lazily between versions, and as flexible as DuSTM.

Chapter 5 describes *Tedsuto*, which is a systematic testing framework for DSU that allows the developer to test updates developed with Rubah, and thus ensure their correctness.

Chapter 6 concludes this document.

Chapter 2

State of the Art

Upgrading a running program without stopping it is a problem well known to the research community. Fabry [Fab76] was the first to notice this research problem, focusing on changing program modules on the fly. Much has changed since then and other approaches have since appeared: Programming languages that support dynamic updates directly at the language level [Ste90, GR83, AVWW96], update systems that increase the productivity of programmers by performing DSU during development time [Orab, WWS10, KV12, PGS+11], runtime environments that support DSU for managed languages [SHM09, ORH02, RA00], operating systems that can install new updates without requiring a reboot [AK09, MR07, GKT13], formal approaches to dynamic software updates [GJB96, BLS+03b, BHSS03, SHB+07, NHFP08], among others. This is an active topic that still gathers the attention of ongoing research.

In this chapter, I discuss the state of the art on dynamic software updating. I start by proposing a classification of software updates in Section 2.1, useful to reason about the applicability of different systems in a common framework that makes comparison easier. At the end of each section, I discuss all presented approaches and relate them back to the original goals that I set in Section 1.2. In Section 2.2, I explain the approaches that formalize DSU to reason about its correctness. In Sections 2.3 through 2.7, I present the state of the art on DSU; Table 2.1 summarizes all the considered solutions. Finally, in Section 2.8, I discuss the presented state-of-the-art as a whole, referring back to Table 2.1.

The Notation of Table 2.1 provides visual clues about whether each particular solution (row) achieves a particular goal (column). Each cell has a circle, which can be empty, fully filled, or partially filled. Empty circles — ○ — mean that the solution does not achieve the goal. Fully filled circles — ● — mean that the solution fully achieves the goal. Partially filled circles — ◔ , ◓ , and ◐ — mean that the solution partially achieves the goal. There is an order relation between partially achieve goals: ◔ is further away from achieving the goal than ◓ , which in turn is further away than ◐ . In the sub-sections with title *Discussion* in this chapter, I refer back to Table 2.1 and justify each symbol choice in the context of that sub-section. Finally, in Section 2.8, I discuss the table as a whole and justify each symbol choice for each solution in the context of the whole state-of-the-art.

2.1 Classification of Dynamic Software Updating Systems

Systems that support dynamic software updates must deal with several orthogonal challenges: The *timing* at which an update takes place; how to perform *state transformation* between versions when updating, and what *semantics* dynamic updates provide. In this section, I propose a classification that compares update systems according to how they solve each of these challenges.

System				Goals			
Target	**Section**	**Name**	**Refs**	**Flexible**	**Correct**	**Efficient**	**Effective**
C programs	2.3	OPUS	[ABBS05]	○	◔	○	●
		Ginseng	[NHSO06, SHB+07, NHFP08, NH09]	◐	●	◓	◐
		POLUS	[CYC+07]	◓	○	○	●
		UpStare	[MB09]	●	◓	○	●
		Ekiden	[HSHF11]	●	◓	◓	◐
		Kitsune	[HSD+12]	●	◓	◓	●
		DynSec	[PBG13]	○	○	○	◐
OS Kernel		K42	[BHA+05]	●	◔	◓	◓
		LUCOS	[CCZ+06]	○	◔	○	◓
		DynaMOS	[MR07]	●	◔	○	◓
		KSplice	[AK09]	◓	◔	◓	◐
		PROTEOS	[GKT13]	◐	◓	◓	◐
Java Programs	2.4	JDrums	[RA00]	◐	◔	○	◐
		DVM	[MPG+00]	◔	◔	○	◐
		HotSwap	[Dmi01, Orab]	○	○	●	◓
		DUSC	[ORH02]	◔	◔	◔	●
		JVolve	[SHM09]	◐	◓	◓	◐
		DCE VM	[WWS10]	◔	○	◓	◓
		JavAdaptor	[PGS+11]	●	○	◓	◓
		JRebel	[KV12]	◔	○	●	◓
OODBMS	2.5	O_2	[FMZ+95, Zic91]	●	N/A	◓	N/A
		Versant	[Ver15]	◐		◓	
		Objectivity/DB	[Obj13]	◐		◓	
		GemStone	[Gem14]	●		◔	
		PJama	[DA99]	●		○	
Language	2.6	LISP	[Ste90]	●	○	◓	◓
		Smalltalk	[GR83]	●	○	◓	◓
		Erlang	[AVWW96]	●	◓	◓	◓
		UpgradeJ	[BPN08]	●	◐	●	◓
Modules	2.7.1	OSGi	[OSG14]	◔	○	●	◔
		Netbeans	[BTW07]	◔	○	●	◔
		Eclipse	[ML05]	◔	○	●	◔
Distr. Systems	2.7.2	Rolling Upgrade	[Bre01]	◓	○	○	◓
		Big flip		●	○	○	◓
		Imago	[DN09b]	●	●	●	◔
Java Programs	3	DuSTM	—	●	◐	◓	◓
	4	Rubah	—	●	◓	●	●
	5	Rubah + Tedsuto	—	●	●	●	●

Table 2.1: Summary of the state-of-the-art on the various approaches to Dynamic Software Update. The columns under *Goals* refer to the goals for practical DSU that I introduce in Section 1.2. The last three rows on the separate table at the bottom provide a very brief overview of the rest of the document.

The goals for practical DSU that I defined earlier in Section 1.2 define a framework that could also be used to classify the different DSU systems. However, the classification that I propose in this section highlights the subtle differences between each different solution, making the task of comparing them much easier. At the end of each section, I shall discuss how each system achieves, or not, each of the goals for practical DSU.

2.1.1 Timing

The timing at which the update can take place is an important decision when designing a solution for DSU. Restricting when an update can take place has technical repercussions that make each particular solution for DSU easier or harder to implement.

Besides the technical aspects of choosing when updates can take place, timing is also closely tied with the correctness of the update process. During the update, the program state is transformed to be compatible with the new program version. The transformation code may assume certain invariants that may not hold during all the possible times at which an update can be performed. Besides, the semantics of the program may change due to the update, and this change may only make sense at certain points in program execution. A correct solution for DSU must not crash the program simply by performing an update at the wrong time; an incorrect solution may crash a program by performing an update at instants that make the program misbehave after the update.

Note that timing is just a part of DSU correctness. Even if the update process itself does not directly crash the program, a bug introduced by the update can still make the program crash, corrupt its data, or otherwise misbehave.

In the following, I discuss the different solutions for DSU timing.

Unrestricted

This approach provides the least guarantees. Updates can happen at any point during the program execution, even if code that the update changes is active.

This approach is, of course, inherently unsafe. However, by assuming that updates may simply fail, this approach greatly simplifies the implementation of the update system and does not require any type of manual intervention from the developer to support DSU.

The sheer simplicity of unrestricted updates makes them the preferred solution when considering DSU as a development time tool [Orab, KV12, PGS+11, WWS10]. For instance, modern IDE's provide stop-edit-continue support when debugging. As the name implies, this feature allows the developer to: *Stop* the program at a breakpoint, *edit* the code while the program is running, and *continue* execution from the breakpoint with the modified code. In this scenario, it is acceptable to use a DSU system that may sometimes crash the program when resuming from the update. It is an improvement over restarting the program after each modification to continue debugging.

Quiescence

Consider that a program is composed by a set of functions. A function f is *active* if the program is executing either function f or some other function that was called by f (i.e., f's return address is in any stack frame). When function f is not active, we can say that f is *quiescent*. We can extend the notion of quiescence to programs: A program is quiescent with relation to a set of functions F if all of the functions in F are quiescent.

```
1  Object global;
2
3  m() {
4    process();
5
6    cleanup();
7  }
8
9  process() {
10   ...
11
12 }
13
14 cleanup() {
15   global = new Object();
16   ...
17   global.hashCode();
18 }
```

(a) Initial program.

```
19
20
21
22
23
24
25
26
27 process() {
28    global = new Object();
29    ...
30 }
31
32 cleanup() {
33
34   ...
35   global.hashCode();
36 }
```

(b) Updated program.

Figure 2.1: Example illustrating why function quiescence is not a valid correctness property. The left-hand side shows an initial program in Java-like pseudo-code. Method `cleanup` initializes and uses an object referred to by a global reference (e.g. static field) `global`. The right-hand side shows an update that moves the initialization code to method `process`. Both methods are quiescent at line 5. However, performing an update at this point results in executing the initial version of method `process` and the updated version of method `cleanup`, which in turn crashes at line 17.

Some solutions for DSU allow updates only when the program is quiescent when considering all the functions that the update modifies. Such systems take this approach by design, to ensure safety, rather than by technical limitations.

Considering quiescence as a safety condition seems like a reasonable and safe approach. It is hard to reason about the behaviour of the program if code is allowed to change in the middle of a function. However, limiting DSU timing to quiescence also limits the functions that an update can change. For instance, functions that are never quiescent, such as `main` in C programs, cannot be changed.

Unfortunately, quiescence still allows for updates to be installed at unsafe instants. For instance, consider the example that Figure 2.1 shows, adapted from Subramanian et al. [SHM09], which shows the initial version of a simple program (Figure 2.1a) and a possible update (Figure 2.1b). Note that all functions (methods, in this case) that the update changes, `process` and `cleanup`, are quiescent in line 5. Still, performing an update at this point results in a crash in line 17 because the updated version of method `cleanup` (lines 14–18) expects that the updated version of method `process` (lines 9–12) executed before, and set the reference `global` on line 10. In this case, however, the initial version of method `process` (lines 9–12) executed instead, leaving reference `global` uninitialized.

Manual Identification of Update Points

Finding program points where it is safe to perform a DSU is undecidable [GJB96]. Automatic approaches to find such program points thus result in false-negatives that can lead to program crashes [HSH+12]. An alternative is to require the developer to manually annotate the points in the program execution where it is safe to perform an update. Let us refer to those points as *safe update points*, or simply *update points*.

Limiting the number of update points greatly simplifies reasoning about the timing at which the update takes place. However, the program must reach update points frequently, otherwise updates may be delayed for a long time. Besides, this approach offloads the problem of safety to the developer, who may identify wrong update points that still result in a program crash at update time.

Transactions

Transactions group units of work in an application. A transactional application evolves in steps of one atomic transaction: Either a transaction executes completely, with its results becoming visible to all other transactions that start after it; or it aborts, with the rest of the system behaving as if it never took place. Transactions are also a composable way of performing concurrency control.

The isolation between transactions naturally provides safe update points for DSU. The idea is to execute each transaction Tx in the same program version in which Tx started. The obvious downside is this approach requires applications to be written in a particular style, i.e., organized around transactions. Even though transactions provide a more structured way of reasoning about old and new code than manually identified program points, performing an update between transactions may still be wrong. Ultimately, it is up to the developer to ensure that it does not happen.

2.1.2 State Transformation

While executing, programs keep a state that is tightly coupled with their code. Dynamic software updates change the code that programs execute. Therefore, DSU must also transform the program state to an equivalent version that is compatible with the new code.

Existing solutions for DSU use different approaches to transform the program state, ranging from fully automatic to fully manual, with intermediate steps that try to combine the best of both extremes.

In the following, I discuss the different solutions for state transformation when performing a DSU.

Automatic

When performing an update, the system compares the new program version with the one in execution and automatically transforms the state to match the structure that the new program version expects. This is the simplest approach, which requires no manual effort from the developer. It is also the most limited form of program state transformation because certain program changes are completely outside the capabilities of any automatic system (e.g. an update that changes a data structure from a linked list to a binary tree).

Update systems designed as development time tools can take advantage of this limited approach. For instance, by simply copying values from unchanged fields, matching names and types between versions, and initializing new fields with default values (e.g. zero, `false`, or `null`) [BFdH+13].

Indirect

An alternative to transfer the program state between versions is to require the old program version to export its state to an abstract format that can be later interpreted by the new program version to initialize its state. Ebraert et al. [EVDB05] name such a solution as *indirect state transfer* and I shall borrow their term in this classification. This solution has the obvious problem of choosing the right abstract format to export and import the program state, which is application-specific.

Direct

A dynamic update system can provide means for the new program version to transfer the program state directly from the old program version. Ebraert et al. [EVDB05] name such a solution as *direct state transfer*, opposed to indirect state transfer, and I shall also borrow their term in this classification.

With direct state transformation, developers use the same programming language in which the program is written to specify the transformation logic. The update system must thus provide means for them to specify such transformation code. Existing update systems that use direct state transfer allow the developer to specify conversion code using one of the following two options:

Manual Transformation. Developers provide code that performs all the required transformation logic. When performing a DSU, the update system executes this code between program versions. When the transformation code finishes, the update system assumes that the program state is now compatible with the new program code.

This is the most flexible and expressive option. The developers are free to traverse each portion of the program state in any particular order. However, they must navigate through the program state and identify each relevant portion that needs to be transformed, which can be a hard problem on itself, depending on the structure of the updatable program.

Assisted Transformation. Programs typically have a modular structure, enforced by the constructs of the programming language in which they are written. Programs keep their data in structures, records, object, or any other type of structured data representation. Assisted conversion takes advantage of this modular program structure, and requires developers to provide code that transforms each different type of data structure that the program uses. The update system then traverses the program state automatically and executes the appropriate transformation logic for each outdated data structure it finds.

While this approach is less flexible than manual transformation, it frees the developers from the burden of traversing the program state and finding all the portions that need to be transformed.

2.1.3 Semantics

Performing a DSU involves changing the code that the program executes and transforming the state the program keeps to be compatible with the new code. The previous section focused on *how* to transform the program state between versions, but it left an important question unanswered: *When* does the program transformation actually take place? The answer to these two questions provides the *semantics* of the update; in the following I focus on the possible answers for the latter and present the different solutions for update semantics.

Offline

Offline update semantics means that the update is not dynamic: The program must stop and restart in the new version, losing all transient program state in the process. DSU improves this scenario in two ways: (1) It does not require any downtime to restart the program, and (2) it transforms the program state to be compatible with the new program version without losing any part of it.

I mention this semantics for completeness and to guide the discussion on the other alternatives.

Immediate

The simplicity of the very strict *stop-the-world* approach that offline semantics provide is enticing: Before the update, the program is executing the old version; after the update, the program is executing the new version. Immediate updates provide a similar semantics by performing DSU in three steps: (1) Pause the program, (2) transform the program state, and (3) resume the program in the new version.

Similarly to offline updates, this semantics is clear and simple: The updated program can see only the transformed program state. Developers writing the new program do not need to consider the possibility of new code accessing old program state.

Immediate updates still impose some update-related downtime because they do not execute the program while transforming its state. Their main advantage, when compared to offline updates, is that immediate updates do not not lose any portion of the program state between versions. Instead they transform it to be compatible with the new program. The state transformation logic operates between the two versions and each DSU solution must specify exactly what the transformation code can access (e.g. both versions of the program state without any restriction, or the outdated program state and just a portion of the updated program state).

Lazy

Immediate update semantics is the logical step ahead of offline semantics, pausing the program execution instead of stopping the whole program. It transforms the program state while the program is paused. It is impossible for new code to access outdated program state. Immediate update semantics thus requires a pause that is proportional to the program state. Given that the program state can be arbitrarily large, immediate semantics has the potential to pause the program for an arbitrarily long period.

Transforming all the program state at update time is a very conservative approach. In fact, a DSU does not need to transform all the program state before starting to execute the new program code. It only needs to ensure that the new code can never access outdated program state. Therefore, DSU systems can delay transforming the program state until the updated program tries to access it for the first time after the update. This is exactly what lazy semantics does.

Lazy update semantics amortizes the pause required to transform the program state over the normal execution of the updated program, thus trading steady state performance for short update-induced pauses. Gharaibeh et al. [GRC11] discuss when to prefer lazy over immediate semantics. Figure 2.2, adapted from their work, plots the perceived value over time of a program given the level of service it provides. The shaded area represents the difference in perceived value of an application. The authors also assume that supporting lazy updates imposes a constant performance overhead on steady-state.

After an update, lazy semantics require the normal execution of the new code to be interleaved with the execution of state transformation logic. Updates performed with lazy semantics thus provide a lower level of service immediately after the update. However, lazy semantics allow updates to be installed more readily, without a large period of zero value due to service unavailability. Operators should deploy lazy updates when the dark shaded area in the plot is greater than the light shaded area.

Lazy update semantics do not offer the same clear division between program versions that offline and immediate semantics do. At version n, a program updated through lazy updates can have up to n portions of its state in different versions. Some systems leave that complexity level for the developer to handle [Ste90, BPN08]; other systems provide means to enforce that the program always accesses its state in the appropriate version.

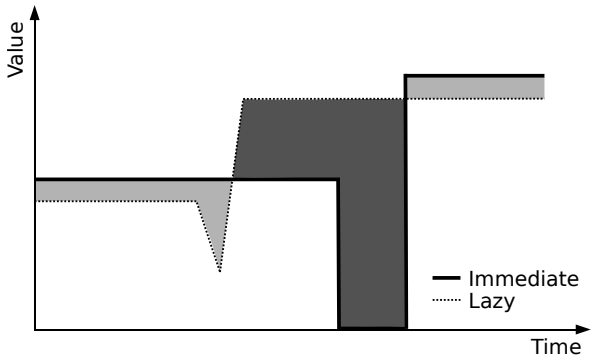


Figure 2.2: Value of an updatable application over time, adapted from Gharaibeh et al. [GRC11]. This plot compares two types of state transformation semantics: Immediate and lazy. The shaded area represents the difference in perceived value of an application. Lazy state transformation adds more steady state overhead than immediate state conversion. It should be preferred when the dark shaded area is greater than the light shaded area.

2.2 Formal Approaches

The ability to change the behaviour of a program while it keeps executing is a powerful tool. Several researchers have studied how to reason accurately about the properties of DSU by formalizing the update model. In particular, the formal approaches focus on the goal of correctness. This section describes those approaches.

2.2.1 General Update Correctness

A possible way of ensuring the correctness of DSU is to state a property about the end-to-end behavior of each update that, when observed, ensures that the update can only be correct. One such property was proposed by Kramer and Magee [KM90]: An update is correct if the updated program preserves all the behaviors of the old program. Bloom and day [BD93] observed that, while intuitive, this correctness property is too restrictive because it rules out updates that fix bugs or add new features.

These first attempts to define an end-to-end general property for update correctness were made in the context of updating a distributed system, without a formal framework. Gupta et al. [GJB96] were the first to introduce a formal framework to reason about update correctness. Their approach considers updates to be correct according to *reachability*: After the update, the updated program eventually reaches some state of the new program.

The authors consider the following model: A running process P has program code Π and state s. An update has new program code Π' and a state mapping function S such that $s' = S(s)$ means that the function transforms the existing program state s to an equivalent version s' that is compatible with the new code Π'. A dynamic update is thus equivalent to stopping P in state s, replacing Π by Π', and resuming P from state s'.

The authors introduce the *update validity property* as follows: A DSU in process P from Π to Π' in state s and using the state mapping function S is valid if and only if P is guaranteed to reach a reachable state of Π' after the change and in a finite amount of time. They, thus, allow a program to behave arbitrarily during a transition period after performing a DSU. However, the new program must eventually behave as if it had been executing from the start.

Figure 2.3 illustrates, informally, the validity property. The top part shows a program updated from version $n-1$ to version n, and then from version n to version $n+1$. Let us consider the validity of the latter update. It is valid if and only if there is a point in time past at which we cannot distinguish the

top part of Figure 2.3 from the bottom part, in which version n never executed. That is, performing a DSU from version n to version $n+1$ is guaranteed to reach a state that is reachable by running version $n+1$ from the start.

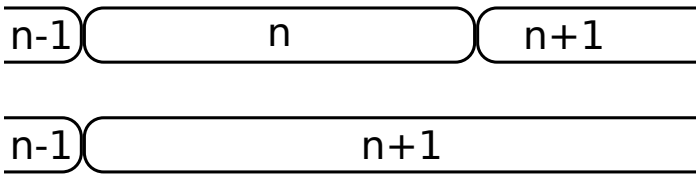


Figure 2.3: Informal illustration of the validity property. The top part shows an update that installs version $n+1$ from n. This update is valid if and only if the program reaches a state after the update that is undistinguishable from the bottom part, in which version n never executed. Note that there is no need to provide a state-mapping function S such that $S(s_{n-1}) = s_{n+1}$ for the bottom half; valid updates ensure this property automatically from both updates on the top half.

When proposing the update validity property, the authors also prove determining whether a given arbitrary update is valid is *undecidable* in the general case. They provide sufficient conditions for ensuring update validity on a procedural language.

Update validity is an intuitive way of reasoning about updates. It captures the notion that the updated program must adopt the behavior of its new code and allows for a bounded period of post-update behavioral divergence that can be used to transform the state between versions.

Unfortunately, as Hayden et al. [HMH+12] point out, update validity is a loose correctness condition that can be both too permissive and too restrictive. They give the example of a version of an FTP server that introduces a feature that limits the number of connections. Performing a DSU to install this version on a server that already has more connections than the limit allows is problematic. We expect a DSU to preserve all active connections. But, in this case, doing so violates validity: The connections, which already exceed the limit when the update takes place, may remain open forever and thus prevent the server from ever entering a reachable state. The alternative is to drop all connections when performing the DSU. Doing so, however, defeats the purpose of DSU in the first place: If we are willing to drop existing connections, we could just restart the server.

However, update validity has important shortcomings as Hayden et al. [HMH+12] discuss. Consider that an update to a server program that adds a limit to the maximum number of connected clients. What happens when this update is performed while the server has more clients connected than the maximum allows? On one hand, allowing those clients to remain connected violates validity because the clients may remain connected for an indefinite amount of time. On the other hand, forcefully terminating client connections at update time does not violate validity but defeats the purpose of using DSU to avoid losing any part of the program state. Either alternative is not acceptable.

2.2.2 Update Calculus

There is a line of work on formalisms [BHSS03, SHB+07, NHFP08] that explores the idea of adding DSU to the programming language itself, at the level of the type system. This approach allows developers to reason about updates directly as a language feature, and compilers to reason about the effects of an update in terms of type-safety between successive versions. Type-safe updates are useful to reason about the overall correctness of a DSU. In this section, I present these formalisms.

The first contribution towards an update calculus was made by Bierman et al. [BHSS03] by extending the first-order simply-typed lambda-calculus with mutually-recursive modules and a primitive for updating them. Their calculus allows to update any module, including changes to types and their definitions. Updates are correct as long as the updated program does not get stuck, i.e., it becomes unable to apply any reduction rule after the update. The update calculus provides an update primitive that lists update points, and the type system accepts only updates that are correct.

Proteus [SHB+07] is the next work in this line. Proteus is a program calculus that supports DSU on procedural, C-like languages. Programs in Proteus consist of functions, data definitions, and definitions of named types. The developers label program points at which an update can occur. Dynamic updates can add/replace types and definitions, and change functions (even while they are active). The developer can provide type transformer functions to transform the program state to be compatible with the new program definition.

The authors consider updates to be correct in Proteus if they satisfy the *con-freeness* property: For each update point and for all types t that an update changes, the program does not use t concretely after the update point. The representation of t can thus safely change. The authors show how this property can be enforced dynamically, at update time. The authors also show how to enforce con-freeness statically, which means that it is possible to annotate every program point with the set of types that can be modified by a correct DSU performed at that point.

Limiting the program points at which updates can happen creates a tension between correctness and timeliness: If these points are not frequent enough, an update may be postponed for a long time until the program finally reaches a point in which the update can take place. Multi-threaded programs only exacerbate this tension by requiring all threads to reach such a program point and wait there until all other threads do the same.

Proteus-tx [NHFP08] addresses the tension between update timeliness and correctness. Proteus-tx extends Proteus with support for multi-threaded programs. It considers that updatable programs are structured around transactions and it proposes a new property for update correctness, called *transactional version consistency (TVC)*: Transactions execute entirely in the same program version, even if an update takes place in the middle of the transaction.

A simple solution to ensure TVC is to not start new transactions when an update is ready, wait for all active transactions to finish, and only then perform the update while no transaction is executing. The authors consider this approach to be overly restrictive. For instance, let us consider the example shown in Figure 2.4. It shows a transaction that calls functions `f` and `g` in this sequence, and an update that becomes available in between calling each function. Updating the program when the transaction finishes clearly satisfies TVC. However, consider the following scenarios:

1. **The update changes only `f`.** Performing the update mid-transaction makes the whole transaction use the old program version, which *satisfies TVC*;

2. **The update changes only `g`.** Performing the update mid-transaction makes the whole transaction use the new program version, which *satisfies TVC*;

3. **The update changes both `f` and `g`.** Performing the update mid-transaction makes the half of the transaction use the old program version and another half use the new program version, which *violates TVC*.

Updates can thus be performed mid-transaction in cases 1 and 2. The challenge here is reasoning about the past and future behavior of active transactions. Proteus-tx solves this problem with *contextual effects*.

```
1 f();
2 // Executing here when update becomes available
3 g();
4 // Update point
```

Figure 2.4: Example of transactional version consistency. All the code in this example is inside a transaction. There is an implicit update point after the transaction finishes, in line 4. Consider that the program is executing line 2 when an update becomes available. Depending on which functions the update changes — `f`, `g`, or both — it can take place immediately — if it changes *either* `f` or `g` — or only after the transaction finishes — if it changes *both* `f` and `g`.

Recent work questions the tension between timeliness and correctness. By studying updates made to six real-world multi-threaded C applications with Kitsune (see Section 2.3), Hayden et al. [HSHF12] show that carefully placed manual update points provide enough update opportunities for all applications to perform updates with acceptable delays, even for applications with soft real-time requirements such as media streaming servers (most below 1ms, with a maximum of 100ms).

Hayden et al. [HMH+12] propose a novel way to reason formally about the correctness of dynamic updates. Based on the Proteus calculus, the authors provide a *program merging semantics* that merges two successive versions of the same program, together with the program-state transformation code that executes in-between. The authors also use *Client-Oriented Specifications (CO-specs)*, which are small programs that state invariants that the update process must always maintain. CO-specs are similar to system tests. The authors then used the CO-specs to verify the merged programs with well known analysis for regular programs (the verification tool Thor [MTLT08] and the symbolic executor Otter [RSM+10]).

2.2.3 Update Modularity Conditions

Boyapati et al. [BLS+03b] describe how to perform dynamic updates to persistent object stores. Client applications of the object store mutate the state of the store inside *application transactions*.

Their system uses *transform functions* [Zic91, FMZ+95] for each updated class whose objects need to be transformed. A transform function is a way of implementing the state transformation logic. For each object that belongs to the outdated class, the update system creates a blank object that belongs to the updated class and invokes a transform function to initialize it using the outdated instance. When the transform function returns, the updated object will take the place of the outdated one. Each transform function runs inside its own *conversion transaction*.

The authors consider transform functions to be *well-behaved* if they access only the object being transformed and the objects it encapsulates. An object A *encapsulates* another object B if objects not encapsulated by A cannot access object B. For instance, consider a stack implemented as a linked-list. The stack encapsulates the nodes that implement the linked-list if no other object outside the stack and the nodes themselves cannot access any node.

It is obvious that well-behaved transform functions can only find the old version of the objects they access. Given that they encapsulate all objects that they use, it is not possible for those objects to be updated before the transform function updates the encapsulating object (e.g. it is not possible for a list node to be updated before the head of the list).

Considering that all transform functions are well-behaved greatly simplifies reasoning about the transformation logic. The authors thus propose a formal *modularity property* for updates that captures the intuition that all transform functions are well-behaved.

The authors present and formalize a set of *update modularity conditions* that ensure that the update system satisfies the modularity property when performing lazy updates. The update modularity conditions state how to order application transactions with conversion transactions and conversion transactions among themselves.

Of course, if all transform functions are well-behaved, the system does not need to enforce any particular serialization order. The authors propose the use of *ownership types* [CPN98, BLS03a] to allow the compiler to infer and enforce encapsulation properties. When encapsulation fails (e.g. in mutually referent objects and circular data structures), the authors propose to use *triggers*: If two objects A and B do not encapsulate each other, there may be a third object C that encapsulates both A and B. Developers can set up a trigger to force the conversion of C before converting either A or B. The transform function of C thus satisfies the modularity property when accessing objects A and B.

When it is not possible to find an encapsulating object, the authors propose to use *versions*: After transforming an object O_0 to object O_1, the update system does not discard O_0. Then, when converting another object with a transform function that needs to access O_0, the update system can still provide it and thus ensure the update modularity property. However, the authors require developers to specify manually where versions are needed and do not discuss when the old versions can be safely discarded.

2.3 Compiled Imperative Languages — C

The C programming language [KR88] is an imperative procedural programming language. C programs are compiled to executable binary code that runs natively on the hardware. Programs written in C have a fine grained level of control over their performance and memory usage.

There is a vast amount of programs written in C that provide the non-stop services that serve as the backbone of most of today's infrastructure. DSU systems designed for C have an unique set of challenges to solve. Performing DSU on programs written in C is thus a pressing research problem.

This section presents and classifies the following dynamic software update systems that target general applications written in C: OPUS [ABBS05], Ginseng [NHSO06, SHB+07, NHFP08, NH09], POLUS [CYC+07], UpStare [MB09], Ekiden [HSHF11], Kitsune [HSD+12], and DynSec [PBG13].

Besides being used to write general applications, the C programming language is also used to write operating system kernels. There is some overlap between DSU systems for operating systems and for applications written in C. I shall, therefore, also consider the following DSU systems that target updating operating systems without requiring the machine to reboot: K42 [BHA+05], LUCOS [CCZ+06], DynAMOS [MR07], KSplice [AK09], and PROTEOS [GKT13]. LUCOS is essentially a version of POLUS that uses VMMs to effect changes in operating systems, all comments made about POLUS also apply to LUCOS.

Table 2.2 summarizes the design decisions that each DSU system for applications and OS kernels take. It also gives an overview of the rest of this section.

2.3.1 Update Preparation Methodology

Most solutions require the developer to prepare a program version as an update before performing it as a DSU on a running program. Each solution provides a set of tools that allows the developer to prepare an update. While preparing an update, the update preparation tools get the chance to analyze the update for correctness, generate stub code for the program-state migration, and pack everything into a deployable update. Figure 2.5 shows the process for each DSU system.

Target	System	Timing	Code Updates	Data Updates	Update Semantics	State Transf.
Apps	OPUS	Quiescence	Trampoline	—	—	—
	DynSec	Unrestricted	Trampoline	—	—	—
	POLUS	Unrestricted	Trampoline	Struct Replacement	Immediate	Assisted
	UpStare	Manual	Replacement	Struct Replacement	Immediate	Assisted
	Ginseng	Manual*	Indirection	Type Wrapping	Lazy	Assisted
	Kitsune	Manual	Replacement	Struct Replacement	Immediate	Assisted
	Ekiden	Manual	Replacement	Struct Replacement	Immediate	Indirect
Kernels	LUCOS	Quiescence	Trampoline	Struct Replacement	Immediate	Assisted
	DynaMOS	Quiescence*	Trampoline	Shadow structure	Immediate	—
	KSplice	Quiescence	Trampoline	Shadow structure	Immediate	Assisted
	K42	Quiescence	Indirection	Shadow structure	Immediate	—
	PROTEOS	Manual*	Replacement	Struct Replacement	Immediate	Assisted

Table 2.2: Systems that support DSU for applications and OS kernels written in C. This table summarizes the discussion on this section. Rows titled timing, update semantics, and state transformation use the classification introduced in Section 2.1. The remaining rows, code updates and data updates, describe how these systems implement updating loaded code and existing data and are described in detail in Sections 2.3.3 and 2.3.4, respectively. Some timing decisions are marked with an asterisk: Ginseng can automatically detect extra update points, DynaMOS can update some functions that never quiesce, and PROTEOS requires the developer to specify when updates cannot happen instead of when they can happen.

Each update system provides either a source-to-source *patch generator* or a *custom compiler*. These tools, implemented using CIL [NMRW02] or LLVM [LA04], typically perform the following tasks:

1. Rewrite the code so that it can be updated in the future;
2. Generate meta-data that describes the current version, so that future versions can be compared against it;
3. Identify the portion of the program that changed between versions, and extract that portion of code as a self-contained patch to be applied.

Figure 2.5 shows the toolchain that each update system supports. The update systems not shown (DynSec, DynAMOS, K42, and PROTEOS) require the developer to prepare patches manually and do not provide tools to help in the process. KSplice is the notable exception in this figure; I shall discuss how it prepares updates later in this section.

Most systems follow a *dynamic patching* approach to DSU: UpStare, POLUS, OPUS, Ginseng, and KSplice. The prepared update is a patch that contains the changed code and information about how to weave it into the running process. Ekiden and Kitsune follow a *whole-program* approach, in which they completely replace the code of the program with new code and transfer the program state between code versions, transforming it in the process.

UpStare, POLUS, and Ginseng use a *patch generator* that extracts the patch as source code. The patch generator also generates stub source code for the program state transformation that the developer can customize before compiling the dynamic patch. Out of these systems, only POLUS generates patches that can be processed by the standard compiler. All the other systems require non-standard compilers to process annotations and meta-data for each patch.

OPUS and Ekiden require the developer to write the state transformation logic from scratch. OPUS provides a *patch analyzer* that can warn the developer about potential errors on the state transformation code, and generate meta-data about the current version for the custom compiler and future updates to use.

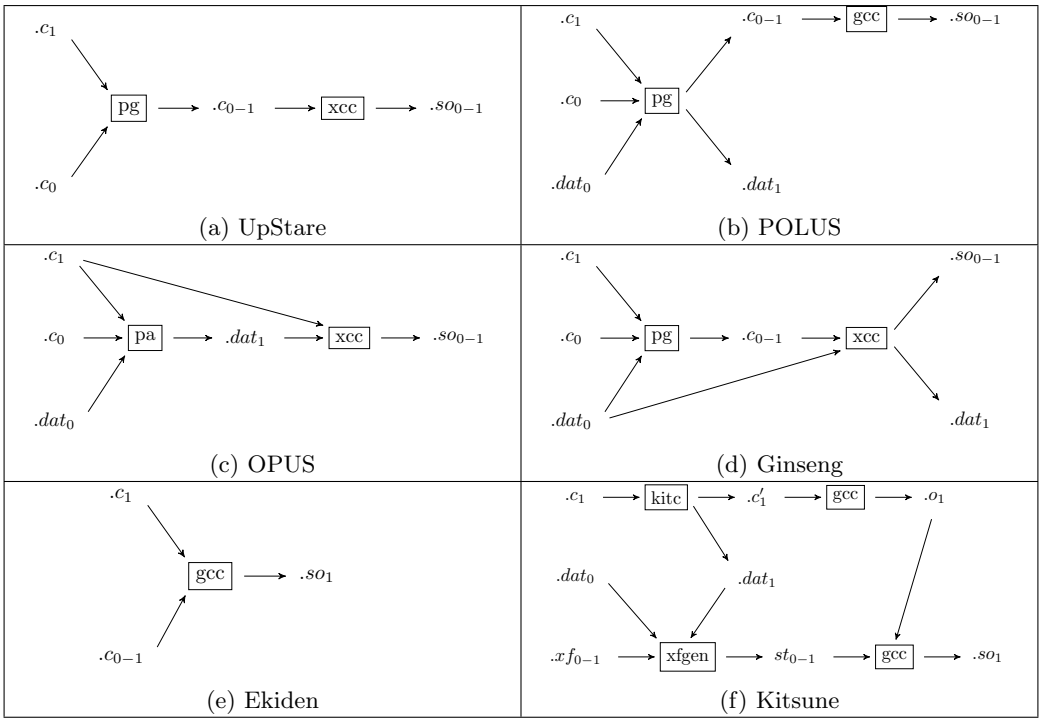


Figure 2.5: Method to prepare updates for several systems. Upstare, POLUS, OPUS, and Ginseng generate dynamic patches to be applied on the running process; Ekiden and Kitsune generate new program versions to replace the current one in execution (whole-program updates). Tools have a solid border around them: **pg** stands for *patch generator*, **pa** for *patch analyzer*, **gcc** for *standard compiler*, **xcc** for *custom compiler*. Other items represent artifacts: **.c** stands for *source files*, **.dat** for *update meta-data*, and **.so** for *binary patches*. The subscript shows the version for which each artifact is: *0* for the old program, *1* for the new program, and *0–1* for the patch/update. Kitsune uses its own tools, discussed in the main text.

Kitsune separates the task of writing the new program code from the task of writing the transformation logic. The new program is processed by a source-to-source translator — *kitc* — that prepares it to be updatable or installed as an update. The developer specifies the state transformation logic through a domain-specific language (DSL) implemented by an automated tool — *xfgen* — that generates the source code for *state transformers* — st_{0-1} — according to the input — $.xf_{0-1}$. Finally, the program is compiled using the standard compiler, and then linked with the state transformation logic to yield the binary update file.

KSplice operates directly on the compiled kernel. This creates problems when detecting exactly what changed between versions due to compiler interference (e.g. different inlining decisions in different versions). KSplice, thus, uses two kernel builds: One with the original kernel — the kernel that is currently executing — and another with the original kernel patched with the update — the kernel that we want to execute without rebooting. This allows KSplice to detect changes introduced by the compiler, and list symbols that are resolved only during runtime. When updating, KSplice checks that the kernel currently in execution matches the expected kernel version, and then resolves all dynamic symbols left unresolved on previous stages using the symbol table that the running kernel keeps.

2.3.2 Timing

Once the update is prepared, we can perform it as a DSU on the running program. To do so, each solution must pause the program at a point where it is safe to perform the update. In the following, I describe how each DSU system pauses the running program.

Unrestricted

POLUS performs updates at the level of function calls and can update live functions. When the update process begins, POLUS pauses all threads where they happen to be. It then inspects the stack of each thread and modifies return addresses of changed functions, making them refer to the new code. As a consequence, a program can execute old and new code at the same time. After POLUS releases all threads, it intercepts write attempts by old code, translating them to proper writes to the new program state using *state synchronization functions* defined by the developer.

DynSec performs updates at the level of basic-blocks. When loading a binary application, it uses a Dynamic Binary Translator (DBT) to rewrite the binary before executing it. When doing so, it adds an update point to the end of each basic block.

Quiescence

OPUS, KSplice, and K42 stop each thread when all functions that the update modifies are not active. DynaMOS employs a similar activeness-checking approach. However, it can update functions that are always active but sleep for long periods of time inside a loop. Typically, such functions are executed by dedicated threads that follow a similar pattern: Sleep until an event that requires action awakes them, execute the loop until the event is processed, and go back to sleep to wait for the next event. DynaMOS updates these functions by injecting code after each sleep invocation that jumps to the new version of that loop if an update took place while the thread was sleeping.

Manually Identified Update Points

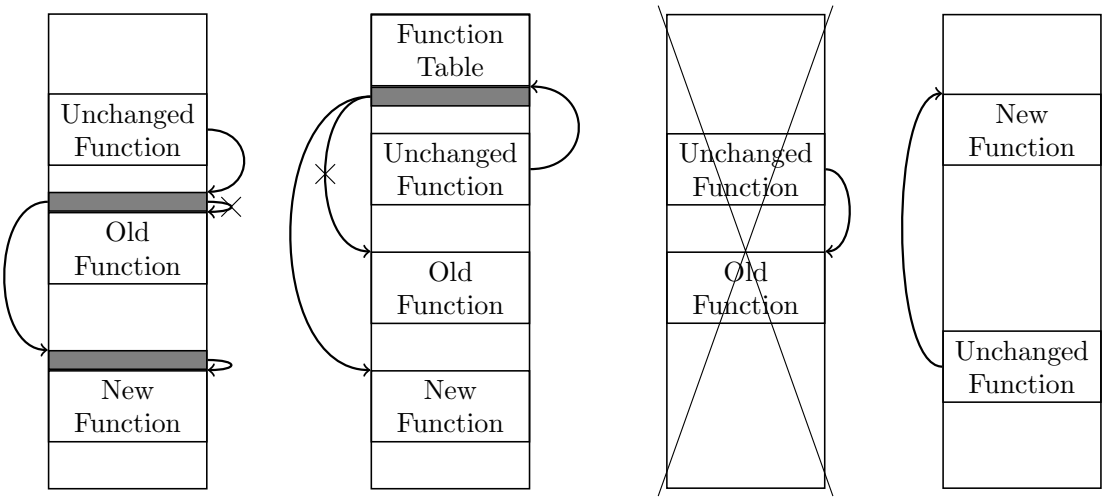
Kitsune, UpStare, Ginseng, and Ekiden allow updates to happen in active code at developer-defined program points (UpStare may also determine these points automatically).

STUMP [NH09] extends Ginseng to automatically expand the available update points besides the ones that the developer manually identifies. If follows a intuition similar to Proteus-tx, presented in Section 2.2.2, to generate a set of *induced update points* that are proved to be equivalent to the manually identified update points using *version-consistent execution traces*, which are computed through contextual effects.

PROTEOS requires the developers to state when updates cannot happen, thus taking a blacklist approach. PROTEOS requires the developer to provide a set of *state filters* to specify constraints on the program state. When an update becomes available, PROTEOS performs it at any point in the execution of the program that does not violate any of the state filters.

2.3.3 Code Updates

The mechanism through which DSU solutions support changing the code of a running program is an important challenge. This is especially true for the C programming language because it is compiled to native code. Mechanisms for updating the running code on a C program can thus be re-used, internally, by other techniques that provide DSU at a higher level of abstraction. Figure 2.6 summarizes the possible alternatives to support code updates; I shall describe them in detail in the rest of this section.



(a) Trampolines (b) Extra level of indirection (c) Whole-program update

Figure 2.6: Different approaches to support updating code on when performing DSU on C programs. Each box represents a program composed by functions. The shaded area represents a trampoline in 2.6a and an entry in the function table in 2.6b. Values overridden by an update are crossed-out.

The first option to update already loaded code is through *trampolines*, shown in Figure 2.6a. When preparing the update, the DSU system prepends a dummy jump at the start of each function called a trampoline. At update time, the DSU system redirects that jump on each updated function to its new version. This technique can also be used without pre-reserving the space for the trampoline and simply overwriting the first few bytes on the code of the outdated function, at the expense of not being able to execute the old version of updated functions after the update. OPUS, KSplice, POLUS, DynaMOS, and DynSec update the code through trampolines.

Another option is to rewrite the original program to add an *extra level of indirection* to each function call, shown in Figure 2.6b. When preparing the update, the DSU system creates a function table and rewrites every direct function call as an indirect call through the function table. At update time, the DSU system updates the function table so that the entries for the updated functions refer the new version of the code. Ginseng and K42 follow this approach.

The remaining option is through *whole-program update* when performing a DSU, shown in Figure 2.6c. The DSU system loads the new program code as a whole, instead of a collection of new functions, and jumps from the current point in the old program to an equivalent point in the new program. UpStare, Ekiden, PROTEOS, and Kitsune follow this approach.

Whole-program updates create the subproblem of matching the old program code with the new one. PROTEOS replaces a process with a new one and rebinds all the end-points of the original process. UpStare performs stack-reconstruction on the updated program, unwinding the old stack and rewinding it into the new stack, one stack frame at the time.

Kitsune and Ekiden require the developer to match both versions manually through *control-flow migration*. The idea is to restart the program from the `main` function, and each thread from its starting function, and allow it to rebuild its stack through developer annotations. Figure 2.7 shows a very simple example. When running for the first time, function `main` parses its arguments. When running after an update, function `main` skips parsing its arguments and calls the `process` function. The call to function `dsu_updating` in line 3 is a part of the control-flow migration code added by the developer.

```
1  int main(int argc, char ** argv) {
2
3    if (!dsu_updating())
4      parse_arguments(argc, argv);
5
6    process();
7  }
```

Figure 2.7: Simple example of control-flow migration on a C program. Function `dsu_updating` returns `false` when the program is running for the first time, `true` when the program is being restarted due to an update. The developer can use it to adapt the behavior of the program after an update and thus rebuild the stack on the new program version.

(a) Shadow data structure (b) Type wrapping (c) Struct replacement

Figure 2.8: Different approaches to support updating data on when performing DSU on C programs. Each box represents a struct in memory. Shaded areas represent modifications to the original structure to support DSU. Type wrapping (2.8b) and shadow data structure (2.8a) update each data structure in-place; struct replacement (2.8c) replaces existing structures with their updated version.

Each alternative to updating the program code represents a different compromise. Trampolines are easy to implement but require a writable code segment, which makes the application vulnerable to code injection attacks. Adding an extra level of indirection centralizes the decision of which code should run for each function. However, it requires a non-trivial program transformation and adds constant overhead during steady-state execution due to the indirect function calls. Neither of these two approaches supports updating programs compiled with function inlining (the inlined functions cannot be updated); updating programs that jump to the middle of functions, which is a valid C idiom; or updating programs with functions that never (or rarely) exit, such as `main` or functions with event-processing loops.

Whole program updating deals directly with all the downsides of the other two approaches: The code segment is never writable; it naturally supports function inlining (and other compiler optimizations that require code mobility) and code that jumps to the middle of functions; and can update active code. The challenge is how to map old program points to equivalent points in the new program, and most systems that follow this approach require developer annotations to solve this problem.

2.3.4 Data Updates

C programs keep their data anchored on a set of globally accessible variables and local variables of functions that are always active, such as `main`. Their data is structured as a collection of *structs* with pointers to other structs that compose a graph. When the definition of a struct type changes between versions, the DSU system must transform the existing structs according to the state transformation logic. There are several options to perform this transformation. Figure 2.8 summarizes the possible alternatives to support data updates; I shall describe them in detail in the rest of this section.

The simplest way is to do so *in-place*. This naturally supports pointers to structs that are spread throughout the application. However, the new structure may require a larger memory representation than the old structure. To deal with that case, DSU systems for C use either *shadow data structures* or *type wrapping*.

Shadow data structures, shown in Figure 2.8a, are extensions to the original structs that have the extra fields that do not fit in the original struct. When preparing the initial version, the DSU system adds a pointer to a shadow structure to the end of every updatable struct. When preparing each update, the DSU system rewrites the code that accesses the new fields to use the shadow instead. This approach adds performance overhead when accessing the new fields due to the indirect access. Given that programs written in C require manual memory management, this option has the added challenge of managing the lifetime of the shadow together with the struct that refers to it. KSplice, DynaMOS, and DynSec use shadow data structures.

Type wrapping, shown in Figure 2.8b, is the alternative to shadow data structures for in-place data updates. When preparing the initial version, the DSU system adds some unused padding to the end of each struct so that it can grow in future updates. This option adds memory overhead to steady-state execution and imposes a hard limit on the maximum size of future versions of each struct. It also breaks programming idioms that rely on the memory alignment and size of structs. Ginseng uses type wrapping.

The alternative to in-place data updates is *struct replacement*, shown in Figure 2.8c. The DSU does not change the original representation of each struct in any way. When performing an update and transforming the program state, the DSU system moves the struct to a new memory location, transforming the struct in the process according to the state transformation logic. Relocating the structure provides the opportunity to reserve more memory for its new version. However, this technique requires the DSU system to find all the pointers in the program state that refer to the old struct, and update them to refer to its new version. Both shadow data structures and type wrapping add steady-state overhead due to reduced cache locality; struct replacement does not add any overhead. It also supports all programming idioms in C and can grow structs without limits. Kitsune, K42, POLUS, Ekiden, UpStare, and PROTEOS use struct replacement.

OPUS and DynSec do not support any type of data migration between versions.

2.3.5 State Transformation and Semantics

The previous section described how each DSU system supports transforming the program state at update time. This section describes how the developer can specify the program state transformation logic.

Of all the systems that support program state transformation (all but OPUS and DynSec), Ginseng is the only one that supports *lazy update semantics*. It rewrites the program to add calls to mediator functions to access updatable structs. These functions then transform outdated structs after an update takes place, as the natural control flow of the new program reaches each struct for the first time. All the others support *immediate update semantics*.

Most update systems for C support assisted manual state transformation. Figure 2.5 shows that the patch generator for UpStare, POLUS, and Ginseng generates source code. In fact, the generated code has stub transformation code that the developer can customize. UpStare and Ginseng automatically copy unchanged program state; the developer just has to specify how to transform the remainder. UpStare has the richest state transformation model: It provides support for the developer to convert all program state, including stack frames, program counters for each thread, and global variables in the heap. POLUS provides the most complex state transformation support. Given that it can execute code from different versions simultaneously, it keeps N copies of the same data for each of the N versions in execution. The developer must keep all versions consistent through a callback mechanism.

Kitsune also provides support for assisted manual state transformation. The developer specifies the state transformation logic using a separate simple DSL. Kitsune provides a tool called *xfgen* to generate source code from the DSL that transfers the state between versions, copying the state that did not change and transforming the state that did change by following the logic that the developer specified.

KSplice supports assisted state transformation. It provides macros to annotate code that must run during the update, while all threads are stopped. The developer can use these macros to annotate functions that run inside the kernel to find and transform outdated kernel data.

PROTEOS also supports assisted state transformation. At update time, it transfers the state from the old process and transforms it to be compatible with the new process. The state transfer is mostly automatic, but the developer can define a set of callbacks that are evaluated every time PROTEOS transfers the intended type.

The authors of DynAMOS and K42 do not discuss how to specify the state transformation logic.

Ekiden supports indirect state transfer, without providing any automated developer assistance. It transfers the state between both versions by requiring the old version to serialize its state and then deserializing it in the new version.

2.3.6 Discussion

This section described the state of the art in DSU systems for the C language and for OS kernels. I now explain its respective rows on Table 2.1, page 12, by relating each DSU system with the goals that I introduced earlier in Section 1.2.

Effectiveness

Some of the systems that I presented in this section target the kernel of operating systems and are not applicable to general applications written in C: K42, LUCOS, DynaMOS, KSplice, and PROTEOS. These solutions target a specific type of C application, with a very specific architecture. This affects their effectiveness — ◓ . Still, KSplice and PROTEOS were implemented for the Linux and MINIX kernels, respectively, which are complex and mature OS kernels used in practice — ◕ .

DynSec, Ekiden, and Ginseng cannot be considered fully flexible — ◕ . DynSec requires the target program to execute inside a sandboxed environment, which limits its effectiveness. Ekiden requires the developer to write code that serializes the program state of each program version and then reads the serialized state in the updated program version, and it does not provide any tool to help the developer in doing so. Ginseng uses type wrapping to support data updates, which breaks memory alignment and forbids valid popular C idioms, and requires the updatable program to be structured in such a way that satisfies its conservative static analysis.

The remaining systems require a special toolchain to build the original C program so that it can be either updated or applied as an update. Section 2.3.1 describes how to prepare an update for every system. While each system has a different toolchain, these are similar and can be automated in the process that builds each release. These DSU solutions are thus effective — ● .

Efficiency

OPUS, POLUS, UpStare, DynSec, LUCOS, and DynaMOS add performance overhead during steady-state execution, i.e., when not performing an update — ○ . Some of the systems transform the program using a source-to-source translator. The transformed program supports DSU but executes slower than the original (UpStare and Ginseng). Other systems require some compiler optimizations to be disabled (POLUS, OPUS, and LUCOS). DynSec requires that the program executes inside a sandbox that adds performance overhead.

Ekiden, Kitsune, K42, KSplice, and PROTEOS introduce a negligible amount of steady-state overhead. However, they introduce a long pause in the execution of the original application when performing a DSU — ◓ . Ginseng, which does introduce steady-state overhead, supports lazy program state transformation between versions. This means that Ginseng does not introduce a long update-induced pause — ◓ .

Flexibility

OPUS, LUCOS, and DynSec are not flexible because they do not support any form of program state transformation — ○ . All the other systems provide some mechanism for transforming the program state between versions. Ginseng, however, imposes a hard limit on the maximum size that a structure can ever take, which limits its flexibility — ◐ .

POLUS and PROTEOS require the code that the update changes to quiesce before performing a DSU, but they do not support transforming the local variables of functions active at update time. They, thus, assume that the program state kept in the local variables of functions active at update time does not need to be transformed. This assumption might not be accurate: Consider an data structure kept in a local variable of function `main` that is always passed as argument to other functions. If an update changes the representation of this data structure, these systems are not able to perform such an update correctly. POLUS also requires updates to be backwards compatible — ◓ . PROTEOS supports non-backwards compatible updates — ◐ .

Update systems that target an OS kernel and require quiescence perform updates only when no process is performing a system call. The only active code corresponds to long-running threads inside the kernel itself. KSplice and DynaMOS provide support for redirecting these threads at update time, which involves transferring any state they keep — ● . K42 is structured around objects, that are accessible through a globally accessible table. Local variables thus refer to objects that the state transformation can access at update time — ● .

UpStare allows the developer to migrate all the program state at update time — ● . This includes stack frames and program counters per thread. Kitsune follows a different approach: It supports migrating local variables between versions, and requires the developer to add control-flow migration code to rebuild the stack per thread after the update, effectively migrating it — ● . Ekiden uses a similar approach.

Correctness

All the systems that I presented in this section allow the developer to perform erroneous updates. This is true in particular for all the systems that rely on quiescence for correctness: OPUS, K42, LUCOS, DynaMOS, and KSplice — ◔ . DynSec and POLUS offer even weaker correctness guarantees due to their update timing — ○ . All these approaches are inherently unsafe, as I discussed in Section 2.1.1.

UpStare, Ekiden, Kitsune, and PROTEOS only allow updates at program points that the developer annotates as safe. They do not provide any way for the developer to assess the correctness of the update process or the updated program — ◓ .

Ginseng is the only system that allows the developer to assess the correctness of updates performed through it — ● . Even though it does not forbid developers from writing erroneous updates, later work uses systematic testing [HSH+12] and static analysis of merged program versions [HMH+12] to assess the correctness of updates performed using Ginseng.

System	Implementation Level	Flexibility	Timing	Update Semantics	State Tansf.
HotSwap	JVM	Methods	Unrestricted	Immediate	—
JRebel	Bytecode	Classes	Unrestricted?	Lazy	Automatic
DCE VM	JVM	Hierarchy	Unrestricted	Immediate	Automatic
DUSC	Bytecode	Classes	Quiescence	Immediate	Assisted
JDrums	JVM	Classes	Quiescence	Lazy	Assisted
DVM	JVM	Classes	Quiescence	Lazy	Automatic
JVolve	JVM	Classes	Quiescence	Immediate	Assisted
JavAdaptor	Bytecode	Hierarchy	Unrestricted?	Immediate	Assisted

Table 2.3: Systems that support DSU for applications written in Java. This table summarizes the discussion on this section. Columns titled *timing*, *update semantics*, and *state transformation* use the classification introduced in Section 2.1. The remaining columns, *implementation level* and *flexibility*, describe the level at which each system is implemented and the types of changes that each system supports, and they are described in detail in Section 2.4.1 and Section 2.4.2, respectively. Some timing decisions are marked with a question mark, meaning that the authors do not discuss this design decision in the original article that describes the DSU system.

2.4 Managed Object-Oriented Languages — Java

Java [GJS96] is a very popular object-oriented programming language, ranking among the top 2 most popular languages for the past 15 years [tio]. However, it does not support DSU directly at the language or runtime level. This is unfortunate because many programs in Java provide non-stop service and would greatly benefit from DSU.

Java programs are typically compiled to a binary format. However, and unlike C programs, the binary is not executable natively by the hardware. Instead, the binary, also called the *bytecode*, is executed by the *Java Virtual Machine (JVM)* [LY99]. Programs written in Java thus execute inside a managed environment. The JVM provides extensive runtime support to Java programs, including a *garbage collector (GC)* that handles memory management and a *just-in-time optimizing compiler (JIT)* that compiles the bytecode down to native code that executes directly on the hardware.

The presence of the GC and the JIT make the problem of supporting Dynamic Software Updates for managed languages such as Java different from compiled languages such as C, described in the previous section. Researchers have already considered the problem of supporting DSU for managed languages, namely Java. In this section, I describe the state-of-the-art on this topic.

This section presents and classifies the following DSU solutions that target applications written in Java: JDrums [RA00], DVM [MPG+00], HotSwap [Orab,Dmi01], DUSC [ORH02], JVolve [SHM09], DCE VM [WWS10], JavAdaptor [PGS+11], and JRebel [KV12], Table 2.3 summarizes the design decisions that each DSU system takes.

Some Java applications provide their functionality by composing several different modules. These applications are structured around a module system that enables module registration and lookup, and provides support for inter-module communication. Examples of such approaches are: The Eclipse Rich Client Platform [ML05], the Netbeans platform [BTW07], and the Open Services Gateway initiative (OSGi) [OSG14]. Such systems typically also support updating loaded modules with a newer definition, which qualifies as a DSU technique. However, given that these approaches are rather based on a particular architectural style, I defer discussing them until Section 2.7.1 (page 55).

Table 2.3 summarizes the design decisions that each DSU system takes. It also gives an overview of the rest of this section.

2.4.1 Implementation Level

Java programs are compiled to bytecode which is then executed by the Java Virtual Machine (JVM). JVM bytecode is a binary representation that is not executable natively on the hardware. The JVM interprets the program, using an optimizing JIT compiler to generate native code for the most frequently executed parts of the Java program. The JVM also provides automatic memory management through a Garbage Collector (GC) that reclaims objects that are not reachable through the transitive closure of a set of root references accessible by the program (references on stack frames and static variables). There are two levels at which a DSU system for Java can be implemented: (1) At the JVM level, and (2) at the bytecode level.[1]

JVM Level

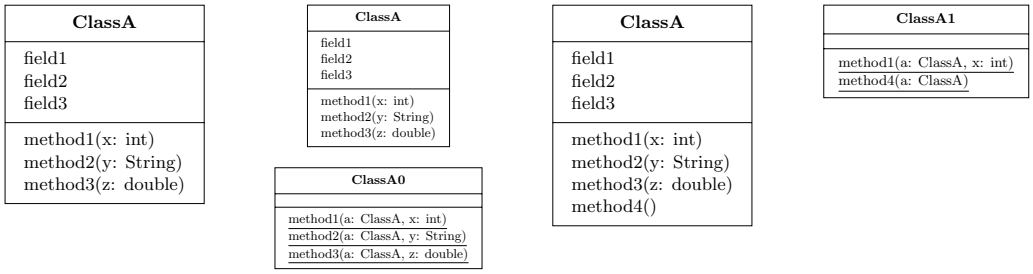
Implementing a DSU system at the JVM level has several advantages: The ability to customize the existing GC to transform the program state when performing a DSU, and the ability to customize the JIT compiler to replace outdated code. JDrums, DVM, HotSwap, JVolve, and the DCE VM are implemented at the JVM level.

Existing GCs are implemented through *mark-and-sweep*, in which the GC follows the transitive closure of reachable (live) instances to determine which instances can be reclaimed. State-of-the-art JVMs also support *generational garbage-collectors*, that copy objects between different generations according to their observed/expected longevity. Garbage-collection algorithms, strategies, and implementation are described in detail elsewhere [JHM11].

By forcing a GC cycle at update time, a DSU system can use the GC to find all the outdated instances. Furthermore, when the GC moves an outdated instance between generations, the DSU system has a great opportunity to transform it. Given that the object is being moved, this approach naturally supports increasing the size of updated objects without requiring shadow structures or type wrapping, as some DSU systems for C do (described in Section 2.3.4). DVM, JVolve, and the DCE VM use a custom GC to transform the state between versions. JDrums modifies the JVM to represent objects internally through an extra level of indirection in the form of an object table. To find and update existing objects, JDrums simply traverses the object table, thus avoiding the need to customize the GC algorithm.

Another advantage of supporting DSU at the JVM level is the ability to install new code for already loaded classes by customizing the JIT compiler to do so. During its normal operation, the JIT compiler may recompile a given method several times, with increasing levels of optimization as the program executes it more frequently. The ability to recompile a method dynamically means that the JIT compiler can also find calls to the outdated compiled method, even inside inlined code, and update them to refer to the newly compiled method instead. In fact, some JIT compilers even support replacing methods that are active on the stack through *on-stack replacement (OSR)* by adjusting return addresses and program-counters on stack frames that refer to the compiled method. The support that JIT compilers provide to replace compiled methods can be adapted to instead replace outdated methods by their new definition after an update. HotSwap, the DCE VM, and JVolve support updating code this way. DVM can only run in interpreted mode, without the JIT compiler enabled. JDrums adds a level of indirection to each method invocation to decide which code to run.

[1] Conceptually, it is possible to implement a DSU system for Java as a source-to-source translator. However, there are no known DSU systems at that level. Also, the bytecode is very similar to the original source and easier to analyze (e.g. all references to fields and methods are fully qualified with the owner of the field/method).



(a) Original initial version

(b) Transformed initial version

(c) Original updated version

(d) Transformed updated version

Figure 2.9: Example of how JRebel transforms an class to be updatable. All the methods on the original class `ClassA` (2.9a) query the JRebels API to find the most recent method to execute located in implementation class `ClassA0` (2.9b). When an update adds method `method4` and changes existing method `method1` (2.9c), JRebel generates a new implementation class `ClassA1` (2.9d) and updates its internal state to delegate method invocation to the new methods. Underlined methods are static.

The great disadvantage of implementing DSU support at the JVM level is portability: DSU systems implemented at the JVM level are tightly coupled with the particular JVM they use. They are thus very hard, if not impossible, to port to a different JVM without a complete rewrite. These systems are therefore hard to maintain, especially if the JVM they support changes the implementation of the GC/JIT compiler between versions.

Bytecode Level

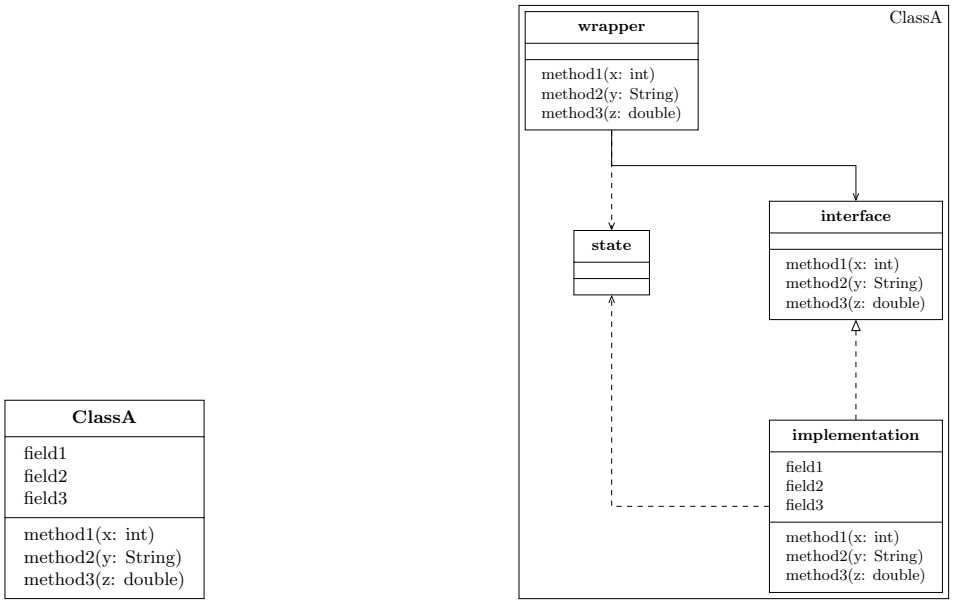
An alternative to support DSU for Java is to rewrite the bytecode program in such a way that the rewritten program has the same semantics as the original program but nevertheless supports DSU. DUSC, JavAdaptor, and JRebel support DSU through bytecode rewriting.

To support performing DSU, each system transforms the original program in a different way. Figure 2.9 shows how JRebel transforms each updatable class. It moves the implementation of all methods out of the original class `ClassA` and into implementation class `ClassA0`. When moving each method to the implementation class, JRebel turns it into a static method that expects the receiver object to be passed as the first argument. Other classes in the program still refer to `ClassA`; each method of this class simply queries the JRebel API to lookup its most recent implementation and then calls the appropriate method. To perform an update, JRebel generates a new implementation class `Class1` with the new method definitions. The authors do not discuss how other classes, which refer to the original (unchanged) `ClassA`, can find new methods added by an update, or how to use this technique to support adding fields.

DUSC also transforms the original program. Figure 2.10 shows how DUSC transforms class `ClassA`. DUSC makes each class C updatable by breaking it into four classes:

1. **Implementation class** C_i, contains the implementation of each version of class C. Each DSU introduces a new implementation class for each modified class;

2. **Wrapper class** C_w, provides the same interface as class C to any client class of C. The implementation of each method delegates on the most recent implementation class. Each implementation class refers to other updatable classes through their wrapper;



(a) Original initial version

(b) Transformed initial version

Figure 2.10: Example of how DUSC transforms a class to be updatable. DUSC breaks the original class `ClassA` (2.10a) into 4 classes (2.10b): The *wrapper*, which other classes refer to; the *implementation*, one per each program version; the *interface*, which the wrapper uses to delegate execution on the implementation; and the *state*, that encodes the state of the outdated implementation to pass it to an updated implementation. The relations between classes, represented as arrows, are as follows: The wrapper *refers to* the interface and *uses* the state when performing an update; the implementation *implements* the interface and *uses* the state when performing an update.

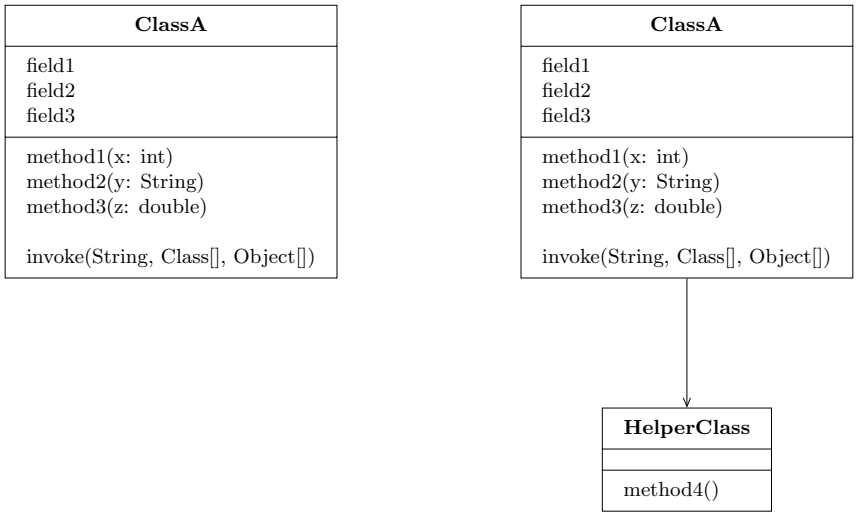
3. **Interface class** C_a, an abstract class that all implementation classes C_i extend. The wrapper class uses interface classes to refer to each implementation class indirectly. This way, when the implementation class changes, the new implementation class is still a subclass of C_a and the wrapper class can still refer to it with the same code;

4. **State class** C_s, encodes the state of an instance of the implementation class, and is used to migrate the state of existing instances of implementation classes to the new program version.

The rest of the approaches that transform the bytecode of the program rely on the presence of HotSwap on the underlying JVM. HotSwap is available on most JVMs since Java version 1.4 [Orab] but it supports a very limited version of DSU that allows only the bodies of existing methods to change.

Kim and Tilevich [KT08] propose a program transformation to support unrestricted changes to Java classes on JVMs that support HotSwap. Figure 2.11 shows how their approach works. They inject method `invoke` on all classes of the initial program version. To support a DSU that adds/changes methods, they move the definition of the new method to an helper class, and adjust the implementation of all client classes, through HotSwap, to call the new/modified method through the `invoke` method. This approach can also support updates that add fields. Still, it does not support updates that change the position of a class in the hierarchy.

JavAdaptor also assumes that the underlying JVM has the ability to update the bodies of methods through HotSwap. It changes the original program to support future changes through *containers* and *proxies*. Figure 2.12 shows an example of how JavAdaptor supports DSU. It transforms the original program by adding a container field to every updatable class (line 7). In this example, an update changes method `averageTemp` (line 2) to `currentTemp` (line 20). JavAdaptor loads the new version of class



(a) Transformed initial version

(b) Transformed updated version

Figure 2.11: Example of how Kim and Tilevich's technique transforms an class to be updatable. The original class is shown in Figure 2.9a and the update is shown in Figure 2.9c. The technique adds method `invoke` to the first program version to redirect calls to methods added in future versions (Figure 2.11a). During the update, this technique creates a new class `HelperClass` to hold the new method `method4` (Figure 2.11b) and rewrites all client classes that call the new method to do so through method `invoke` instead.

`TempSensor`, changing its name to `TempSensor_v2` to avoid name clashes (line 19). It also generates a container class (lines 40–42) and a proxy class (lines 44–46). Finally, it uses HotSwap to change the existing outdated code to use the container class (line 28) and the proxy class (line 34).

This section focuses on the program transformations that directly add support for DSU. Besides these, each approach needs to perform minor transformations to deal with other properties of JVM bytecode. For instance, fields may not be directly accessible from other classes and each technique might change their modifier to `public` or inject accessor methods (getters/setters) to be able to intercept field manipulation. DUSC and JRebel represent what would be one instance in the original program with several instances in the rewritten program. These two approaches thus have to rewrite how the transformed program constructs its instances, invokes methods on a super class, handles the `this` reference, among others.

2.4.2 Flexibility

Being an object-oriented language, the basic unit of encapsulation in Java is a *class*. Java classes keep their state in a set of *fields* and define their behavior through a set of *methods*. Classes are related with each other through an inheritance relationship: Each class has a single parent class[2] and may implement several interfaces. The inheritance relationship defines the position each class takes in the *class hierarchy*.

Dynamic update systems for Java can thus support two types of changes: Class structure modifications, which change the set of fields and methods of a class; and class hierarchy modifications, which change the position a class takes in the class hierarchy. I refer to the former as *internal class changes* and the latter as *external class changes*. Other authors provide alternative classifications of class changes [Gus03, WWS10], which I adapt throughout this section to highlight the different design choices of each DSU system for Java.

[2]Except for class `java.lang.Object`, which has no parent class.

```
1  class TempSensor {
2    float averageTemp() { ... } ;
3  }
4
5  class TempDisplay {
6    TempSensor ts;
7    IContainer cont;
8
9    void displayTemp() {
10     ts
11       .averageTemp();
12     ...
13   }
14
15   TempSensor getSensor() {
16     return ts;
17   }
18 }
```

(a) Inital version

```
19 class TempSensor_v2; {
20   float currentTemp() { ... } ;
21 }
22
23 class TempDisplay {
24   TempSensor ts;
25   IContainer cont;
26
27   void displayTemp() {
28     ((Container) cont).ts
29       .currentTemp();
30     ...
31   }
32
33   TempSensor getSensor() {
34     return new Proxy((Container) cont).ts;
35   }
36 }
37
38 // Generated for this update
39
40 class Container implements IContainer {
41   TempSensor__v2 ts;
42 }
43
44 class Proxy extends TempSensor {
45   TempSensor__v2 update;
46 }
```

(b) Update

Figure 2.12: JavAdaptor program transformation. The left-hand side shows the original program version. The update changes method `averageTemp` (line 2) to `currentTemp` (line 20) , and updates all calls accordingly (line 11 becomes 29). All changes that JavAdaptor introduces are highlighted. JavAdaptor also generates two extra classes: `Container` and `Proxy` (lines 40–46).

Internal Class Changes

The least flexible DSU system for Java is HotSwap, which can update only the bodies of existing methods — ○ . However, HotSwap is deployed on most available production JVMs. As explained in the previous section, Kim and Tilevich [KT08] propose a program transformation to support unrestricted updates to Java classes on JVMs that support HotSwap.

DUSC is slightly more flexible than HotSwap. It allows any type of internal class change, as long as it is backwards compatible with the previous version. That is, new versions can add new fields and methods, as long as they still define all the fields and methods that were present on the initial program version — ◓ .

JRebel allows almost any type of internal class change. It still imposes some restrictions on the types of modifier changes it supports on fields and methods. For instance, it does not support adding `synchronized` or `transient` modifiers. It may also crash the program when running static initializers for updated classes. JRebel also does not support any type of custom state transformation between program versions — ◔ .

The remaining DSU systems for Java — JDrums, DVM, JVolve, the DCE-VM, and JavAdaptor — allow any type of internal class change as long as the resulting updated program is type-safe. For instance, if a field is removed and the signature of a method changes, the new program version must not use the

removed field anymore and must use the correct signature when calling that method. The developer can use the Java compiler to enforce this property: If the new program verifies when loaded, it is type-safe and can be installed as a DSU. Ensuring that the new program version compiles ensures that the JVM can verify the updated program.

External Class Changes

The inheritance relationship of all classes in a Java program defines a tree. Any changes to the class hierarchy are changes to the shape of this inheritance tree.

Adding leaf classes does not pose any challenge because the JVM already supports it through its lazy class-loading mechanism [LB98]. On its own, this is not very useful because the old program code cannot refer to the newly added classes. However, even the simpler DSU system for Java — HotSwap — allows to change existing code to refer to new classes. Alternatively, the old program can use reflection to detect new classes that implement/extend old interfaces/classes. All DSU systems for Java thus support adding new leaf classes. JDrums, DUSC, JVolve, and JRebel do not support any other form of external class changes — ◒ .

DVM, the DCE VM, and JavAdaptor support external class changes. DVM requires that the program resulting from the update is type-safe, as explained earlier. The DCE VM does not require type-safe updates and, as a result, may crash the programs due to a DSU that violates type-safety. JavAdaptor generates new Java classes, at update time, for each class that the update modifies. It then changes the bodies of the methods in the outdated Java class (through HotSwap) to call a method in the updated class instead. JavAdaptor uses proxies to support updated methods that return an updated type incompatible with the return type of the outdated method. However, the authors do not discuss how JavAdaptor supports updates that remove methods from the outdated class. Unfortunately, DVM and the DCE VM only support automatic program transformation when performing an update — ◔ . JavAdaptor is the only DSU system for Java that is fully flexible — ● .

2.4.3 Timing

Each approach can apply updates to a running Java program at different points in its execution. I now discuss the update timing options supported by each DSU system for Java.

Unrestricted

The simplest DSU system for Java programs, HotSwap, has also the simplest update timing restrictions: A DSU can be performed at any point in program execution. If a DSU changes a given method m, replacing it with m', that is active at the time of the update, HotSwap will continue executing m after the update until the call returns. Future calls will execute method m'.[3] However, if the outdated method calls an updated method, then the call will be resolved to the new code, which may cause program crashes after a DSU.

JavAdaptor and JRebel build on top of HotSwap, so they are bound by the same update restrictions. However, the authors do not discuss any restrictions or constraints about update timing.

The JVM pauses the execution of the program frequently to perform thread scheduling, garbage collection, JIT compiling, and other runtime related tasks. Update systems implemented at the JVM level can use these pauses to perform DSU. To pause the program without any risk of crashing it, the JVM introduces the notion of *VM safe-points*, which are statically determined points in program

[3]This feature is used to implement stop-edit-continue features in IDEs such as Eclipse. To force the program to run the new method, the IDE typically also pops the stack frame correspondent to the active call. That is why, when changing a running method, it looks like the method call restarts.

execution where it is safe to pause the program and perform all these tasks. For instance, the OpenJDK documentation[4] defines a GC safe-point as: *"A point during program execution at which all GC roots are known and all heap object contents are consistent."*

The DCE-VM performs updates at GC safe-points. However, from the point of view of the application, this is no different from performing unrestricted updates. In particular, methods that an update modifies can be active at GC safe-points. In this case, the DCE-VM behaves as HotSwap and keeps executing the outdated code until the method returns. If the outdated method calls a method that the update removed, the DCE-VM will attempt to resolve that call to new (non-existent) code and crash the program in the process.

Quiescence

JDrums, DVM, and DUSC require that no method from any class that the update modifies is active at update time. These approaches do not perform an update if there are any such active methods. In this case, the program keeps executing in the old version and the update can be retried at a later time.

JVolve, similarly to the DCE-VM, performs updates at GC safe-points. However, JVolve checks if any method of any class that the update changes is active before performing an update at a particular GC safe-point. If so, JVolve skips that GC safe-point and attempts to perform the update at the next one the program reaches. If JVolve fails to perform an update within a certain amount of time, the update fails and the program is left executing in the old program version.

JVolve allows the developer to blacklist methods that cannot be active at update time, even if the update does not modify them. For instance, in the example that Figure 2.1 shows, in page 14, the developer can blacklist method `m` and thus remove the possibility of crashing the program if JVolve performs an update at line 5.

2.4.4 Update Semantics

Performing an update involves changing the code that the program executes and updating the state that the program keeps to be compatible with the new code. The new program code has to be installed at update time, so that the updated program follows the new behavior. However, each DSU solution has more freedom to choose when to transform the program state. This section describes the possible semantics used by DSU systems for Java.

Immediate Updates

Programs written in Java are typically not compiled to native code that is executable directly by the CPU. They are instead compiled to bytecode that is executed by the JVM. Besides interpreting the bytecodes of a Java program, the JVM also provides runtime support in the form of automatic memory management through a garbage collection (GC) mechanism.

As described in Section 2.4.1, DSU approaches implemented at the JVM level can take advantage of the existing GC algorithms to find and transform the outdated program state. The DCE-VM and JVolve force a GC run at update time, which stops the program until all the GC algorithm traverses all the program state. They use a customized GC that detects outdated instances and transforms them while traversing the program state.

DUSC is not implemented at the JVM level, so it cannot take advantage of the GC to transform the program state between versions. It instead rewrites the program to make it updatable. When doing so, DUSC adds an *instance vector* to each wrapper class to track all instances of the interface class across the

[4] http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

application. It also changes the constructors so that they register the new instance on the appropriate vector, and adds a finalizer method to every implementation class that removes the garbage-collected instance from its vector. This way, DUSC can traverse all live instances of updatable objects while the program is stopped.

JavAdaptor also supports immediate update semantics and, as DUSC, is not implemented at the JVM level. JavAdaptor uses the *Java Platform Debugger Architecture (JPDA)* [Oraa] to find all outdated instances that need to be transformed. This API provides method `instances` that finds all objects of a given class, and method `referringObjects` that, given an object o, find all objects in the heap that refer to o. Internally, method `referringObjects` is implemented through a GC run. The authors of JavAdaptor do not discuss the timing of the updates explicitly. The original paper suggests that JavAdaptor supports immediate updates.

HotSwap simply installs new code for already loaded classes and does not transform any program state in the process. We can consider that it has immediate update semantics.

Lazy

Some GC algorithms allow the program to keep executing concurrently with the GC process [JHM11]. This allows the GC to be performed *incrementally*, thus reducing the pause in the program execution otherwise required to reclaim unused memory.

DVM stops all program threads just to change the program code and to prepare the state transformation logic. It then starts an incremental GC run, customized to detect outdated program state and transform it. Of course, the program can find outdated program state before the GC transforms it. To handle that case, DVM traps all accesses that the program makes to fields and methods, so that the outdated instances can be transformed, if needed, as the program uses them.

JDrums is implemented at the JVM level, but it does not use a modified GC to transform the state between program versions. Instead, it adds an extra level of indirection to the internal representation of objects in the JVM: Objects refer other objects through handles. After an update, JDrums can use handles to intercept when the program manipulates an outdated object. At that point, JDrums transforms the state of the outdated object into a new object, and then it updates the respective handle to refer to the new object instead.

JRebel instruments the body of every method with a runtime redirection call. After an update takes place, JRebel can use the redirection to detect when the program is about to use an outdated instance, and update it before allowing the program to use it.

2.4.5 State Transformation

The DSU systems for Java that this section presents are flexible enough to support updates that change the set of fields that classes have. As a result, these DSU systems need to transform outdated instances to conform to their new representation. This section discusses how existing DSU solutions for Java do that.

HotSwap is the exception because it does not support any type of updates that changes the representation of modified classes; I shall not discuss it further in this section. All other approaches support either automatic or assisted program state transformation.

```
1  class Conversion_Point {
2    static class Old {
3      static double x, y;
4    }
5
6    static class New {
7      static double rho, theta;
8    }
9
10   public static void convertObject() {
11     New.rho =
12       sqrt( square(Old.x) + square(Old.x) );
13     New.theta = arctan( Old.y / Old.x );
14   }
15 }
```

(a) JDrums

```
16 public class V1_Point {
17   double x, y;
18 }
19
20 public class JvolveTransformers {
21   ...
22   static void jvolveObject (Point to,
23                             V1_Point from) {
24     double squared_x = square(from.x)
25     double squared_y = square(from.y)
26     to.rho = sqrt( squared_x + squared_y );
27     to.theta = arctan( from.y / from.x );
28   }
29 }
```

(b) JVolve.

Figure 2.13: Example of a JDrums conversion class (2.13a) and a JVolve transformer class, on the right hand side (2.13b). Both classes convert a point in rectangular coordinates (x, y) to polar coordinates (ρ, θ) through the transformation $(\rho = \sqrt{x^2 + y^2}, \theta = tan^{-1}(y/x))$.

Automatic

Automatic approaches match fields between versions by name and type and copy all the fields that the update does not modify. DSU systems that support automatic state transformation can use the default field initialization rules that the Java programming language defines [GJS96] to initialize the set of fields that an update modifies. DVM, the DCE-VM, and JRebel support only this style of automatic state transformation.

Assisted

Automatic approaches minimize the burden on the developer regarding state transformation code, at the cost of supporting only very simple updates that do not change how the program represents its state. For instance, consider the case where the data kept in a list in version 0 is kept in a tree in version 1. This modification, clearly out of the capabilities of automatic state transfer, requires manual intervention.

Some DSU systems for Java support assisted state transformation, in which the DSU system automates as much as it can but still allows the developer to customize the transformation logic. JDrums, DUSC, JVolve, and JavAdaptor follow this approach.

DUSC generates a *state class* for each updatable class in the application, as shown in Figure 2.10. State classes encode the state of instances of updatable classes. Each state class has the same set of fields as the original class used to generate it. To transform an object, DUSC instantiates the appropriate state classe, copies the fields from the outdated object to the instance of the state class, and then passes that instance to the transformation code that the developer wrote. Unfortunately, the authors of DUSC do not show any example of the transformation code.

JavAdaptor automatically maps fields that exist in both versions and performs default initialization for fields added by the update. The authors state that JavAdaptor supports more complex mappings through mapping functions manually defined by the developer. Unfortunately, the authors do not show any example of manually defined mapping functions.

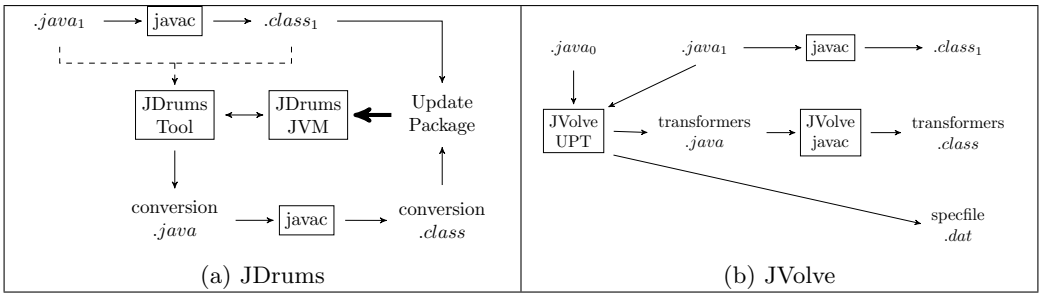


Figure 2.14: Method to prepare updates for JDrums (2.14a) and JVolve. (2.14b). JDrums queries the JVM for information about the current version of the program; JVolve requires the operator to provide the source for the current version of the program. JDrums updates take the form of an *update package*; JVolve uses a special tool (omitted) to load the new version of the application together with the customized transformer code and the update specfile as a DSU.

JVolve and JDrums allow the developer to specify the transformation logic through different approaches. To introduce them, consider the example of a class `Point` that represents a geometric point in a two dimensional plane. Consider that points are internally represented using rectangular coordinates (x, y) in version 0 and polar coordinates (ρ, θ) in version 1. Each point in the form (x, y) can be transformed to (ρ, θ) through the transformation: $(\rho = \sqrt{x^2 + y^2}, \theta = tan^{-1}(y/x))$. Figure 2.13 shows how JVolve and JDrums allow the developer to specify the transformation logic.

For each class that an update modifies, JDrums creates one *conversion class*. Each conversion class has two inner classes, *New* and *Old*, that reflect the fields of the new and old versions, respectively. Figure 2.13a shows an example of a conversion class for the point-transformation example. The developer specifies the conversion logic by manipulating static fields on each inner class. When transforming an outdated point into an updated one, JDrums translates each manipulation of the static fields to the correct manipulation of the respective field in the correct version.

JDrums provides a tool to generate stub conversion classes. Figure 2.14a shows how to use the tool JDrums provides. It takes as input the new version of the program[5] and it queries the JDrums JVM directly to compare the new version against the current version in execution. It then generates a stub conversion class, that the developer can customize with the program state transformation logic. JDrums copies automatically all the fields that did not change between versions. The developer then creates the *conversion package*[6] by bundling together the conversion class and the updated code. At this point, it is possible to start the DSU process by signaling JDrums that a conversion package is ready.

JVolve generates a *transformer class* that contains an *object transformer method* for every class that an update changes. Figure 2.14b shows an example of a transformer class for the point transformation example that we are following. Each transformer method takes two arguments: A blank new object and an outdated object. The transformer method transfers the state from the outdated object to the blank object.

JVolve provides a tool called *Update Preparation Tool (UPT)* to generate stub transformer classes, and the types to represent the outdated objects. Figure 2.14b shows how to use UPT. UPT compares both program versions and generates code to copy the state that did not change between versions. The resulting transformer class may contain invalid Java code, such as writing to final fields. JVolve thus provides a modified compiler to generate the bytecode for the transformer class. Once the developer customizes the transformer logic, he can can signal JVolve to perform an update, providing the bytecode for the transformer class together with the new program version and an *update specfile* with metadata about the update, which is also generated by UPT.

[5]The authors are not specific about the form of the input, source-code or bytecode.
[6]The authors do not specify how to do this step.

2.4.6 Discussion

This section described the state of the art in DSU systems for the Java language. I now explain its respective rows on Table 2.1, page 12, by relating each DSU system for Java with the goals that I introduced earlier in Section 1.2.

Flexibility

Programs written in Java are composed by a set of classes, that are related with each other through an inheritance relationship in which each class has a parent class and implements a set of interfaces.

All systems support updates that simply add new classes to the application. In fact, the JVM itself already supports that feature through lazy classloading [LB98]. However, the ability to load new classes on itself is not a very flexible DSU system.

HotSwap is the least flexible DSU system for Java. It allows updates to change only the bodies of existing methods. Updates cannot change the class signature in any way, and this includes changing the signature of any method — ○ .

The rest of the DSU systems for Java support any internal class change, i.e., updates that change the set of methods and fields that each class defines. The vast majority of the remaining systems — JDrums, DUSC, JVolve, and JRebel — do not support updates that change the position of existing classes in the class hierarchy — ◓ .

The only DSU systems for Java that support internal and external class changes are DVM, the DCE VM, and JavAdaptor — ● . However, the DCE VM does not support custom program state transformation — ◐ .

Efficiency

The JVM executes Java programs by interpreting their bytecode. To improve performance, it uses a Just-in-time (JIT) optimizing compiler to generate native code for the portions of the Java program that are most frequently interpreted. The JIT compiler is thus an important part of the performance of Java programs. JDrums and DVM disable it, using a modified interpreter to execute, and update, Java programs. Therefore, these DSU solutions impose a large steady-state overhead when not performing an update. However, both DVM and JDrums support lazy program-state migration, which means they do not impose a large update pause when performing an update — ◓ .

The other systems implemented at the JVM level — HotSwap, JVolve, and the DCE VM — can update programs without disabling the JIT compiler. They impose little, if any, performance overhead in steady-state execution. However, JVolve and the DCE VM require a GC run to transform the program state at update time. The program does not execute during this GC run, whose duration is proportional to the overall size of the program state — ◓ .

JavAdaptor and DUSC are not implemented at the JVM level but also pause the application at update time to transform its state — ◓ . Besides, DUSC adds a non-trivial amount of steady state overhead — ○ .

The remaining DSU systems for Java, HotSwap and JRebel, impose low overhead on steady-state and do not pause the program at update time for arbitrarily long periods of time. HotSwap does not provide any support for transforming the program state and JRebel is able to do so lazily — ● .

Correctness

All the DSU systems for Java that I present in this section can crash a running program by performing an update at the wrong time.

When performing an update, both HotSwap and the DCE VM keep executing the old version of updated methods active at update time. These approaches thus execute old code concurrently with new code, which is inherently unsafe. The authors of the DCE VM point out that deleting a method can crash the application if any method active at update time calls the deleted method after the update — ○ .

Even though the authors of JRebel and JavAdaptor do not discuss the timing at which updates happen, these systems are based on HotSwap. I assume that they have similar update timing constraints — ○ .

To perform an update, JDrums, DVM, and DUSC require modified methods to become quiescent. This approach precludes some obviously wrong updates, but still allows updates to happen at the wrong time, as I explained when introducing the goal of correctness in Section 1.2.2 (page 5) — ◔ .

JVolve also relies on quiescence, but it uses a blacklist to prevent updates while sensitive methods are active. The blacklisted methods must be identified by the developer. This blacklisting approach is an important step towards correctness. However, it does not provide any tool that allows the developer to identify which methods to blacklist, or to test that the blacklist is not missing any method — ◓ .

Effectiveness

All DSU systems presented in this section target a popular language, Java; none requires that the updatable program follows a particular architecture; and none restricts the set of language features and idioms that a generic Java program can use to still be updatable. This is an important part of the effectiveness goal.

Some DSU systems are built specifically as a development time tool — HotSwap, JRebel, the DCE VM, and JavAdaptor. A DSU approach designed to be used as a development time tool has more relaxed constraints than a DSU system designed to be used in production. For instance, it is perfectly reasonable to expect updates to fail randomly and crash the program in the former case (as long as it does not happen too frequently), whilst an update-related crash defeats the purpose of a DSU system in the latter case. We can use the same argument for efficiency and flexibility. Therefore, DSU systems designed as development time tools are not fully effective — ◓ .

Other DSU systems require a custom JVM — JDrums, DVM, JVolve, and the DCE VM. Given that the JVM plays a central role in executing Java programs, restricting it to one particular implementation limits the effectiveness of these approaches — ◕ .

DUSC is the only DSU solution for Java that targets programs running in production and does not require a custom JVM. Therefore, it is the only approach fully effective for Java programs — ● .

2.5 Object-Oriented Database Management Systems

A large class of applications do not persist their own state, they delegate that responsibility to a Database Management System (DBMS). A popular choice is to use a Relational DBMS (RDBMS) that represents the state of the application through a set of relations between tables. The structure of the data that the database keeps; i.e., the names of the tables, the columns of each table, and the relations between columns in different tables; is called the *database schema*, or just *schema* for short. The data itself is kept in rows in all the different tables, and is thus consistent with the database schema.

System	Language Bindings	Change Detection	Update Semantics	State Transf.
O_2	C++	API	Lazy / Deferred	Assisted
Versant	C++, Java	API	Lazy	Automatic
Objectivity/DB	C++, Java, ...	Comparison	Lazy / Deferred	Automatic
GemStone	Smalltalk, Java, C	API	Immediate	Manual
PJama	Java (OPJ)	Hybrid	Offline	Assisted / Manual

Table 2.4: Systems that support DSU for OODBMSs. This table summarizes the discussion on this section. Rows titled update semantics and state transformation use the classification introduced in Section 2.1. The remaining rows, language bindings and change detection, describe which languages each OODBMS supports and how each OODBMS detects changes between versions, and are described in detail in Sections 2.5.1 and 2.5.2, respectively.

RDBMSs need to evolve to support unanticipated changes to the original schema. They do so through a special language, typically a dialect of SQL, to modify the schema. Each different RDBMS provides a different level of support to transforming the data to be consistent with the new schema. The literature on updating schemas without stopping the RDBMS is plentiful but, however, outside the scope of this document. I shall not discuss it any further.

An alternative to using RDBMSs is to use Object-Oriented DMBSs (OODBMSs) to persist the state of the application through a set of interconnected objects. The database schema in this case is a set of classes, each class declaring a set of fields, and the inheritance relationship between all the different classes. The data itself is represent by a graph of objects. Each object belongs to a class, which keeps the data consistent with the database schema.

This is very similar to how programs written in object-oriented languages structure their state, as I described in the previous section. Given the similarity between the two, this section briefly describes the state of the art on OODBMS systems. In particular, it describes how each OODBMS supports changes to the structure of the data it keeps, either online or offline.

This section presents and classifies the following OODBMSs, according to how they support changing the structure of the data they keep, and transforming the existing data to conform to a different structure: O_2 [FMZ+95], Versant [Ver15], Objectivity/DB [Obj13], GemStone [Gem14], and PJama [DA99].

Table 2.4 summarizes the design decisions that each OODBMS takes, in terms of supporting schema changes. It also gives an overview of the rest of this section.

2.5.1 Language Bindings

O_2 and PJama are research OODBMSs. The rest of the OODBMSs considered — Versant, Objectivity/DB, and GemStone — are commercial systems.

O_2, Versant, and Objectivity/DB provide bindings for C++. Versant and Objectivity/DB also provide bindings for Java. Objectivity/DB supports bindings for more languages: C#, Python, SQL++, and Smalltalk.

GemStone is designed for Smalltalk and also provides bindings for Java.

PJama is a research project for *Orthogonally Persistent Java (OPJ)* programs [AM95]. It includes a modified JVM (PJVM) and an object store to persist the state of the modified JVM.

2.5.2 Change Detection

Generally, OODBMSs do not restrict the types of class updates they support as I classified them in Section 2.4.2. However, they provide two different ways for the developer to specify how the database schema changed between versions: (1) The OODBMS *compares* the new schema and the old one and detects the changes automatically, or (2) the developer uses an *API* to declare the changes between schema versions.

Change API

O_2, Versant, and GemStone require the developer to use an API to describe what changed between two successive database schemas.

O_2 provides a special DSL to declare the schema changes. In this DSL, the developer lists all updated classes, and all fields that were added/removed/modified in each updated class. The DSL also allows the developer to specify the transformation logic to make existing instances consistent with the new schema.

Versant provides a change API that supports adding new classes and adding/removing/renaming fields belonging to an existing class. It does not provide direct support for any other class change, such as renaming a class or changing its position in the class hierarchy. In this case, Versant considers that the update deleted the old class and introduced a new class. As an unfortunate side effect, this action deletes all the objects that belong to the old class. To avoid losing data this way, the developer must follow a multi-step process that involves: (1) Adding the updated class to the schema, (2) writing a program to manually transform all instances of the old class into instances of the new class, and (3) removing the outdated class.

GemStone does not allow classes to change. The developer must declare class updates as new classes. GemStone provides the concept of *class histories* to relate class updates: Each class belongs to a class history and GemStone provides API support to declare that a new class belongs to the same class history of an existing class. GemStone then allows instances of any class to be transformed into instances of any other class in the same class history. The two classes do not need to be similarly named, or have anything in common except the class history. When transforming instances to the newest class on their class history, GemStone provides a change API to map fields between classes.

Schema Comparison

When the schema changes, Objectivity/DB can compute all the differences between the old and the new database schema. It does not require the developer to specify any change explicitly. Objectivity/DB then keeps the old schema in a *schema-evolution history*, which it uses to construct programs for converting each object to the latest schema during on-demand object conversion (discussed in detail in Section 2.5.3).

Hybrid Approaches

PJama detects schema changes following a hybrid approach: It compares the schema to detect changes at the class level, e.g. adding/removing/renaming fields, but it requires the developer to explicitly list schema changes to the class hierarchy through its change API.

2.5.3 Update Semantics

Typically, OODBMSs execute in their own process, separate from the applications that use them. There may be several client applications connected to a single OODBMS, modifying it concurrently. Performing a dynamic schema update on a database kept in an OODBMS is thus different from performing a dynamic

update on a running program. One main difference is *backwards compatibility*: Maintaining backward compatibility with clients already using the old version of the schema. Backwards compatibility enables OODBMSs to support schema changes without requiring clients to disconnect. DSU systems for programs that do not allow old code to execute after the update do not need to provide backwards compatibility.

In this section, I present the different update semantics that each OODBMS considered provides.

Lazy

O_2, Versant, and Objectivity/DB provide lazy update semantics by default. This makes sense because schema updates in OODBMSs require transforming the data to be compatible with the new schema. The main goal of an OODBMS is to keep a potentially large amount of data accessible to a set of client applications. Immediate updates impose a pause in the service provided by an OODBMS that is proportional to the size of the database, which is not acceptable in this scenario.

Deferred

Deferred update semantics is a variant of lazy update semantics where a background thread ensures progress in the transformation of the program-state ("pushes the world forward") by traversing the program state and triggering lazy updates eagerly. The program state, in this case, is the data that the updated database keeps.

Deferred update semantics has the advantage that updates can be performed as fast as lazy updates, with the guarantee that eventually all the program state will have been transformed to the new program version, i.e., the new schema.

O_2 and Objectivity/DB provide support for deferred update semantics. O_2 does so through a tool that traverses the database and performs the lazy migration algorithm on-demand, for all objects that need to be transformed. Objectivity/DB allows a finer grain of control when performing deferred data transformation: The developer can start an *update transaction* and update a set of objects, on demand, inside that transaction. Objectivity/DB provides API to transform all objects in a container and in a database during an update transaction, and the manual encourages developers to use deferred updates to control the performance impact of data transformation.

Immediate

GemStone provides immediate update semantics. After the update, the developer starts a transaction to perform the immediate state transformation. In GemStone, a class is updated by adding a newer definition to the same class history and then explicitly transforming the outdated instances to the updated class. However, the developer can decide not to transform some instances and thus leave them in the old schema.

Offline

Some OODBMSs require that no client is accessing a particular database while it is transforming the data to the latest schema definition. This corresponds to the *offline update semantics*.

PJama requires that no client is accessing a particular database while it is transforming the data to the latest schema definition. Versant also supports offline update semantics, besides the default lazy update semantics. Versant provides a tool — *schevol* — to transform all instances in a database to the latest schema definition. The Versant manual describes that multiple schema versions in the database can negatively impact query performance, as the query is repeated for each schema version. The provided tool *schevol* thus ensures good query performance in the presence of schema updates. However, *schevol* must be only used in an offline database, i.e., without any active transaction, and may take a long time to complete.

2.5.4 State Transformation

The main goal of using an OODBMS is to keep the data that a program uses, persist it, and make it readily available to the client programs. Transforming the state between schema updates is thus an important part, if not the most important part, of considering schema updates in the context of DSU. In this section, I discuss the different approaches to transform the data that each OODBMS supports.

Automatic Transformation

All the OODBMSs that I consider in the section — O_2, Versant, Objectivity/DB, GemStone, and PJama — support automatic transformation of the data that a database keeps when performing a schema update. During automatic transformation, all fields that were unaffected by the schema change retain their value. Automatic transformation also initializes new fields to a default value — zero, false, or null, depending on the type of the field.

Versant and Objectivity/DB do not support any other type of state transformation.

Assisted Transformation

Besides automatic state transformation, O_2 and PJama also support direct assisted state transformation. Both do so through *conversion functions* or *conversion methods*, one for each class that an update changes. Each conversion function takes an existing object that belongs to the outdated class, O_{old}, and a blank object that belongs to the updated class, O_{new}. After the update, O_{new} takes the place of O_{old} in the object graph. The developer writes the code that initializes the state of O_{new} given O_{old}.

O_2 classifies conversion functions as either *simple* or *complex*. Simple conversion functions access only the objects being transformed (O_{old} and O_{new}); complex conversion functions access more objects besides the one being transformed. To implement complex transformation code, O_2 keeps O_{old} after O_{new} takes its place in the new object graph, hidden as a *screened value*. Future complex conversion functions that require O_{old} can still access it. O_2 statically analyzes the transformation code and computes a *dependence graph* to avoid keeping screened values when they are not needed, i.e., for classes that are never accessed by complex transformation code. O_2 cleans all the screened values after a deferred update completes.

PJama does not support lazy program state transformation; it transforms the whole state in a single step. PJama always keeps O_{new} separate from the object graph until the transformation finishes. That is, the transformation code of other objects can only see O_{old}, even if PJama has already produced O_{new}. This way, the transformation code can traverse the whole heap during transformation, and always find old objects. After transforming the whole program state, PJama fixes all references to outdated objects so that they refer to their updated version instead. Then, PJama can safely dispose of all the outdated objects.

```
1  // Class C changed
2  static void convertInstance(C_old_ver_ c0, C c1);
3  static C convertInstance(C_old_ver_ c0);
4  static <? super C> convertInstance(C_old_ver_ c0);
5
6  // Class C deleted
7  static void migrateInstance(C c0, <? super C> c1);
8  static <? super C> migrateInstance(C c0);
9
10 // Perform automatic conversion
11 static void copyDefaults(Object oldObject, Object newObj);
```

Figure 2.15: API that PJama provides for the developer to specify the state transformation logic when updating the schema of an existing datastore. When an update changes a class that requires any transformation logic, the developer defines one of these methods in a conversion class. Each conversion method provides the logic to transform all objects of an outdated class by initializing a blank object of the updated class. Some methods take an outdated object as argument, with the type prefixed by `_old_ver_`. PJama uses a modified Java compiler and a modified JVM that understand the special meaning of these class names. Method `copyDefaults` is an utility that the developer can call from conversion methods to perform the automatic (default) conversion on-demand. PJama predates Java templates but I use them to specify the constraints on some of the arguments and return values.

PJama provides several ways for the developer to write the state transformation logic. Figure 2.15 shows the possible alternatives. When a class changes, the developer can define one of the possible methods to transform objects of the changed class. Some methods take two instances and require the developer to initialize the new instance with the state on the existing outdated instance (lines 2 and 7). Other methods require the developer to create the new instance (lines 3, 4, and 8), which gives the developer the chance to change the class of the new instance. When using these methods, the developer can still use the automatic conversion through method `copyDefaults` (line 11). The methods shown in lines 4 and 8 allow the developer to break the type system by transforming an existing instance to an instance of an incompatible type. Once the developer writes all conversion methods for a particular schema change, he places them in one *conversion class*.

Manual Transformation

PJama supports two types of state transformation: *Bulk* and *fully controlled* conversion. In bulk state transformation, PJama applies the same transformation logic to all the objects that belong to the same class, using the conversion methods defined in the preceding text in this section. It traverses all the program data to find all the outdated instances. For each outdated instance it finds, PJama transforms it by copying all the unmodified fields and initializing new fields with a default value through automatic transformations, and then running any custom transformation code present in the conversion class.

The alternative is to perform fully controlled conversion. In this mode, the developer provides transformation code that runs *instead* of any automated transformation logic. When using fully controlled transformation, the developer writes all the code to traverse the relevant portion of the state and to transform the outdated instances. Fully controlled conversion is useful to ensure a specific order of transformation of outdated objects, or to restructure the data in addition to transforming it to comply with the new schema.

GemStone also provides support for manual transformation. The developer is required to start a new transaction to transform instances to be compatible with a new schema definition. Only transactions that start after such a transformation transaction finishes will see the transformed state. GemStone requires the developer to write code to find all the outdated instances and then transform them to their latest definition. It does make the task easier by providing methods to iterate over all instances of a particular class.

2.5.5 Discussion

The problem of performing a dynamic schema update on a database kept in an OODBMS is similar to the problem of performing a Dynamic Software Update on a running program. However, despite the similarities, these are two separate problems and some of the goals that I defined in Section 1.2 do not map well to OODBMSs. In the following, I explain how each goal can be interpreted in the context of OODBMS schema changes, and how each OODBMS considered achieves, or not, each goal. The rows on Table 2.1, page 12, relative to this section, summarize this discussion.

Flexibility still makes sense in this context. An OODBMS is flexible if it supports the developer to both: (1) Change the schema between versions in any arbitrary way, and (2) transform existing databases conforming to the old schema to a state that is equivalent and conforms to the new schema. Versant and Objectivity/DB are not flexible because they support only automatic database transformation between schemas, which does not generate an equivalent database in the presence of complex schema changes — ○. All other OODBMSs considered — O_2, GemStone, and PJama — are flexible because they allow the schema to change arbitrarily between versions and provide the developer with a way to express custom state transformation logic — ●.

Efficiency is another goal that still makes sense in this context. An OODBMS is efficient if it both: (1) Supports schema changes without performance degradation, and (2) does not impose a pause that is proportional to the size of the database to perform a schema change. PJama is not efficient because it supports only offline schema changes — ○. O_2 and Objectivity/DB support lazy database transformation after a schema change, but the particular implementation may affect post-update performance — ◓. To deal with that, they also provide *deferred* updates, that "push the world forward" by traversing the database in a background thread and transforming all relevant state to the new schema. Furthermore, Objectivity/DB allows the developer to transform a subset of objects immediately after an update to control the performance impact of a schema change.

The efficiency of GemStone is hard to fit in this model because all schema changes are backwards-compatible, and GemStone may keep objects that belong to multiple schemas — ◔. When updating, the developer starts an update transaction to manually transform existing objects to the new schema while transactions still execute with the old schema, isolated from such an update transaction. The developer does not need to transform all objects to the new schema.

Effectiveness and *Correctness* do not make much sense in this context. Effectiveness requires the DSU system to target popular languages to maximize its impact, which does not make much sense for the schema update solutions discussed for OODBMSs. Correctness requires that the developer can ensure that the updated program behaves as expected after a DSU takes place. Given that OODBMSs do not execute application code, this goal also does not make much sense.

2.6 Programming Language Support for DSU

Supporting Dynamic Software Updates directly at the programming language level is appealing: All programs written in such a language directly benefit from the high availability that DSU provides.

Some programming languages provide support for DSU. In this section, I present the most relevant work on DSU support at the programming language level. In particular, I discuss the support for DSU on the following languages: Common LISP [Ste90], Smalltalk [GR83], Erlang [AVWW96], and UpgradeJ [BPN08].

2.6.1 Common LISP

LISP[7] is a multi-paradigm programming language. LISP is an *homoiconic* language: LISP programs represent code in the same way as data. Code is, therefore, a first-class citizen. This makes LISP a naturally reflective language [BGW93] that has direct support for *program introspection* — the program can learn about its own structure — and, depending on the particular implementation, *program intercession* — the program can modify its own structure. LISP also provides an extensive system of macros that enables programmers to extend its syntax simply by writing LISP code that processes LISP code as a any other data structure.

LISP directly supports DSU through the *Common LISP Object System (CLOS)*. CLOS is an object-oriented programming language built around the concept of *generic functions*, which are functions whose behavior depends on the classes of the arguments supplied to it, and *methods*, which are specializations of generic functions for a particular set of argument classes.

To explain CLOS using a more popular programming language, let us consider how they map to Java. CLOS classes hold their data in *slots*, which are similar to fields of Java classes. Generic functions can be considered as virtual (i.e., non-final) Java methods. Each CLOS method is an implementation of a Java method. For instance, consider Java method `Object.toString()`. In CLOS, it would be a generic function defined as `(defgeneric toString (obj))` with a default method defined as `(defmethod toString (obj Object) ...)`. Overriding Java method `toString` means defining a new method for the generic function `toString` in CLOS.

One major difference between Java (or similar well-known object-oriented programming languages, such as C++ and C#) and CLOS is how methods are selected. When invoking a method in Java, there is a special argument called the *receiver*. Java performs dynamic dispatch, i.e., it selects which overriden method to run, based on the type of the receiver. CLOS also performs dynamic dispatch to select which method to execute when calling a generic function. However, CLOS considers the types of *all the arguments*.

The behavior of CLOS is defined through a *Meta-Object Protocol (MOP)*. The MOP is written in CLOS and specifies a set of interactions between generic functions which defines the behavior of CLOS itself. For instance, generic function `compute-effective-method` selects which method to execute when calling a generic function, and generic functions `slot-value-using-class` and `(setf slot-value-using-class)` implement reading and modifying slots, respectively.[8] The developer can provide new methods for all generic functions defined in the MOP, and thus customize the behavior of CLOS.

CLOS supports redefining classes simply by defining a new class with the same name. When redefining a class, the set of slots that a class defines can change. In that case, CLOS propagates the changes to the instances of the redefined class and its subclasses. The moment in which the instances are updated is left implementation dependent, as long as it is no later than the first access made to a slot of the outdated instance. This allows implementations to support either immediate or lazy update semantics.

[7] Common LISP is a dialect of the LISP programming language [Ste90]. Throughout this section, I shall refer to Common LISP simply as LISP.

[8] This example is very simplified. The exact protocol to perform these tasks involves the interaction of more generic functions.

The MOP specifies a 2-step process to update instances, when the new class definitions has a different set of slots:

1. Modifying the structure of the instance by adding new slots and discarding slots that are not present in the new class definition. Slots present in both the old and new class definitions retain their value. This step is completely automatic. It retains the identity of each updated instance, even if the implementation needs to allocate physical space for the new slots.

2. Initializing the newly added slots and performing any other custom actions. Slots that are present in both the old and new class definitions retain their value. Newly added slots are initialized, by default, with the slot initialization logic present in the new class definition. The developer can further customize this step by providing methods for generic function `update-instance-for-redefined-class`, which takes 4 arguments:

   (a) The instance being updated;

   (b) List of names of new slots;

   (c) List of names of discarded slots;

   (d) Property list with the values of the discarded slots.

CLOS also provides support for changing the class to which an instance belongs to, through function `change-class`. The MOP specifies a 2-step process to do this, similar to the process for updating instances. The developer can customize it in the same way, by providing methods for generic function `update-instance-for-different-class` which is called on step 2.

2.6.2 Smalltalk

Smalltalk is an object-oriented programming language. Smalltalk programs work by passing messages between objects. Each object holds some state, private to itself, and receives messages sent by other objects or itself, by executing a method. While processing messages, the object can send messages to other objects and to itself. Calling a method in Smalltalk is equivalent to sending a message to an object (the receiver) with all the method arguments.

Similarly to LISP and CLOS, Smalltalk is almost entirely written in itself. However, unlike LISP, Smalltalk is not homoiconic. Instead, Smalltalk provides a comprehensive support for *reflection*, both introspection and intercession. Smalltalk provides a *meta-object protocol (MOP)* that gives meaning to Smalltalk programs by describing all aspects of the language, including compilation. Rivard [Riv96] describes the reflective capabilities of Smalltalk in detail; in the following, I provide a brief overview of the parts that provide support for Dynamic Software Updating.

The classes of the MOP that are most relevant to DSU are `Object`, and `Behavior`. Everything in Smalltalk is an object, which means everything is an instance of class `Object`. This class provides the basic protocol common to all objects. Class `Behavior`, as the name indicates, provides behavior that is common to all classes. Class `Behavior` is a subclass of `Object`.

The process of defining a new class in Smalltalk involves interacting with class `Behavior` to create a method dictionary for the new class. The new method dictionary can then be used to compile, and thus make available, all the methods that the new class defines. During program execution, class `Behavior` keeps the method dictionary of each class and allows the program to inspect and modify it. This enables changing the code while the program is running, which in turn provides basic support for DSU.

```
1  - module(xyz).
2
3  loop(Arg1, ..., ArgN) ->
4    receive
5      ...
6    end,
7    loop(NewArg1, ..., NewArgN).
```

```
8  - module(xyz).
9
10 loop(Arg1, ..., ArgN) ->
11   receive
12     ...
13   end,
14   xyz:loop(NewArg1, ..., NewArgN).
```

Figure 2.16: Example of code update in Erlang. Both sides define module `xyz` with function `loop`. Function `loop` on the left-hand side ends with a tail-call to itself, on line 7, that Erlang resolves at load-time. However, function `loop` on the right-hand ends with a fully qualified tail-call, on line 14, which Erlang resolves dynamically at every call to the most recent version of module `xyz`. This code replacement feature can be used to replace module `xyz`.

Besides defining new code, Smalltalk also supports traversing and transforming the program state. Class `Behavior` supports listing all the instances of a particular class through method `allInstances`, and executing a block of code for each instance of a particular class through method `allInstancesDo`. Given one particular instance, class `Behavior` can list all its instance variables through method `allInstVarNames`. Unfortunately, Smalltalk does not support manipulating instance variables directly. However, it supports injecting methods that manipulate each instance variable. Besides manipulating the state of existing objects, Smalltalk also supports copying them through methods `shallowCopy` and `deepCopy` of class `Object`. Class `Object` also provides method `become` that takes a single argument and swaps the identity of the receiver and the argument. Smalltalk thus provides more than enough support to traverse and transform the program state when performing a DSU.

Of course, an important part of the program state is the call stack and program-counter. Smalltalk reifies[9] the call stack as a chain of linked stack frames called *contexts*. Each method can access its own context through variable `thisContext`, which contains the current method being executed and its arguments, the program-counter, and the receiver object. All Smalltalk implementations support inspecting the call stack. Modifying the call stack depends on the particular Smalltalk implementation. Still, this provides support for accessing/transforming the control-related program state when performing a DSU.

2.6.3 Erlang

Erlang is a programming language designed for building large-scale distributed systems. Erlang provides language support for concurrency through the concept of a *process*. Each process in Erlang is executed by a single thread and communicates explicitly with other processes by exchanging asynchronous messages. All data that a process keeps is private to itself, so there is no implicit communication through shared data, which removes the need for complex concurrency control mechanisms such as locks.

Erlang code is structured in *modules*. Each module defines a set of functions. Erlang provides support DSU through code replacement at runtime at the level of modules. Figure 2.16 provides an example of Erlang code. The example defines module `xyz` with a single function `loop` that calls itself with a tail-call on line 7.

[9] Reification is the process of encoding execution state as data. [BGW93]

The code replacement mechanism in Erlang is built on top of the dynamic module loading mechanism. Erlang allows new modules to be loaded dynamically while the program is executing. Dynamic code loading happens when the program first references a module that is not yet loaded. When loading each module, Erlang resolves references to itself and other modules. For instance, in the code example shown in Figure 2.16, Erlang resolves the tail-call in function loop (line 7) to itself when the module is loaded. All future calls will be made to the function resolved at load-time.

Erlang treats fully qualified accesses differently, like the one on line 14. Erlang dynamically resolves fully qualified accesses every time they are made, and not once when it loads the module. As a result, the operator can load a new version of module `xyz` and wait for function `loop` to reach the fully qualified tail-call. At that point, Erlang will resolve that call to the function defined in the updated module, effectively performing a DSU.

The code of each Erlang module can have up to two versions: The *current* and the *old* version. When Erlang loads a module for the first time, its code becomes *current*. If Erlang later loads a new instance of that module, the code of the previous instance becomes *old* and the new code becomes *current*.

Erlang allows *old* and *current* code to execute concurrently in the same program. The *current* code can be accessed through fully qualified accesses, the *old* code can still be accessed through lingering processes. If Erlang loads a third instance of the same module, it removes (purges) the old code and terminates all processes still lingering in it.

2.6.4 UpgradeJ

UpgradeJ is an extension to the Java programming language that provides language support for DSU at the class level. UpgradeJ extends Java syntactically by requiring all classes to be annotated with a version number. Figure 2.17 provides an example of a simple class `Button` written in UpgradeJ, together with two updates. This example highlights the syntactic changes and shows the three different types of class updates that UpgradeJ supports:

New class upgrades: (line 1) Add a new class definition to the program. New class upgrades are similar to dynamically loading new classes, which the JVM already supports through class-loaders [LB98].

Revision upgrades: (line 11) Change the method bodies of a class. Revision updates cannot change the signature of the updated class. Revision updates are very similar to the DSU support provided by HotSwap [Orab], described in detail in Section 2.4. In UpgradeJ, revision upgrades allow existing code to refer to classes added by new class upgrades.

Evolution upgrades (line 13) Add methods and/or fields to a class. Evolution upgrades must be *backwards compatible*, i.e., retain all methods and fields that the outdated class defines.

A program written in UpgradeJ can declare instances to be *exact* (line 3) or *upgradable* (lines 4 and 5). An upgradable instance automatically uses the latest revision of its class (and its superclasses), and an exact instance always uses the same revision of its class. In the example shown in Figure 2.17, note how UpgradeJ propagates the revision upgrade performed on line 19 to all upgradable instances.

UpgradeJ does not propagate evolution upgrades to existing upgradable instances. To use the latest evolution upgrades, developers create upgradable instances using the syntax shown in lines 5 and 25. This creates an instance in the latest evolution upgrade of a class. However, the instance will remain in the same evolution upgrade throughout its lifetime.

```
1  new class Button[1] { public Color getColor() { return Color.WHITE;} }
2
3  Button[1] a = new Button[1=](); // Creates an exact version instance of Button[1]
4  Button[1] b = new Button[1+](); // Creates an upgradable instance of Button[1]
5  Button[1] c = new Button[1++]();// Creates an upgradable instance of Button[1]
6
7  a.getColor(); // White
8  b.getColor(); // White
9  c.getColor(); // White
10
11 new class Button[2] revises Button[1] { public Color getColor() { return Color.BLACK; } }
12
13 new class Button[3] evolves Button[2] {
14   private Color color = Color.RED;
15   public Color getColor() { return this.color; }
16   public void setColor(Color color) { this.color = color; }
17 }
18
19 upgrade(); // Installs classes Button[2] and Button[3]
20
21 a.getColor(); // White
22 b.getColor(); // Black
23 c.getColor(); // Black
24
25 c = new Button[1++](); // Creates an upgradable instance of Button[3]
26 c.getColor(); // Red
```

Figure 2.17: Example of UpgradeJ class updates.

The restrictions on each type of class upgrade that UpgradeJ supports allow instances in different versions to co-exist in the same program and still retain type-safety. This allows UpgradeJ to perform DSU without traversing the whole program state to transform it to be compatible with the new program version. The authors explain how revision and evolution upgrades can be used together to provide a flexible update model. The authors also formalize the type system of UpgradeJ and prove its soundness.

2.6.5 Discussion

Providing DSU support directly at the programming language level is a promising approach. Ideally, all programs written in a language that supports DSU can provide a high level of availability with no extra effort in development. Unfortunately, all the approaches that I consider in this section fall short of the goals for a pragmatic DSU system that I defined in Section 1.2. The relevant rows of Table 2.1, in page 12, summarize this discussion.

The approaches that I consider in this section require the program to be written in a particular programming language to support DSU. The applicability of these approaches is thus limited to the programs written in each language. Unfortunately, these languages are not very popular[10] and that limits the applicability of these approaches for DSU — ◓ .

Efficiency is hard to reason about for DSU supported at the programming language level. Steady-state performance — pre- and post-update — and update pause are all left implementation-dependent. Still, LISP, Smalltalk, and Erlang allow developers or implementations to provide lazy update semantics, and thus perform updates without inducing a long pause in the execution of the application — ◓ . UpgradeJ is carefully developed to support DSU at the type system level, which means that the compiler can optimize version checks and the update process does not impose a long pause — ● .

Regarding correctness, LISP and Smalltalk leave the exact moment at which an update takes place completely implementation dependent — ○ . In Erlang, the points in execution at which updates take place are explicitly identified in the program text, one per module. However, it is hard to reason about

[10]Some, like Erlang, are very popular inside specialized niches.

updating several, inter-dependent, modules in Erlang. Furthermore, given that Erlang uses a dynamic type-system, an update can break the type system and cause the program to crash — ◓ . UpgradeJ is designed around a static type system and the authors prove its soundness even in the presence of updates. However, it is still possible to design an update that breaks the program semantics without breaking the type system, e.g. by programmer error, and UpgradeJ does not provide any way for ensuring that the updated program will behave as intended — ◐ .

Finally, all programming languages that support DSU achieve the flexibility goal because they allow programs to change arbitrarily between successive versions — ● .

2.7 Other Approaches

There are some well known existing approaches to support DSU that require programs to be written in a particular style, follow a particular design architecture, or that require redundant hardware. This section describes the most relevant of such approaches.

2.7.1 Modular Systems

Systems such as OSGi [OSG14], the Netbeans Platform [BTW07], or the Eclipse Rich Client Platform [ML05] allow software to be developed by creating a set of *modules* that interact with the platform and with each other to provide the required functionality. The platform itself simply manages the life-cycle of each module by providing support for adding, removing, or swapping modules. The framework also provides a module discovery service, so that modules can discover other modules and establish communication channels for inter-module collaboration.

Developing software for these platforms naturally promotes good practices of software engineering, such as encapsulation and code re-use. This eases the challenge of providing support for DSU. For instance, given that a module A communicates with another module B though its public interface, without knowing the specific implementation of B, it is possible to swap B by another module B' that provides the same interface but has a different implementation. In fact, this is exactly the approach that Fabry [Fab76] proposes on the earliest research on DSU.

However, modular systems are far from being a panacea for DSU. Gregersen and Jørgensen [BTW07] point out the main shortcomings when they consider DSU on the NetBeans modules framework through module reloading. In the following, I summarize the most relevant points they make.

Dependent modules. Consider the example that I introduced earlier: Module A communicates with module B. To do so, module A uses the module discovery service that the platform provides to discover module B. It then proceeds to invoke methods of B through its interface. However, in doing so, A holds a reference to the current module B. If the platform later reloads module B, swapping it for B', that implements the same interface through a different implementation (e.g. an updated version of B), module A will still hold references to the outdated module B. When module A uses those references to invoke a method on the now outdated module B, the invocation will either execute on the outdated module or fail and crash module A. Both options are incorrect.

State transfer. The previous point describes how modular systems typically update existing modules, by replacing each outdated module B by their updated version B'. However, B might hold state that was created during its lifetime by interacting with other modules. Unfortunately, no existing platforms provide any way of transferring that state to B', which results in data loss.

Updating multiple modules. Module platforms replace one module at the time; there is no atomic step that allows a set of modules to be replaced in one indivisible step. Therefore, if two modules are inter-dependent, there is a period of time in which one of those modules is in the new version while the other is in the old version. This limits the flexibility of updates because each module has to be backward compatible with older versions of all the other modules it uses.

Module systems replace modules without pausing the whole application. In fact, all other modules can remain providing service while a module is being replaced. This makes DSU through module systems efficient — ● .

Unfortunately, module systems do not achieve any other of the stated DSU goals. DSU in module systems is not correct because it is hard to reason about the behavior of an update, due to all the reasons that I discussed earlier in this section — ○ . Updates through module systems are also not flexible because each module has to retain the same interface between updates so that other client modules can still use it — ◔ . Finally, module systems are not effective ways of performing DSU because they require updatable applications to follow a particular program architecture — ◔ .

2.7.2 Distributed Systems

A distributed system uses several machines — *nodes* — to provide a service. Distributed systems improve their availability by tolerating node faults and improving their overall scalability by adding more nodes. The presence of redundant hardware poses new challenges and opportunities for DSU in the context of distributed systems, but also ultimately limits the *effectiveness* of DSU systems for distributed systems — ◓ .

Brewer [Bre01] discusses two basic algorithms to perform DSU on a distributed system: *Rolling upgrades* and *big flip*. In a rolling upgrade, each node is updated at a time. The node being updated is taken offline for the update process while the remaining nodes keep providing service. Rolling upgrades thus require the old and new program versions to be compatible. A big flip takes half the nodes offline and updates them while the other half keeps providing service. Then, the up-to-date nodes are brought online to provide service while the outdated nodes go offline to be updated.

The main problem with these two techniques is that they do not explain how to transfer the program state that each node keeps between successive versions. As a result, the downtime required for updating each node is not well-defined, which may lead to inefficient DSU — ○ . Rolling upgrades require the new and old versions to be compatible, which limits their *flexibility* — ◓ . None of these techniques provides any way for the developer to ensure the *correctness* of a DSU — ○ .

Kramer and Magee [KM90] propose a technique for updating a distributed system without requiring any node to be taken offline. In their system model, nodes can only communicate through transactions and are either *active* or *passive*. An active node is operating fully, it can initiate new transactions and participate in ongoing transactions. A passive node continues to participate in ongoing transactions but cannot initiate any new transaction. A node n becomes *quiescent* when: n is passive, all transactions in which n participated or had initiated terminate, and no other node will initiate any transaction that requires service from n. The authors describe how to design a system to achieve quiescence and outline a protocol for updating quiescent nodes.

Quiescence as proposed has several drawbacks, as Vandewoude et al. [VEBD06] discuss. Enforcing quiescence on a distributed system is very disruptive: The node to be updated must be in a passive state, other nodes cannot start transactions that need the participation of the quiescent node, and every node involved in a transaction must be aware of the state of all other nodes involved in that transaction. All these factors increase the coupling between nodes. The authors propose a different property called *tranquility* that, even though weaker than quiescence, is sufficient for DSU. It is based on a black-box

abstraction: Each node knows the services that each other node exports but does not know the state of other nodes nor their involvement in ongoing transactions. Using tranquility, nodes can participate in transactions unaware that their actions are part of such a transaction. Tranquility also requires less nodes to be in the passive state than quiescence to perform a DSU.

Dumitraş and Narashimhan [DN09b] propose a dynamic update system called Imago that is able to perform end-to-end upgrades on distributed systems. Imago requires updatable distributed systems to have an an *ingress interceptor*, between the world and the front-end of the system, and an *egress point*, between the system and the persistent storage. Imago can then instantiate a parallel system, running the new version, and initializes it with state from the datastore. Then, it enforces quiescence either using the ingress interceptor to block incoming write requests or using the egress interceptor to make all data read-only. Finally, it atomically switches to the new system.

Imago is able to deploy the new system in an environment similar to the production environment. It is also able to test the update using two approaches: (1) Static offline system testing, or (2) running both systems in parallel, feeding the same requests to both systems, and testing if both produce the same result at the data-store level. Imago, therefore, provides support for correct DSU — ●. Imago also supports flexible DSU: The authors used it to update the RUBiS benchmark, changing it from the Java implementation to the PHP implementation — ●. The fact that Imago requires enough redundant hardware to run both versions in parallel limits its effectiveness — ◔. However, each version can run without any performance penalty, which makes it efficient — ●. The authors suggest the concept of *Upgrade as a Service* [DN09a], in which the owner of the system undergoing an update can rent the infrastructure needed to perform an update, to alleviate this shortcoming.

2.8 Discussion

This chapter presented the state of the art in Dynamic Software Updating found in the existing literature. There are existing solutions that reach some of the goals that I propose for practical DSU, in Section 1.2. However, no existing solution reaches all of the goals. This section concludes this chapter by discussing the state of the art for each goal. Table 2.1, in page 12, summarizes this discussion.

2.8.1 Flexibility

Flexibility is the goal that most existing solutions for DSU achieve — 16 out of the 35 considered fully achieve it; 13 of the remaining 19 partially achieve it. This goal can be considered the hallmark of Dynamic Software Updating, and most solutions maximize the ways in which a running program can change between successive versions. The solutions that achieve this goal succeed at removing any constraint on the types of changes that a new program version can make to the version in execution at the time of the update.

The most restrictive DSU system of all considered is HotSwap. It can update only the bodies of existing methods. It does not support any type of structure change to the static structure of a Java program, such as adding/removing/modifying fields/methods.

While executing, programs build state that is tightly coupled with their code/behavior. Updating the code without transforming the state limits the flexibility of DSU: The new code has to be compatible with the old program state. OPUS, DynSec, and LUCOS do not support any program state transformation at all; DVM, the DCE-VM, and JRebel support only automatic (default) program state transformation. Versant and Objectivity/DB place limits on what custom transformation code can do to the existing program state at update time.

HotSwap requires all updates to retain the same class signature for all classes between program versions. This is an extreme case of requiring the program to retain the same interface between versions. This constraint can be relaxed by breaking the program into units of abstraction, e.g. modules or classes, and require that only the public interface of each unit of abstraction to remain constant between updates. The public interface is the interface that other units of abstraction use. In this case, DSU can be implemented by simply swapping the implementation by another that provides the same interface. All module systems presented in Section 2.7.1 follow this approach. DUSC follows a similar approach, considering Java classes as units of abstraction: Classes can be updated if the new version retains the same interface as the previous version.

Another way in which DSU systems limit flexibility is by requiring updates to be *backwards compatible*, which means that the behavior that the update introduces is compatible with the behavior of the program version in execution. This eases the challenge of supporting DSU because the program can execute both versions at the same time. Rolling upgrades and POLUS require updates to be backwards compatible.

Some DSU systems limit their flexibility for technical reasons. POLUS and PROTEOS do not support update functions that are always active throughout the entire lifetime of a program, such as `main`. Ginseng allocates extra space at the end of existing structures, so that future updates can increase their size. However, this imposes a hard limit on how updates can change existing structures. JDrums, DVM, DUSC, JVolve, and JRebel do not support updates that change the class hierarchy of a Java program, which is notoriously hard to implement.

2.8.2 Efficiency

Apart from flexibility, discussed in the previous section, efficiency is the goal that most existing solutions achieve — 7 out of the 36 considered fully achieve it; and 17 out of the remaining 29 partially achieve it. I consider solutions for DSU to be efficient if they do not impose any overhead when executing in steady-state, i.e., when not performing an update, and if they do not impose arbitrarily large pauses to perform updates.

Solutions for DSU need to choose the instant at which the update takes place. At that instant, they pause the program to load the new version. After the pause, the program resumes execution in the new program version. This creates an opportunity for transforming the program state while the program is paused and thus ensure that the new program can only access transformed state. However, when performing such *immediate updates*, the execution of the program is paused while the transformation is taking place. The length of the pause is proportional to the size of the program state, which may be arbitrarily large. Ekiden, Kitsune, PROTEOS, JVolve, the DCE VM, and JavAdaptor all impose long pauses when performing DSU.

One option to avoid transforming all the program code at once is to rewrite the program code so that it manipulates the program state through a level of indirection. The system can then use the level of indirection to intercept when the program manipulates the program state for the first time after an update, and transform it on-demand. This approach, however, trades a long update-time pause for a constant level of performance overhead in steady-state execution. Ginseng, K42, and JDrums do exactly this. There are other reasons for the updatable program to run slower in steady-state: Unsupported compiler optimizations (OPUS, POLUS, and LUCOS), updatable program executed inside a sandbox (DynSec), Just-In-Time compiler disabled (JDrums and DVM).

DUSC suffers from these two problems. It both imposes a long update-induced pause and it adds steady-state overhead.

LISP and Smalltalk leave enough room for the implementation to decide how to support their DSU features, allowing implementations that provide inefficient support for DSU. Erlang leaves this problem entirely up to the developer to solve. UpgradeJ can perform DSU without any steady-state overhead or long update-time pauses.

PJama requires that no clients are connected to the data-store undergoing state transformation. GemStone requires explicit manual intervention to transform the program state. Rolling upgrades and big flip omit details about how to transform the program state between program versions. All of these approaches leave room for inefficient DSU.

2.8.3 Effectiveness

An effective solution for DSU can be readily applied to existing code that was not developed with DSU in mind. It targets popular languages and does not restrict the style in which the program is written or the tools required to build and execute the program. 5 of the 36 solutions fully achieve the effectiveness goal; 26 out of the remaining 31 partially achieve it.

This definition directly rules out DSU solutions that target niche applications: OS kernels and OODBMSs. It also rules out DSU solutions that require the updatable program to be written in a particular style or architecture: Module systems and distributed systems. It also rules out DSU solutions that require programs to be written in a different language. The only categories that can fully achieve this goal are DSU systems that target C or Java programs.

For C programs, Ginseng does not reach the effectiveness goal because changes the size of structures defined in the updatable C program, which breaks memory alignment and thus forbids the updatable program to use any C idiom that relies on memory alignment. Ginseng also requires the updatable program to have a form that is easy to understand by its conservative static analysis. For Java programs, HotSwap, JavAdaptor, and JRebel are development-time tools that do not target programs running in a production environment, therefore they cannot be considered effective. JDrums, DVM, JVolve, and the DCE VM require a non-standard execution environment (a custom JVM) to support DSU.

2.8.4 Correctness

The correctness goal can be summarized as ensuring that the updated program behaves as expected. This is a very strong property that requires developers to specify what is the expected behavior of the updated program. Only 2 out of the 36 solutions considered for DSU fully achieve this goal; and 15 out of the remaining 34 partially achieve it.

Correctness is related, to some extent, with the timing of the update. The most common approach to ensure correctness is to limit the instants at which updates can take place while the program is running. One option is to perform DSU only when *the updated code is not active (is quiescent)*. However, this approach does not preclude all bad timings that may crash a program at update time. Worse still, all completely automatic approaches to ensure update correctness have false positives that result in program crashes, as Hayden et al. discuss [HSH+12].

As a consequence, the developer has to be involved in specifying which are the program points where it is safe to perform an update. However, the developer can still make mistakes in identifying the program points where an update can take place and, as a result, crash the running program at update time. Or specify an erroneous transformation logic that corrupts the program state after the update, or makes the updated version misbehave and diverge from the expected behavior in some way. To avoid this scenario, the whole update process must be verifiable or testable. Ginseng allows updates to be tested before applying them, and Imago allows both offline testing to the updated system or running the updated system in parallel with the outdated system, by feeding both the same input, and then compare their

output. These are the only two approaches that provide support for ensuring the correctness of the update process and the updated program.

All other approaches fail to provide the developer with any way to ensure that the update will not crash the program and that the new program will behave as expected after the update. Some options do not restrict the instants at which the update may take place: POLUS, DynSec, HotSwap, the DCE VM, JavAdaptor, JRebel, all the module systems. Others rely on the (unsafe) activeness checking: OPUS, K42, LUCOS, DynaMOS, KSplice, JDrums, DVM, and DUSC, all follow that approach. JVolve follows a similar approach but allows the developer to blacklist functions that, even though did not change, cannot be active when an update takes place. The approaches that require developers to identify program points where updates can happen do not provide any other way to ensure that these program points lead to correct updates: UpStare, Ekiden, Kitsune, PROTEOS, Erlang, and UpgradeJ. The remaining approaches — LISP, Smalltalk, rolling upgrades, and big-flip — do not specify exactly when updates take place; they instead leave that decision up to each particular implementation. UpgradeJ provides a strong type-system that forbids a vast category of incorrect updates due to timing and interaction between old and new code. However, it does not provide any way for the developer to test the correctness of the updates.

2.8.5 Rest of this document

The last three rows of Table 2.1 provide a brief overview on the rest of this document.

Chapter 3 presents *DuSTM*, a DSU system for the Java programming language that supports flexible class and hierarchy updates between versions. DuSTM requires the updatable system to be structured around transactions, which limits its effectiveness but provides a stronger guarantee of correctness than manually identified update points. DuSTM does not require a custom JVM; it re-writes the bytecode instead, to add an extra level of indirection to support future updates with lazy update semantics. However, that extra indirection comes at a cost of steady-state overhead.

Whilst DuSTM is a step in the right direction, it has a number of shortcomings; most notably, the requirement on a particular architecture for the updatable application. Chapter 4 presents *Rubah*, a DSU system for Java that lifts that requirement and still provides the same flexibility and support for lazy update semantics. Besides describing the design of Rubah and its prototype implementation, Chapter 4 also provides empirical evidence of Rubah's effectiveness by applying it to 5 real-world applications that were originally developed without a support for DSU, together with an experimental evaluation using those applications to benchmark the performance of Rubah and thus show show that it is also efficient.

When compared to DuSTM, Rubah improves on effectiveness and efficiency at the cost of correctness. Rubah requires retrofitting applications with support for DSU and adding annotations about where, in the program, is safe to perform an update. DuSTM can take any transactional application and does not require any change to support updates, extracting all information it needs about update points from the transactions that make up the application.

To bridge the gap between effectiveness/efficiency and correctness, Chapter 5 presents *Tedsuto*, which is a systematic testing framework for DSU systems. Tedsuto takes existing system tests and re-runs each test several times, performing an update during the test and checking if the test still passes. Chapter 5 describes a prototype implementation of Tedsuto for Rubah which did find real bugs in 2 of the 5 applications used to evaluate Rubah. Tedsuto does not rely on any particular part of Rubah and can be implemented for other DSU systems.

Together with Tedsuto and borrowing ideas from DuSTM, I claim that Rubah is the first practical solution for Dynamic Software Updating. The rest of this document explains why.

Chapter 3

Composable Updates

The applications with high availability requirements are the ones that most benefit from DSU support. These applications are typically highly concurrent so that they can take full advantage of the hardware and network resources available. Developing correct concurrent applications is itself a challenge. One way to ease that burden is to structure the application in sequences of steps that are logically related and that execute in isolation — *transactions*. When a transaction finishes, the system validates it against all other transactions that executed concurrently and either makes the transaction globally visible to the rest of the system in an atomic step — *commits* — or re-runs the transaction using the most recent data — *rollbacks*. Rollbacks are due to conflicts in which other transactions that have already committed overwrote something that the finished transaction reads or writes, in which case the transaction cannot be allowed to commit because it executed over stale or inconsistent data.

Database Management Systems (DBMSs) are great examples of such transactional systems. DBMSs execute transactions providing **A**tomicity, **C**onsistency, **I**solation, and **D**urability. This set of properties is typically referred to as *ACID*. I have already described isolation (transactions execute without seeing the changes made by other concurrent transactions) and atomicity (transactions either take place atomically or are rolled-back without changing the system). ACID systems also ensure that transactions preserve data integrity (consistency) and persist the changes that transactions make to the system (durability). More recently, researchers explored the idea of bringing the atomicity, consistency, and isolation properties to the application level with *transactional memory* to simplify developing highly-available concurrent applications without requiring a DBMS to support transactions.

Transactions are therefore composable units of work that group logically related actions. Transactions expect a consistent state when they start and are required to leave the system in a consistent state when they finish. This greatly simplifies reasoning about concurrent systems: The developer only needs to care about the interleaving of transactions as a whole. This same property can also be used to simplify reasoning about the correctness of DSU. If we assume that each transaction always executes on the same program version in which it starts, the only program points at which a DSU can occur are those between transactions.

When reasoning about DSU and transactions, it makes sense to consider the update itself as a transaction that changes the program code that new transactions will execute and that migrates the program state to an equivalent version that is compatible with the new program code. An important observation here is that migrating the program state has to happen, regarding other transactions, in the same instant as the transaction that performs the DSU. We can imagine a system that keeps that illusion but, in fact, migrates objects just before they are needed for the first time after the DSU takes place. By performing such a *lazy program state migration*, this system minimizes the time required to perform a DSU on a program with an arbitrarily large program state.

The major advantages that transactions bring to DSU can thus be summarized as: (1) With transactions, it is easy to find points in the program execution where it is safe to perform the DSU, and (2) the program state can be migrated lazily with the same semantics as if it was completely migrated at update time.

In this chapter, I describe the design and implementation of a system — *DuSTM* — that enables DSU for transactional Java programs that use a versioned TM called JVSTM. I start by claiming how DuSTM achieves some of the goals for a practical DSU system in Section 3.1. I then introduce the notation I shall use throughout this chapter in Section 3.2. In Section 3.3, I describe the update semantics that DuSTM provides and how the developer can use it to reason about DSU. Then, I describe how to implement DuSTM, in Section 3.4; evaluate the performance of a prototype implementation, in Section 3.5; and discuss DuSTM as a solution for practical DSU, in Section 3.6.

This chapter assumes that the reader is familiar with current Transactional Memory (TM) terminology and implementation. Appendix A provides all the detail about TM needed to follow the rest of the discussion.

3.1 Claims

Throughout the remainder of this chapter, I shall describe DuSTM in detail. This section explains how DuSTM reaches the goals that I defined earlier in Section 1.2.1, with particular focus on how each following section contributes to each particular goal. This section also discusses the extent to which DuSTM reaches each goal.

Flexibility DuSTM considers updatable application to be composed of *updatable classes* and *non-updatable classes*. Updatable classes are the classes that the updatable application defines. Non-updatable classes are all the other classes that the updatable application uses, such as libraries and classes belonging to the Java runtime environment. Only updatable classes can change due to program updates. Section 3.3.6 explains the difference between updatable and non-updatable classes in further detail.

DuSTM supports a wide range of program changes made to updatable classes. It supports any class structural changes, such as adding/removing/changing the signature of fields/methods. It also supports changing the bodies of existing methods. Besides changes local to a single class, DuSTM supports updates that change the class hierarchy with a few restrictions. The types of program modifications that DuSTM supports are discussed in further detail in Section 3.4.

Besides all the different program changes it supports, DuSTM also supports custom program state migration between successive program versions. In fact, DuSTM provides tools, such as the *update class* and *old classes*, that allow the developer to describe the program state migration using regular Java code that resembles the actual program classes being migrated. The program state migration that DuSTM supports is described in further detail in Section 3.3.6.

I claim that DuSTM fully achieves the goal of flexibility as defined in Section 1.2.1 — ● following the notation introduced by Table 2.1.

Correctness DuSTM considers updates to happen inside an *update transaction* that installs the new code and migrates the program state. Transactions that start after the update transaction commits execute in the new program version and can access only the migrated program state. Transactions that start before the update transaction and finish after always execute the old program.

DuSTM ensures that no program code executes on the wrong program version. Custom program state migration code, however, has to execute between two program versions by design. DuSTM migrates each object in its own separate *migration transaction*. Every migration transaction behaves as if it was the

first migration transaction to execute: It expects all the program state to be still in the previous program version and uses that program state to transfer the state of an outdated object to an updated blank object that will take the identity of the outdated object in the new program version. Section 3.3.5 describes the semantics of DSU in DuSTM in further detail.

DuSTM naturally extends the transactional model to support DSU, ensuring that the updated code does not execute over outdated program state; and it does not require the developer to adapt the original program in any way to support future updates. These are important steps towards correctness. However, DuSTM still requires the developer to write program code that executes in-between versions, to migrate the program state; and does not provide the developer any way to verify, test, or otherwise ensure that a program will not misbehave after an update.

I claim that DuSTM partially achieves the goal of correctness as defined in Section 1.2.2 — ◐ following the notation introduced by Table 2.1.

Effectiveness DuSTM targets transactional programs written in the Java programming language and does not require a custom JVM or a custom compiler to support DSU. DuSTM uses a *post-processor* to turn a mature program version into an updatable program version by rewriting the bytecode. Section 3.3.6 describes how to turn a program version into an updatable version using DuSTM, and Section 3.4 explains the bytecode rewriting process in detail.

I claim that DuSTM only partially achieves the effectiveness goal as defined in Section 1.2.4 — ◓ following the notation introduced by Table 2.1.

Efficiency DuSTM supports lazy program state migration. On programs that have an arbitrarily large program state, migrating the program state lazily avoids a proportionally arbitrarily large pause in program execution to perform a DSU. Instead, DuSTM amortizes that pause over the execution of the new program version. Section 3.5.1 describes the experimental evaluation that shows that the pause induced by a DSU is constant, despite the overall size of the program state, when using program state migration.

DuSTM supports DSU by introducing an extra level of indirection in the form of *handles*. Section 3.4 describes how DuSTM uses the post-processor to inject handles, and transform the original program, while retaining the original semantics.

Although lazy program state migration is necessary for efficient DSU, it is not sufficient. The overhead for transactional applications is within reasonable bounds, as Section 3.5.1 explains. However, Section 3.5.2 describes an experimental evaluation that shows that the cost of adding the required extra level of indirection to existing non-transactional applications can add up to 56% performance overhead on steady-state execution.

I claim that DuSTM partially achieves the efficiency goal as defined in Section 1.2.3 — ◓ following the notation introduced by Table 2.1.

3.2 Notation

Throughout this chapter, I shall discuss concurrent executions in which several threads perform operations in parallel. This section introduces the graphical notation that I shall use in the diagrams that depict concurrent executions.

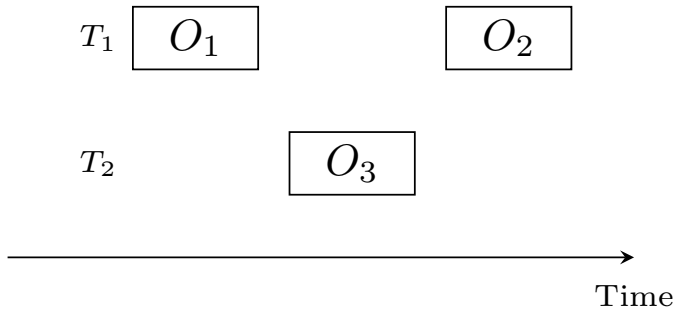


Figure 3.1: Notation for executions of concurrent threads. In this example, thread T_1 performs operations O_1 and O_2, and thread T_2 performs operation O_3.

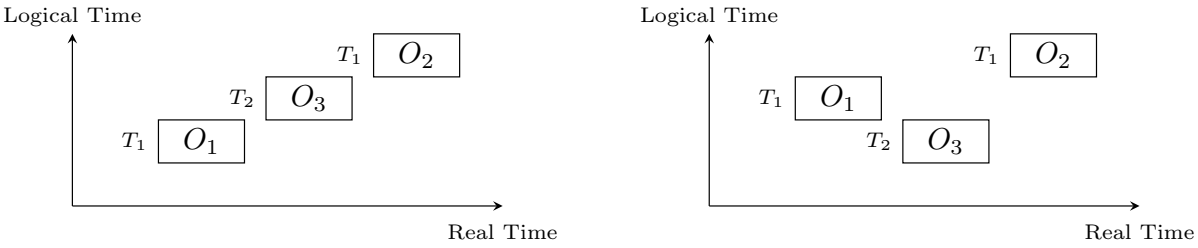


Figure 3.2: Notation for real-time order and logical order. This figure shows how the same real-time order of operations might result in a different logical order. On the left-hand side, logical order matches real-time order: Operation O_3 sees the result of operation O_1 but not operation O_2. On the right-hand side, operation O_3 does not see the result of operation O_1 or O_2.

Figure 3.1 shows the basic diagram of a concurrent execution involving two threads. Operations performed by the same thread are in the same row. In this case, thread T_1 performs two operations: O_1 and O_2. Thread T_2 performs a single operation: O_3. The horizontal axis denotes time. In this particular execution, we can see that no thread executes any operation in parallel because because there is no overlap in time between operations on different threads.

The execution that Figure 3.1 shows suggests that operation O_3 sees the result of operation O_1 but not the result of operation O_2 because O_3 starts after O_1 finishes and ends before O_2 starts. This might not be always true. It is possible that the *logical order* in which operations take place — the order in which a thread sees operations performed by other threads — might not match the *real-time order* in which those operations were executed — the order in which the CPU actually executed each operation on each thread.

Figure 3.2 shows two possible logical orderings for the execution that we are following. In this figure, the horizontal axis now refers to real time and the vertical axis refers to logical time. We can see that the same real-time order might lead to different logical orders. Given that rows now mean logical time instants, each operation is labelled with the thread that executes it. However, in some executions, the mapping between threads and operations is not relevant to the diagram. In those cases, I shall omit it.

Concurrent operations can overlap in real-time order, but never in logical order. The notation with the two time axes that denote logical- and real-time orders is thus limited to atomic operations. I shall use the single axis notation to discuss executions with non-atomic operations or when both real-time and logical orders are the same.

```
1  class Point {
2    private final double x, y;
3
4    public Point(double x,double y) {
5      this.x = x; this.y = y;
6    }
7
8    public double dist(Point p) {
9      return sqrt(square(p.x-x)+square(p.y-y));
10   }
11 }
12
13 class Rectangle {
14   private final Point topLeft, botRight;
15
16   public Rectangle(Point tl,Point bt) {
17     topleft = tl; botRight = br;
18   }
19
20   public double area() {
21     Point tr = new Point(botRight.x,topLeft.y);
22     return topLeft.dist(tr) * tr.dist(botRight);
23   }
24
25
26 }
```

Figure 3.3: First version of the updatable application example. Methods `sqrt` and `square` compute the square root and the square of their argument, respectively.

3.3 Atomic Dynamic Software Updates with DuSTM

This section introduces *DuSTM*, which is a technique for performing DSU on transactional programs implement using JVSTM as their transactional memory.

The *Java Versioned Software Transactional Memory* (JVSTM) [CRS06] is a software implementation of a Transactional Memory that provides optimistic concurrency control through lazy version management, lazy conflict detection, and implements a flat nesting model. JVSTM uses special memory locations to keep transactional values. These memory locations are called *Versioned Boxes*, or just *VBoxes*. A conventional memory location keeps a single value which is the last value that was written to it. A VBox is unconventional because it keeps a *history* of values that were written to it. I refer the reader to Appendix A, in particular Section A.4 in page 167, for details about the implementation and particular semantics of JVSTM. For the reader familiar with STM terminology, it suffices to know at this point that JVSTM provides opaque transaction semantics [GK08].

3.3.1 Updatable Application Example

To guide the upcoming discussion and to provide concrete code that highlights the challenges of supporting DSU, this section introduces a simple application that shall be used throughout the remainder of this chapter as a running example.

The application represents geometric data, namely points and rectangles. Figure 3.3 shows the first version of the application. Points are represented using rectangular coordinates. Rectangles are represented using two opposite vertexes.

```
1  class Point {
2    private final double rho,theta;
3
4    public Point(double rho,double theta) {
5      this.rho = rho; this.theta = theta;
6    }
7
8    public double dist(Point other) {
9      double raa = square(this.rho), rbb = square(other.rho);
10     double rab = this.rho*other.rho;
11     double cosab = cos(this.theta-other.theta);
12     return sqrt(raa+rbb+2*rab*cosab);
13   }
14 }
15
16 class Rectangle {
17   private final Point topLeft, botRight, botLeft;
18
19   public Rectangle(Point tl,Point bt,Point bl) {
20     topleft = tl; botRight = br; botLeft = bl;
21   }
22
23   public double area() {
24     return topleft.dist(botLeft) * botRight.dist(botLeft);
25   }
26 }
```

Figure 3.4: Second version of the updatable application example. Methods `sqrt`, `square`, and `cos` compute the square root, the square, and the cosine of their argument, respectively.

Note that both points and rectangles are immutable, so they can be freely shared across threads without any synchronization or without adding VBoxes. This makes the code on Figure 3.3 to be as simple as possible. The rest of this chapter, however, shall consider that points are part of a larger application that creates and manipulates them inside transactions. For instance, we can consider that the application uses a transactional list to keep points sorted by their distance to the origin $(0, 0)$ and rectangles sorted by their vertexes.

Figure 3.4 shows a possible second version of the example application. The new version changes the internal representation of both points and rectangles. Points are now implemented using polar coordinates. Rectangles are represented using one extra vertex, so that the application supports rotated rectangles.

3.3.2 Atomic Updates and Quiescence

To support DSU, we have to deal with several challenges as Section 1.1 explained. One of the major design challenges is how to find *update points* — program points in which it is safe to perform a DSU.

Finding update points is simpler for the case of programs structured around memory transactions than for the general case. The major concerns in this case are: (1) To ensure that transactions observe *transactional version-consistency (TVC)* [NHFP08], which means that each transaction starts and finishes in the same program version;[1] and (2) to ensure that transactions that start after an update only execute new program code. Figure 3.5 shows an example of the two possible alternatives of enforcing single-version consistency for transactions: Either the system requires transactions to *quiesce* to install an update, or it isolates transactions running at the time of the update and thus in the old program version — *outdated transactions* — from the new program version.

[1] Transactional Version Consistency (TVC) was explained in further detail in Section 2.2.2, page 19.

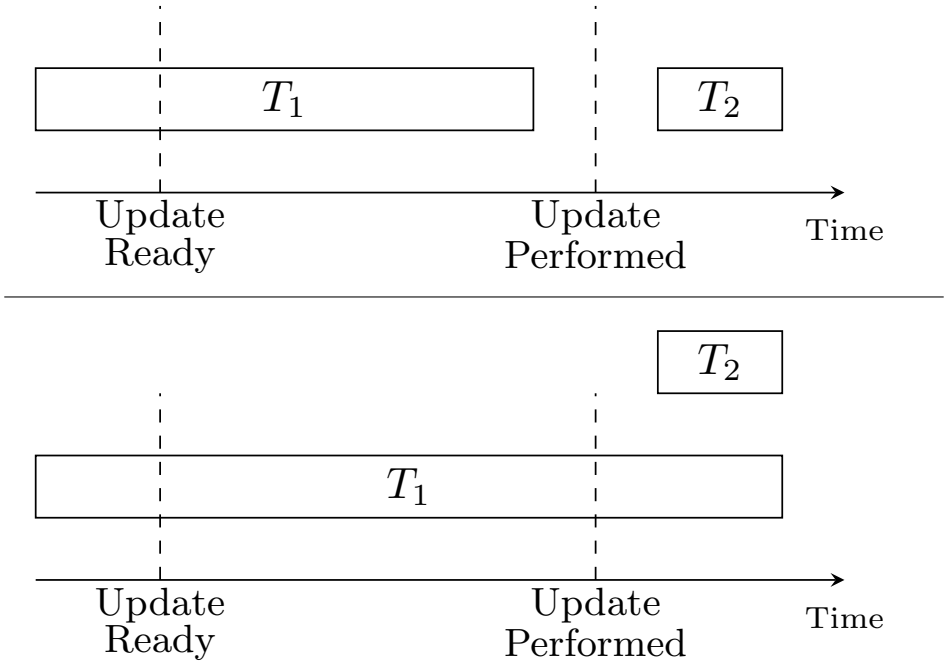


Figure 3.5: Possible options for the atomic DSU semantics. The instant when an update becoming ready is different from the instant when the system performs DSU. The top half of this figure shows an alternative where the system reaches quiescence, i.e., no transactions running, before performing a DSU. The bottom half shows the alternative where the system isolates running transactions at the time that the DSU takes place so that they keep executing in the old program version. In both alternatives, transaction T_1 always executes the old program version and transaction T_2 always executes the new program version.

Ensuring each transactions that starts after DSU only executes the new program guarantees progress. Without this constraint, it is not possible to distinguish a system that does not support DSU from a system that supports it but keeps executing all transactions in the old version after the DSU.

Ensuring that DSU can only happen at the granularity of whole transactions relieves the developer of the burden of mapping program points in the old program code to equivalent program points in the new program code. Even though this solves the problem of mapping program points, it does not address the problem of mapping program-state between successive versions.

3.3.3 Immediate Update Semantics

The simplest way to map the program-state between versions is to do it while the program is performing the update. If transaction quiescence is required to perform a DSU (top half of Figure 3.5) then there are no transactions reading/writing the program-state while the update takes place. The alternative case (bottom half of Figure 3.5) is not much more complex. Note that, in this case, the system already has to provide isolation between transactions running on the old program version and transactions running on the new program version. A trivial solution is to abort all transactions that were still running the old program code when they finally commit after the update takes place.[2]

[2]In fact, the system only has to abort transactions that perform any write. If an outdated transaction does not perform any write, the transactional system already has ensured that it never sees an inconsistent version of the program-state. Such a read-only outdated transaction can still be allowed to commit.

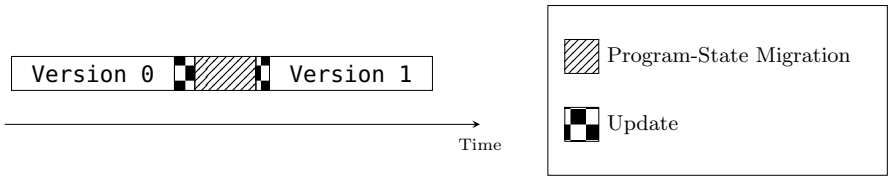


Figure 3.6: Immediate update semantics. All the program-state is migrated when the DSU takes place.

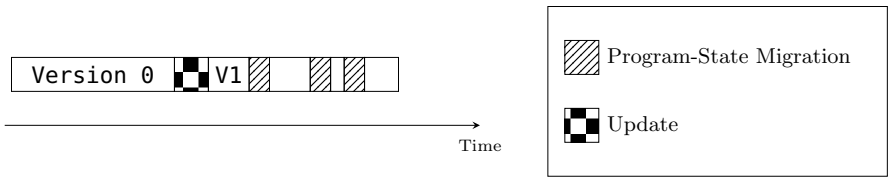


Figure 3.7: Lazy update semantics. The program-state is migrated lazily as the natural control-flow of the new version reaches each portion of it for the first time after the DSU. V1 is short for Version 1, which starts executing after the DSU takes place.

Figure 3.6 shows an example of the atomic update semantics that highlights program-state migration. The program executes version 0 for a while, which creates some program-state. Then, the system performs a DSU that migrates all the program-state at once. After that, the new program version starts to execute. It is obvious that, at this point, the new program version can access only program-state that was migrated during the DSU. This atomic update semantics, in which the DSU migrates all the program-state before starting to run the code of the new program version, is called *immediate update semantics*.

3.3.4 Lazy Update Semantics

The state that a program keeps can grow to be arbitrarily large. As I discussed on Section 1.2.3, requiring the DSU to migrate all the program-state before the program starts executing the new version might, therefore, impose an arbitrarily large pause in program execution and, ultimately, partially defeat the main motivation for supporting DSU.

Fortunately, there is a way to solve this problem. Note that the program-state is divided into logical portions. In Java, and other object-oriented languages, those portions are called *objects*. Given that DuSTM targets Java, I shall refer to those logical portions as objects as well.

The key observation is that the new program version will not need the whole program state to be migrated in its entirety when it starts executing. In fact, a DSU can delay migrating each individual object until the instant at which the natural control-flow of the new program version reaches that object for the first time after the update. This is called *lazy update semantics*. Figure 3.7 shows how the example from Figure 3.6 looks like with lazy update semantics.

The choice between immediate and lazy update semantics is orthogonal to requiring transactions to quiesce or not before performing a DSU. In particular, note that immediate non-quiescent updates make sense. This combination migrates the program-state between versions in parallel with old transactions still running. This is not a problem as long as the underlying transactional memory isolates the old transactions from the migrated program state. Quiescence in this case can be seen as a clear way of enforcing isolation between transactions executing in different program versions, which is equivalent to the isolation that the transactional memory already provides to support non-quiescent immediate updates.

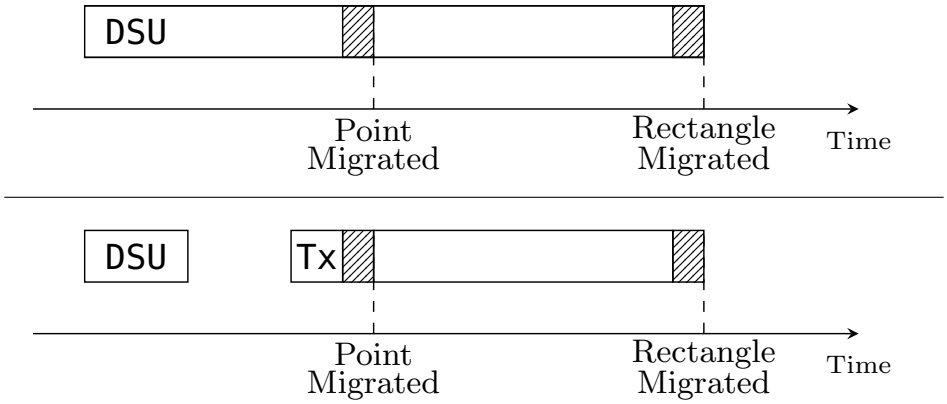


Figure 3.8: Conversion-ordering problem for both immediate (top) and lazy (bottom) semantics. The program-state in this example is, at the time of the update, one rectangle with two points. In both cases, the point is migrated before the rectangle. When the rectangle is migrated, each of its points is in a different program version.

3.3.5 Program-State Migration Semantics

A flexible DSU system, in the sense that Section 1.2.1 defines, allows the developer to customize how to migrate the program-state between successive versions. The main goal of custom program-state migration is to allow the developer to write code that operates over the old program-state and generates an equivalent program-state that is compatible with the new program code.

As discussed in the previous section, let us consider that the program-state is divided into logical portions called objects. A simple approach to ensure flexibility is to provide support for the developer to customize migrating each object individually. That is, the developer provides code that, given an object O_{old}, produces an equivalent object O_{new} that will take the place of the outdated object O_{old} in the new program-state.

Limiting the developer to only use the outdated portion is very restrictive. For instance, consider the geometric application example introduced in Section 3.3.1. To migrate the rectangle between versions, the developer needs to access the points of the old rectangle to compute the extra points that the new rectangle keeps in its internal representation.

Giving the developer unrestricted access to the program-state increases flexibility at the cost of complexity and safety. The order in which each object is migrated now becomes relevant. For instance, following the same example, let us now consider that one of the points in a rectangle gets migrated before the rectangle. This can happen if, for instance, the point is aliased somewhere else in the program-state. In this case, by the time the rectangle migration code runs, each point is in a different program version. This is the *migration-ordering problem*. Figure 3.8 depicts the migration-ordering problem for both the immediate and lazy semantics.

There is an interesting middle ground between the two extreme positions of restricting the custom migration code to access only the object being migrated and giving it complete access to all the program-state. The key observation is that the new program-state is equivalent to the old program-state at the time the update takes place. The two are merely different representations of the same state. We can thus limit the custom migration code to access only the old program-state without any loss of expressivity. The only part of the new program-state that the custom code can access is the object being migrated.

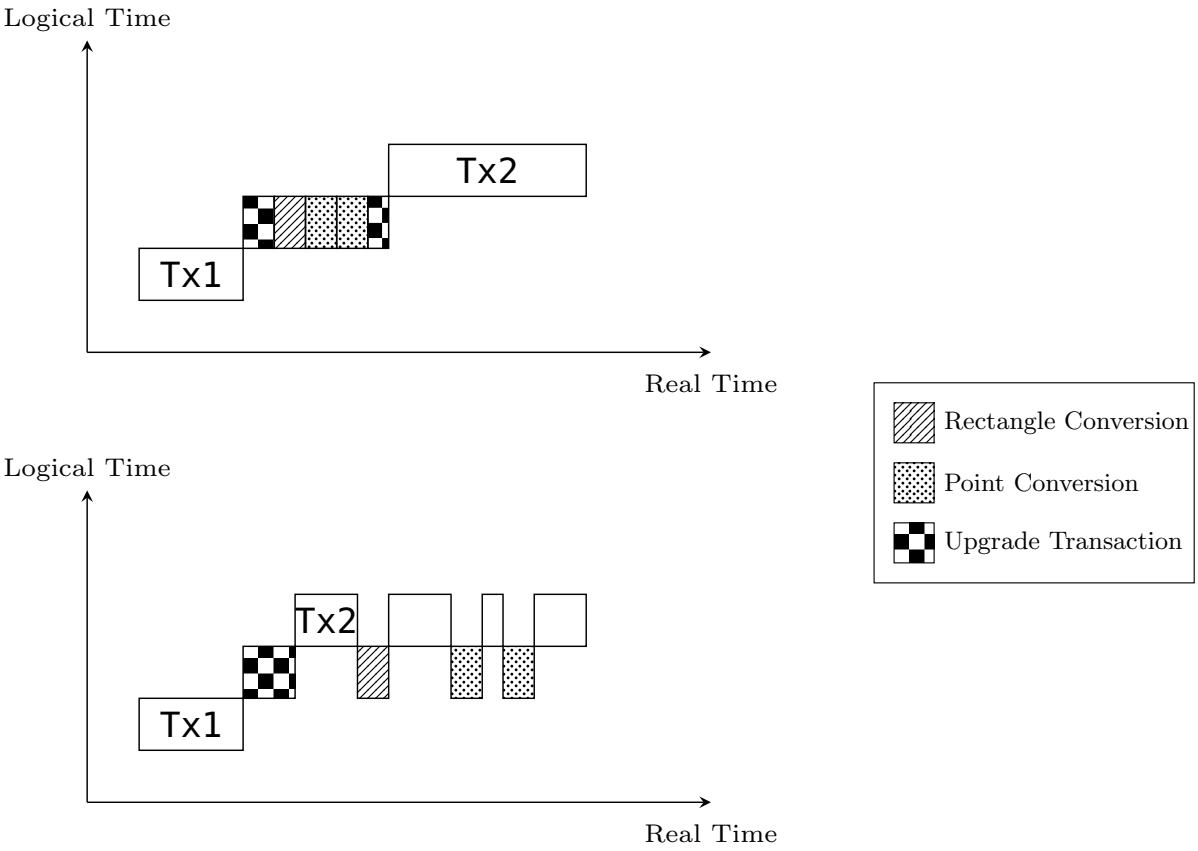


Figure 3.9: Atomic update semantics for both immediate program-state migration (top) and lazy program-state migration(bottom). The program-state before the update is, in this example, a rectangle with its two respective points.

The transactional memory model allows us to define this update semantics in terms of transactions and their ordering. The idea behind this semantics is very simple: All objects are migrated in the same logical instant as the one in which the update takes place. First, let us consider that the update happens within a special transaction, the *update transaction*, which installs the new code and migrates the program state. Also, let us assume that every object is migrated in its own separate transaction, the *migration transaction*.

The *atomic update semantics* that I propose simply states that all migration transactions must take place in the same logical instant as the update transaction that install that version. Figure 3.9 shows this update semantics for both immediate and lazy updates.

A consequence of the atomic update semantics is that the real-time order in which each object is migrated is no longer relevant. In fact, any possible real-time order should result in exactly the same migrated program-state.

The atomic update semantics greatly simplifies reasoning about migration code. In particular, it means that each migration transaction T sees the program state as if transaction T was the first migration transaction to execute after the update. This means that transaction T can find all the old program-state as it was at the time of the update and it cannot find any new program-state because it does not exist yet, except for the particular object being migrated.

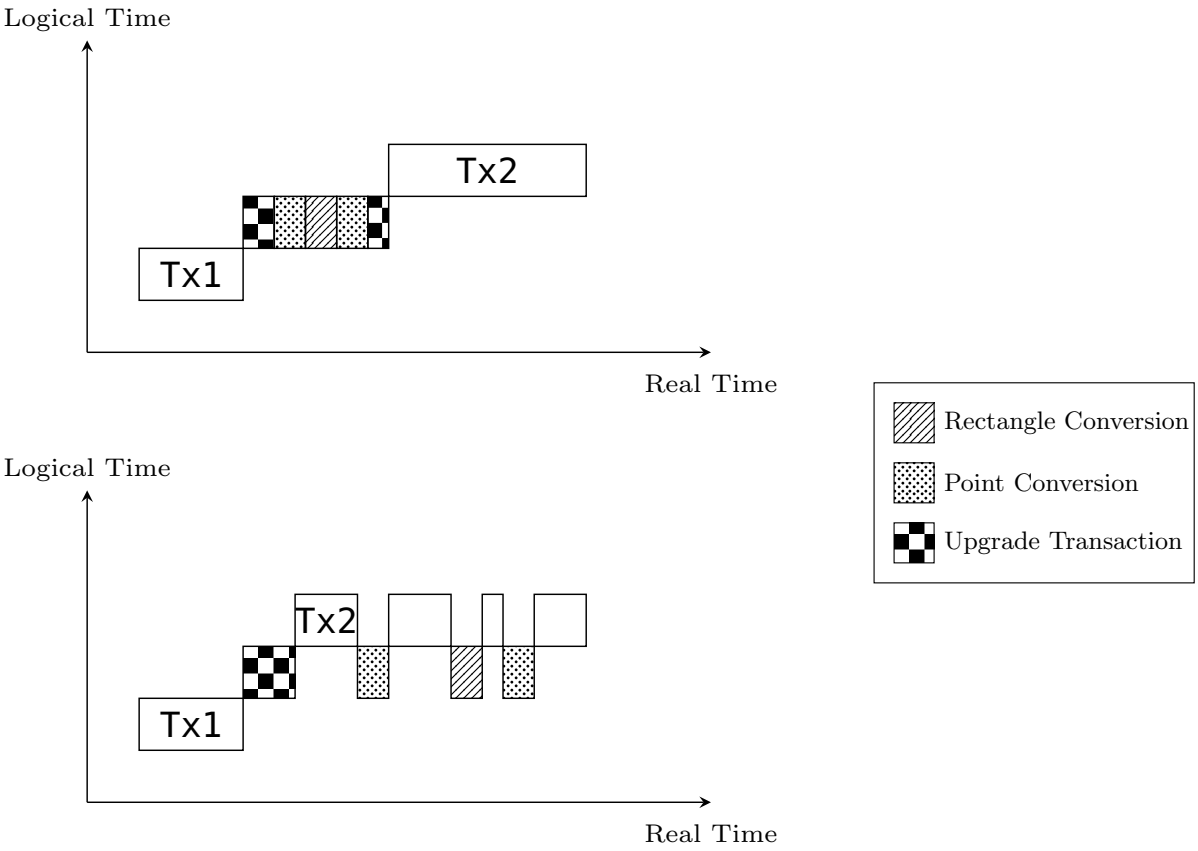


Figure 3.10: Solution to the conversion ordering problem using atomic update semantics for both immediate program-state migration (top) and lazy program-state migration (bottom). Even though some portions of the program state are migrated in a different order, these executions should result in the same migrated program-state as the ones shown in Figure 3.9.

Figure 3.10 shows how atomic update semantics naturally solves the migration-ordering problem. When the rectangle gets migrated, the custom migration code sees the old rectangle with the old points as they existed at the time of the update even though one of the points was already migrated.

3.3.6 Developing and Updating Applications

DuSTM [PC11a, PC12] implements the atomic update semantics that the previous section introduced. This section describes how to build an application that can perform DSU using DuSTM.

Updatable Application Structure

DuSTM provides support for DSU on transactional applications that use JVSTM. By transactional, I mean applications that: (1) Delimit transactions that perform consistent, atomic modifications to the program-state; and (2) protect program-state shared between transactions with VBoxes.

Besides this restriction, the application must be structured in two separate layers, as Figure 3.11 shows. The *updatable code*, which is the top layer, contains all the classes (and interfaces) that the application defines. The *execution platform* contains the remainder of the code that the application requires to execute: Third-party libraries, classes that belong to the Java platform and DuSTM itself. Classes that belong to the updatable code can directly reference classes that belong to the execution platform, but not the reverse. Of course, library classes do not directly reference application classes, so this restriction poses no practical difficulty.

Updatable Code (e.g. `Point`)
Execution Platform (e.g. `java.lang.String`)

Figure 3.11: Layered structure of an application updatable through DuSTM. The top layer contains all the code that the application defines. The bottom layer contains all other code that the application needs to execute (third-party libraries, Java runtime environment, and DuSTM itself). Classes in the bottom layer cannot reference directly any class in the top layer. DuSTM supports updates to only classes in the top layer.

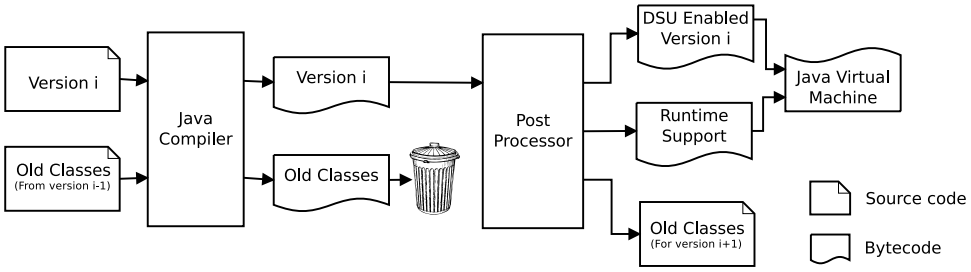


Figure 3.12: Process and tools for generating an updatable version of a program version using DuSTM. This process uses an unmodified Java compiler and the post-processor tool that DuSTM provides. The post-processor rewrites the application bytecode in a way that is semantics-preserving and that supports performing DSU in the future without requiring a modified Java Virtual Machine.

DuSTM can update only classes that belong to the updatable code layer. It cannot update any class that belongs to the execution platform. The developer can thus use DuSTM to update any class that the application defines with few restrictions[3] as long as the new version requires the same execution platform.

Development Process

DuSTM does not require a custom JVM to support DSU. Instead, it rewrites the program bytecode to generate a DSU-enabled version that has the same semantics as the original program. Figure 3.12 shows the full process and tools required to prepare a DSU-enabled version.

Besides rewriting the bytecode of the original program, the post-processor tool that DuSTM uses also generates helper *old classes* that allow the developer to refer the correct version of the program-state when writing the migration code. The following section presents a more detailed example that shows how to use the old classes.

The developer can execute the tool-chain shown in Figure 3.12 only when a mature version of the program is ready to be deployed into production via DSU. Before that point, he does not need to post-process the bytecode. Consequently, he can use the regular development process to develop the new version of the application. That means that DuSTM does not interfere with any development-time tools such as IDEs, debuggers, and profilers. Section 3.4 explains in detail how DuSTM rewrites the original program code.

Migrating the Program State

DuSTM supports custom program-state migration between versions. That means that the developer can write code to migrate the program-state existing at update time to a version that is equivalent but compatible with the updated program.

[3]Section 3.6 explains these restrictions in further detail.

```
1  class Point {
2    private double rho,theta;
3    ...
4    static void convert(old.Point o,Point n) {
5      n.rho = sqrt( square(o.x) + square(o.x) );
6      n.theta = arctan( o.y / o.x );
7    }
8  }
9
10 class Rectangle {
11   static void convert(old.Rectangle o,Rectangle n) {
12     n.topLeft  = o.topLeft.convert();
13     n.botRight = o.botRight.convert();
14     old.Point botLeft = new old.Point( o.topLeft.x, o.botRight.y);
15     n.botLeft = botLeft.convert();
16   }
17   ...
18 }
```

Figure 3.13: Conversion code for the geometric application example. The types declared on package `old` — the *old types* — are shown in Figure 3.14. The ellipsis denotes the rest of the code defined for each type on Figure 3.4.

Let us denote a class that the update modifies by C. The old version of C, that exists before the update, is C_0. The new version of C, that the update introduces, is C_1. For each such class C that an update modifies, the developer writes a *migration method* named `convert`, in C_1, to migrate the program state from instances of C_0. Migration methods take two arguments: An instance of C_0 that holds the state to be migrated, and an instance of C_1 that will replace C_0 in the migrated program-state. DuSTM then runs the migration methods to migrate each object in the program-state between successive versions inside a migration transaction, as explained in Section 3.3.5.

Figure 3.13 shows the migration methods for the geometric application example that Section 3.3.1 introduced. Each method defines the migration logic between the previous version of the same class. Note that the first argument of each of the migration methods belongs to package `old`. This is, in fact, a synthetic *old class* that DuSTM's post-processor generates, as shown in Figure 3.12.

Old classes allow the developer to refer to each version of the program-state unambiguously. They provide symbolic integration for both versions of the program, the old and the new, at the level of the migration code so that the developer can use the unmodified Java compiler to generate the bytecode for the new version.

For each class in the old program version, the post-processor generates a corresponding old class with the same fields and methods but with two differences: (1) fields and methods are publicly visible, so that the developer can fully access the old program version; and (2) methods have empty bodies[4]. Figure 3.14 shows the old classes that DuSTM generates for the geometric application example that we are following.

On each old class it generates, DuSTM adds an extra method named `convert`. This method allows the developer to specify that a field in the new version retains the same object it had in the previous version. To understand the need for these methods, let us consider the code that migrates a Rectangle, shown in Figure 3.13. More specifically, consider line 12. Note that the line `n.topleft = o.topLeft` does not compile because `n.topLeft` has type `Rectangle` and `o.topleft` has type `old.Rectangle`, which are incompatible. In this case, the developer uses the `convert` method to write `n.topLeft = o.topLeft.convert()`, as shown in the Figure.

[4]The body of methods with a return type different than `void` is a single return statement of either `0` or `null`.

```
1  class old.Point {
2    public double x, y;
3
4    double dist(old.Point p) {return 0.0;}
5
6    public Point convert() { return null; }
7  }
8
9  class old.Rectangle {
10   public old.Point topLeft, botRight;
11
12   double area() { return  0.0; }
13
14   public Rectangle convert() { return null; }
15 }
```

Figure 3.14: Old classes that DuSTM generates for the geometric application example. The old classes have the same fields and methods as the original classes, except that they are made publicly visible and methods have a dummy implementation. Each old class has an extra method `convert` to allow the developer to state that new fields should retain the same object they had in the previous version. Refer to line 12 on Figure 3.13 for an example of its usage.

3.4 Implementing Atomic Updates

DuSTM is implemented as a bytecode post-processor and a runtime library. The post-processor takes as input the bytecode that the Java compiler generates and produces classes enhanced with DSU support via a semantics-preserving binary refactoring performed using ASM [BLC02, Kul07]. This section describes how the post-processor modifies the original bytecode.

3.4.1 Handles as Transactional Proxies

A Java program can be broadly defined as a set of classes related to each other in a hierarchy. A *well-formed Java program* only contains classes that refer to only other classes existent in the same Java program.[5] It is trivial to generate well-formed Java programs using the Java compiler: If a program compiles, then it can only be well-formed. An update modifies a subset of those classes by changing their structure (adding/removing fields or methods, changing the class hierarchy) and their behavior (changing the code of existing methods).

A *program update* can be defined as a set of *class updates*. Each class update defines a new version of an existing class. A *well-formed program update* contains all class updates that make the new version of the program well-formed. Again, this is trivial to ensure using the Java compiler: If the new program version compiles, then it can only be a complete Java program and we can get a complete program update by considering all classes that changed between versions as class updates.

Let us consider that a class update C_{upd} changes class C_0 into class C_1. When DuSTM performs a DSU, the program has class C_0 loaded and is executing its methods inside JVSTM transactions. Performing a destructive change on class C_0 to transform it into class C_1 would restrict DuSTM to quiescent updates, as defined in Section 3.3.2 (top half of Figure 3.5). Besides, the JVM does not support making destructive changes to loaded classes. As a consequence, implementing this approach would require a custom JVM.

DuSTM follows a different approach to support class updates when performing DSU: It introduces a new Java class C_1 for a class update that a program update contains for class C_0. The bytecode post-processor renames all classes to avoid name collisions between their current version and possible future

[5] Let us consider that the Java runtime is a part of a Java program, for the sake of simplicity and clarity of presentation.

```
1  class Handle {
2    static VBox<Integer> systemVersion;
3    VBox<Pair> pair;
4
5    static Object getObject(Handle h) {
6      Pair p = h.pair.get();
7      if (systemVersion.get() > p.version) { p = h.convert(); }
8      return p.object;
9    }
10
11   Pair convert() {
12     //Creates an instance in the new program version
13     //Creates a new transaction in the past
14     //Invokes the custom convert method
15     //Puts the migrated instance in the VBox pair
16     //Returns the new pair of object/version
17   }
18 }
19
20 class Pair { Object object; int version; }
```

Figure 3.15: Outline of class `Handle`. The code shows how handles use VBoxes to keep an history of updatable instances. Handles also provide the opportunity to intercept the first access to an outdated instance after a DSU takes place and thus migrate it (line 7).

updates. When post-processing the geometry application example, defined in Section 3.3.1, DuSTM renames class `Rectangle` to `Rectangle__0` in version 0 and `Rectangle__1` in version 1. It renames all the other updatable types, just class `Point` in this case, in the same way.

Note that classes C_0 and C_1 are two separate and unrelated Java classes. Class C_1 is free to define a completely different set of fields/methods or to be in a different position in the class hierarchy. The only restriction is that they look the same to the execution platform, the layer that contains all the non-updatable code as explained in Section 3.3.6. The discussion at the end of Section 3.4.2 shall explain this restriction in further detail.

Performing class updates this way allows DuSTM to support DSU without requiring a custom JVM. However, introducing a new Java class per class update creates a new problem: A single object can now be represented by several instances of different classes. For instance, consider that a rectangle refers to a point p. In version 0, the rectangle refers to instance p_0 of class `Point__0`. After the update, in version 1, that same rectangle refers to instance p_1 of class `Point__1`.

Both instances p_0 and p_1 represent the same conceptual point p in different versions of the program. The identity of p is thus kept by a different instance in each version of the program. But, at the time a DSU takes place, the rectangle has a reference to instance p_0. After the update, that reference should be fixed to refer to instance p_1 instead. A possible way to solve this problem is to use a modified garbage-collector that finds all references to p_0 and replaces them by references to p_1. That approach has three problems: (1) It requires a custom JVM; (2) it restricts update semantics to immediate updates, as defined in Section 3.3.3; and (3) it restricts DuSTM to quiescent updates, as defined in Section 3.3.2.

DuSTM solves this problem using *handles*. A handle is an extra level of indirection responsible for keeping the identity of an object across program versions. When transforming the bytecode, DuSTM ensures that all updatable objects are accessed through their handle. Handles, in turn, ensure that the program manipulates the right instance according to the current program version it is executing.

Handles are transactional memory locations implemented using JVSTM VBoxes, as Figure 3.15 shows. An important property of handles is that they keep a *history of instances*, each one corresponding to a version of the object in a particular program version.

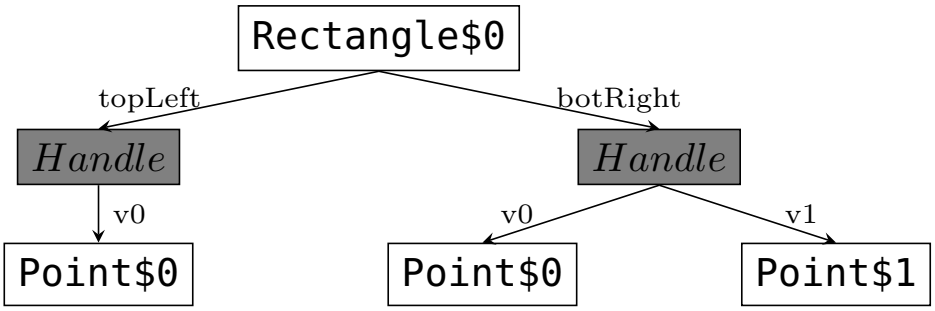


Figure 3.16: Rectangle represented using two points, as described in Section 3.3.1, after post-processing. Note how classes were renamed according to their version (`Rectangle` to `Rectangle$0` and `Point` to `Point$0` and `Point$1`). In this figure, the rectangle has not been migrated yet from version 0 but one of its points, the bottom right vertex, already has. The handle for the bottom right vertex keeps both versions of the point.

Using VBoxes to implement handles is more than a particular implementation decision. VBoxes, which embody JVSTM multi-versioned approach, play a key role to ensure DuSTM's atomic update transactions: When executing a conversion transaction, which takes place in the logical past as explained in Section 3.3.5, handles use their VBox to return the correct (older) instance.

Handles naturally solve the conversion ordering problem that was introduced in Section 3.3.5. Figure 3.16 shows how DuSTM introduces handles to the geometric application example that we are following. Note how the rectangle keeps a reference to handle `botRight`, which in turn has a reference to an instance of class `Point_0` in version 0 and `Point_1` in version 1. This handle is thus responsible for keeping the identity of the point across program versions. When the program accesses this handle during transaction T, the handle checks the current program version in which T is being executed and returns the correct instance.

3.4.2 Supporting Inheritance

The post-processor that DuSTM uses transforms the program to ensure that all instances of updatable types are always manipulated through their respective handle. This means replacing the type of fields, local variables, and method arguments by handles.

We can see the results of the replacement process in Figure 3.17. In this particular example, the figure shows a single handle class that is used for both points and rectangles. This works in this case because both points and rectangles look the same for a class in the execution platform: Given that they both extend `Object` and do not implement any non-updatable interface, both classes look like `Object` to code outside the updatable types.

Using a single handle class to represent all possible updatable types does not generalize. For instance, consider the example that the left-hand side of Figure 3.18 shows. In this case, the handle DuSTM would use for both classes `Square` and `Pixel` has the following Java signature: `class Handle extends Rectangle implements Comparable`. Such a general handle has two problems: (1) Class `Square` does not implement interface `Comparable` but the handle used to represent its instances does; and (2) the general handle would require multiple inheritance to represent any other updatable class that inherits from a non-updatable class different from `Rectangle`. Solving problem 1 would make the post-processing more complex, but problem 2 makes the general handle approach impossible to implement because Java does not support multiple inheritance.

```
1  class Point__1 {
2    private final double rho,theta;
3
4    public Point__1(double rho,double theta) {
5      this.rho = rho; this.theta = theta;
6    }
7
8    public double dist(Handle other) {
9      double raa = square(this.rho), rbb = square( ((Point__1)other.get()) .rho);
10     double rab = this.rho* ((Point__1)other.get()) .rho;
11     double cosab = cos(this.theta- ((Point__1)other.get()) .theta);
12     return sqrt(ra+rb+2*rab*cosab);
13   }
14 }
15
16 class Rectangle__1 {
17   private final Handle topLeft, botRight, botLeft;
18
19   public Rectangle__1(Handle tl,Handle bt,Handle bl) {
20     topleft = tl; botRight = br; botLeft = bl;
21   }
22
23   public double area() {
24     return ((Point__1)topleft.get()) .dist(botLeft)
25         * ((Point__1)(botRight.get()) .dist(botLeft);
26   }
27 }
```

Figure 3.17: Example of the second version of the geometric application example after being post-processed. The original code is shown in Figure 3.4. This example uses a single generic handle class, which is the one defined in Figure 3.15. The code that the post-processing tool introduces or modifies is highlighted with a gray background. The first version of the geometric application, shown in Figure 3.3, can be post-processed in the same way. In this code, the name of method `Handle.getObject` was shortened to `get`.

The solution to support handle inheritance in a way that generalizes to any Java program is to use several specific handles for each branch in the class hierarchy that crosses the boundary between updatable types and the execution platform. The updatable types that we find closest to this boundary are the *root updatable types*, which are the updatable types that inherit directly from non-updatable types. In this case, class `Square` is the only root-updatable type. For each root updatable type, DuSTM generates two classes: A *primary handle* and a *handle helper*. For each remaining updatable type that implements aditional non-updatable interfaces, DuSTM generates a *secondary handle*.

The right-hand side of Figure 3.18 shows how the classes from its left-hand side look after DuSTM post-processes them. Primary handles and handle helpers (classes `Rectangle__Handle` and `__Rectangle`, respectively) override all the methods they inherit from the root updatable type. Secondary handles (class `RectangleComparable__Handle`) implement all the methods defined by the set of non-updatable interfaces they implement.

Downward Methods

All the handle classes that Figure 3.18 shows on the right-hand side override all the methods that they inherit. These are the *downards methods* and their implementation forwards the execution to the same method on the updatable instance that the handle keeps. Figure 3.19 shows several options for implementing handles and downward methods.

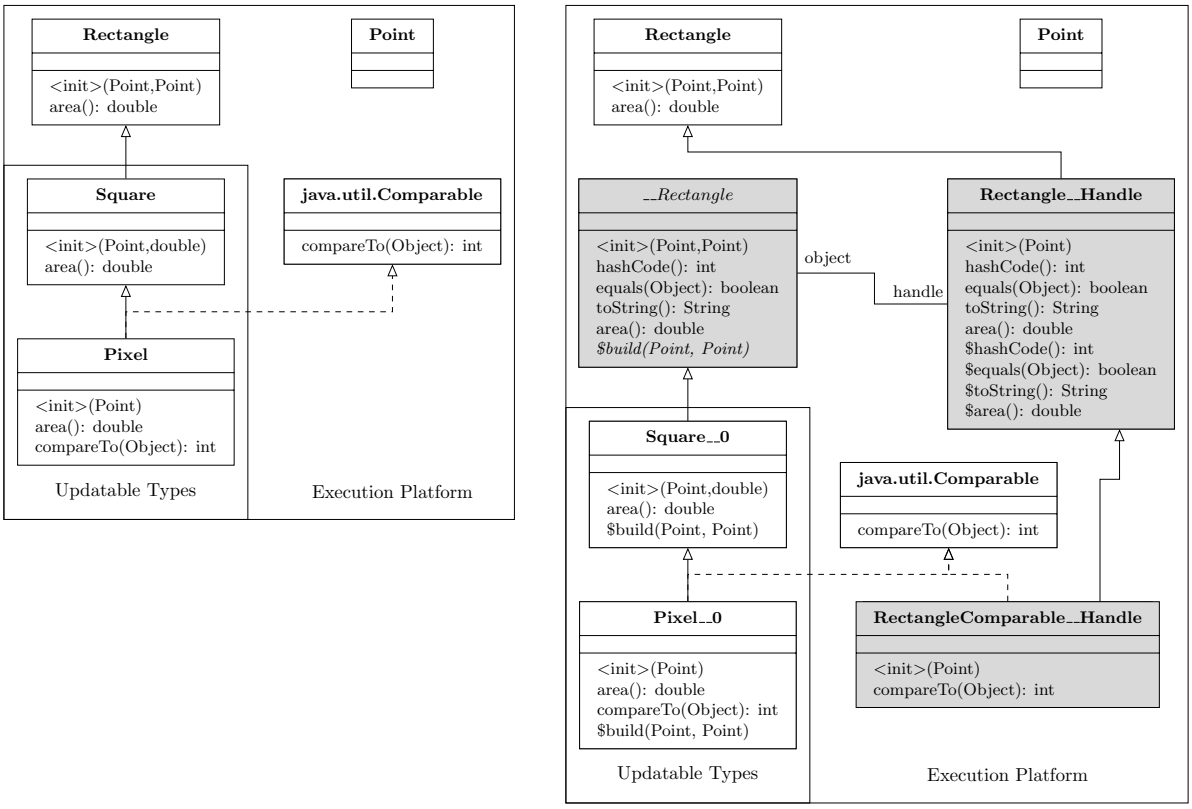


Figure 3.18: Example of a small updatable application before and after DuSTM post-processes it. The updatable application is composed of two classes: `Square` and `Pixel`. Note that classes `Rectangle` and `Point`, in this case and unlike the application introduced in Section 3.3.1, are not updatable. The left-hand side shows how the application looks before post-processing. The right-hand side shows how DuSTM transforms the original code when post-processing and all the handles and helper classes that DuSTM generates, shaded in gray. Constructors are denoted by method `<init>` with no return type. The return type of method `$build` is `Rectangle__Handle`. Method `$build` in class `__Rectangle` is abstract, which makes the class abstract as well.

Note that DuSTM broke the inheritance relationship between updatable and non-updatable types during post-processing. On the right-hand side of Figure 3.18, classes `Square__0` and `Pixel__0` are not subclasses of class `Rectangle`. As a consequence, handles cannot keep the updatable instance typed as a the non-updatable type. Option 1 in Figure 3.19 is thus incorrect.

An alternative approach is to generate handles that keep the object typed with its updatable type, as option 2 in Figure 3.19 shows. However, this creates a strong coupling between the handle and one particular program version. What happens when a new program version defines a new version of class `Square`? Given that handles are non-updatable themselves, this option cannot support any future class updates. Handle helper classes bridge the gap between updatable and non-updatable types by providing the same interface as their respective non-updatable type. Downward methods can use them to delegate on the correct method of the updatable instance. Option 3 of Figure 3.19 shows this approach.

Default and Updward Methods

Helper classes provice a *default method* for each each downward method on the handle class. To understand the need for these methods, let us consider invoking method `toString` on an instance of handle `RectangleComparable__Handle` that keeps an instance of class class `Pixel__0`. The first method invoked is the downward method `RectangleComparable__Handle.toString`, which invokes method `__Rectangle.toString`.

Option 1

```
1 class Rectangle__Handle {
2   Rectangle object;
3
4   double area() {
5     return object.area();
6   }
7 }
```

Option 2

```
1 class Rectangle__Handle {
2   Square__0 object;
3
4   double area() {
5     return object.area();
6   }
7 }
```

Option 3

```
1 class Rectangle__Handle {
2   __Rectangle object;
3
4   double area() {
5     return object.area();
6   }
7 }
```

Figure 3.19: Possible implementations for downward methods. This figure shows 3 possible implementations for downward method `Rectangle__Handle.area()`, shown in Figure 3.16. Only option 3 is correct. Option 1 does not work because `__Rectangle` is not a subclass of `Rectangle` and option 2 creates a strong coupling between the execution platform and the updatable code that prevents updating root updatable types in the future. Option 3 uses handle helpers to implement the downward method without any of these issues.

At this point, the JVM performs a *virtual method invocation* to select the most specific method to run based on the type of the receiver. In this case, it follows the order: (1) `Pixel__0.toString`, (2) `Square__0.toString`, and then (3) `__Rectangle.toString`. Given that methods 1 and 2 do not exist, the JVM invokes method 3 — the default method on the handle helper.

To match the semantics of the original program, default methods have to perform *non-virtual method invocation* to call the original method on the non-updatable instance, `Rectangle.toString` on this case.[6] However, the JVM forbids non-virtual method invocation outside the same inheritance branch. DuSTM solves this problem by injecting an *upward method* to the handle for each downward. Upward methods use non-virtual method invocation to call the original non-updatable method. Default methods can thus simply invoke the respective upward method.

To complete this example, the default method `__Rectangle.toString` invokes upward method `Rectangle__Handle.$toString`[7], which in turn invokes method `Rectangle.toString` thus matching the original program semantics.

Regular Java programs use non-virtual method invocation to call methods on the super-class through the keyword `super` or to invoke constructors. The post-processor adjusts all relevant invocations through `super` to use upward methods instead. I shall discuss how DuSTM handles constructor calls to build updatable instances in the following section.

3.4.3 Post-Processing Method Bodies

The JVM is a stack-based virtual machine [LY99]. Method arguments are pushed into the operand stack in the right order before invoking the method, which consumes all arguments and leaves the return value, if any, on the top of the stack. Arithmetic and logic instructions also take their operands from the operand stack, consuming them in the process and leaving the result on the top of the stack.

DuSTM transforms the updatable code following the rationale: Direct references to instances of updatable types reach the stack only if they are going to be immediately used by the following bytecode instruction. Otherwise, any reference on the stack is either to an handle or to an instance of a non-updatable type.

[6] Performing a virtual method invocation would result in an infinite loop.

[7] Upward methods have the same name as the downward method that originated it, with a `$` character prepended to the original downward method name.

For instance, let us consider the bytecode instruction that reads a field from an instance — `GETFIELD`. This bytecode instruction expects a direct reference to an instance of the right type to be on the top of the operand stack.[8] After post-processing, there will be a handle instead of the expected direct reference. The post-processor thus injects a sequence of instructions to consume the reference to the handle and leave a direct reference that matches the type that the `GETFIELD` instruction expects.

For the sake of readability, throughout the rest of this section I shall use the term *direct reference* meaning a direct reference to an updatable type.

Constructing Updatable Instances

DuSTM transforms the original program so that each updatable instance is always accessed through its handle. Constructing a new updatable instance has three challenges: (1) The correct handle that will represent the identity of the new instance must be constructed at the same time of the instance, (2) no direct references to the newly constructed instance can escape, and (3) all the constructors that would be called in the original program, one per super-class, must be called in the same order. For instance, considering the example introduced by Figure 3.18 that we are following, the challenges are: (1) Constructing an instance of class `Pixel__0` implies constructing an instance of class `RectangleComparable__Handle`; (2) no references to the new instance of class `Pixel__0` can escape, they could later be used to bypass the handle; and (3) constructors should run in the following order to match the semantics of the original program: `Rectangle`, `Square__0`, and `Pixel__0`.

Invoking the constructor of an updatable type on the post-processed program naturally invokes the constructors of all its parent updatable types until this chain of invocation reaches the handle helper class. For instance, on the example that we are following, invoking the constructor of class `Pixel__0` naturally results in invoking the constructor of classes `Square__0` and then `__Rectangle`. However, given that the post-processor broke the hierarchy at this point, the construction does not continue naturally to class `Rectangle`. To solve this problem, and to address challenge 3, the constructor of a handle helper starts by creating the handle, thus resuming the sequence of constructor invocations in the same order as in the original program.

Solving challenge 1 is more complex. The handle helper must create the right type of handle: `Rectangle__Handle` for class `Square__0` and `RectangleComparable__Handle` for class `Pixel__0`. For that purpose, DuSTM injects a method named `$build` to every updatable class that creates the right type of handle. In the example that we are following, method `__Rectangle.$build` is abstract, method `Square__0.$build` creates an instance of `Rectangle_Handle`, and method `Pixel__0.$build` creates an instance of `RectangleComparable__Handle`. The constructor on the helper class invokes method `$build` to create the approriate handle and resuming the sequence of constructors in the correct order.

This approach, however, has a subtle problem: What happens if the constructor of class `Rectangle` invokes method `area`? Assuming that we are building an instance of class `Pixel__0`, the method that should be executed is `Pixel__0.area`, which should be invoked through downward method `Rectangle__Handle.area` as explained earlier. However, for downward methods to work, field `Rectangle__Handle.object` should be set to the updatable instance that the handle keeps. When the constructor of class `Rectangle` calls method `area`, this field is not yet set.

DuSTM solves this problem by exploring the semantics of object construction at the bytecode level [LY99], which is slightly different from the semantics at the Java level [GJS96]. In Java, every constructor should start by invoking a constructor of its superclass.[9] However, at the bytecode level, constructing an object involves two bytecode instructions. First, `NEW` creates an uninitialized object of

[8] The type signature and the name of the field are passed as immediate operands in the bytecode instruction itself.

[9] Except for class `java.lang.Object` which has no superclass.

```
1  // Original
2  class Square extends Rectangle {
3    void accept(Visitor v);
4  }
5
6  class Pixel extends Square {
7    void accept(Visitor v);
8  }
9
10 class Visitor {
11   void visit(Square s);
12   void visit(Pixel p);
13 }
```

```
1  // Post-processed
2  class Square__0 extends __Rectangle {
3    void accept(Object__Handle v);
4  }
5
6  class Pixel__0 extends Square__0 {
7    void accept(Object__Handle v);
8  }
9
10 class Visitor__0 {
11   void visit__Square(Rectangle__Handle s);
12   void visit__Pixel(Rectangle__Handle p);
13 }
```

Figure 3.20: Implementation of visitor pattern to show how DuSTM handles overloaded methods. This code builds on the example introduced in Figure 3.18. It introduces a new type of handle for class `Visitor` that inherits directly from `java.lang.Object`: `Object__Handle`. The implementation of this handle is not important for this example and is thus omitted.

a given class. Second, `INVOKESPECIAL` is used to invoke the correct constructor through non-virtual method invocation. In between these two instructions, the program can perform any arbitrary computation that either does not involve the uninitialized object, or only writes values to fields on the uninitialized object.[10] When generating the body of each `$build` method, the post-processor injects bytecode to set field `object` on the uninitialized handle object before invoking the handle's constructor.

Finally, challenge 2 is the easiest to solve. Constructing an updatable instance leaves a direct reference to that instance on the operand stack. DuSTM injects code immediately after invoking the updatable constructor to replace the direct reference by a reference to its respective handle.

Method Signatures and Overloading

Any method m that belongs to an updatable type can declare updatable types as its arguments. In that case, the body of m expects direct references to instances of updatable types to be passed as arguments when invoking m. To follow the rationale about direct references only being used just before bytecode instructions that require them, the post-processor modifies the signature of method m so that it declares each argument with an updatable type as being the type of the primary handle for that updatable type. For instance, consider the code that Figure 3.20 shows. Note how the post-processor changes the signature of both methods `visit` in class `Visitor` so that they take instances of `Rectangle__Handle` as their argument.

When replacing argument types from updatable types to handles, the fact that several updatable types share the same primary handle can generate name and signature collisions for overloaded methods. For instance, in the example that Figure 3.20 shows, the direct substitution would yield two undistinguishable methods with the same name and signature: `void accept(Rectangle__Handle h)`. Given that method overloading is resolved at compile time, the post-processor scans all classes and renames overloaded methods.[11] In the example that we are following, note how methods named `visit` in the original program get renamed to `visit__Square` and `visit__Pixel` in the post-processed program.

[10] The javac compiler uses this feature to support building instances of inner classes, setting the reference to the instance of the outer class before invoking the constructor of the inner class. Any other operation on uninitialized objects, such as reading fields or invoking methods, throws an exception.

[11] Method overloading can occur by declaring a method with the same name on different classes along the same hierarchy. The post-processor can handle this, and renames methods consistently.

```
1  // Original
2  class C {
3    C(Square s);
4    C(Pixel p);
5  }
6
7
8
9
10
11 new C(new Square());
12 new C(new  Pixel());
```

```
1  // Post-processed
2  class C__0 {
3    C__0(Rectangle__Handle s, Dummy1 d);
4    C__0(Rectangle__Handle p, Dummy2 d);
5  }
6
7  class Dummy1 { /* Empty */ }
8
9  class Dummy2 { /* Empty */ }
10
11 new C(new Square__0().handle, (Dummy1) null);
12 new C(new  Pixel__0().handle, (Dummy2) null);
```

Figure 3.21: Example of how DuSTM handles overloaded constructors. This code builds on the example introduced in Figure 3.18. The extra argument that DuSTM adds to each constructor is used just to differentiate between the signature of the two constructors after replacing the arguments that have updatable types by their handle. DuSTM also changes all sites that invoke these particular constructors so that they pass a null reference as the extra argument, which is then ignored by the body of the constructors.

The post-processor cannot rename constructors, even though they can be overloaded and suffer from the same problem as regular methods. In this case, the post-processor adds a dummy argument of a synthetic type to distinguish them. For instance, consider class `C` in Figure 3.21. It declares two constructors, one that takes a `Square` and another that takes a `Pixel`. The post-processor generates two synthetic classes `Dummy1` and `Dummy2` and adds an extra argument to each constructor with a different synthetic type. The two constructors are now distinguishable. The post-processor also updates all sites that call the modified constructors to pass a `null` reference as the extra argument.

Receiver Reference `this`

There are two cases in which a direct reference can reach the stack: (1) On non-static methods, the receiver reference `this` is a direct reference; and (2) when creating an instance of an updatable type, the constructor leaves a direct reference on the stack. This section has already explained how DuSTM deals with case 2 (page 80). To eliminate direct references escaping through case 1, the post-processor adds a bytecode preamble to every non-static method that an updatable type defines. The preamble is equivalent to the Java code `this = this.handle`. The `this` reference, in bytecode, translates to a regular local variable with number zero. Therefore, even though the Java equivalent is not correct Java code because the receiver reference `this` cannot be a left-value, the bytecode preamble is correct and accepted by a standard JVM.

Replacing Handles with Direct References

When the post-processor finds a bytecode instruction that requires a direct reference, it can safely assume that a reference to its handle is on the stack position where the instruction expects a direct reference. It must thus inject a sequence of bytecode instructions to produce a direct reference from the handle and place it on the right position on the operand stack.

We can group the bytecode instructions that require direct references in two groups: (1) Field manipulation instructions, and (2) method invocation instructions. Figure 3.22 shows how the stack looks like just before executing each of these instructions. Field manipulation instructions expect the direct reference to be within the top 2 positions of the stack. When the handle is on the top of the operand stack — left-hand stack on Figure 3.22 — the post-processor injects a single instruction to call method

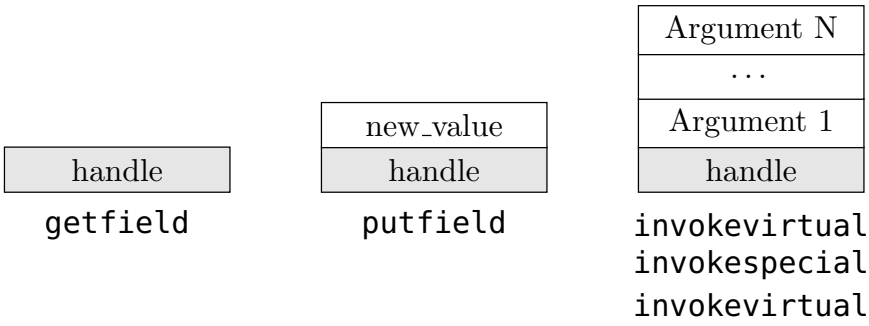


Figure 3.22: Location of handles in the operand stack just before each type of bytecode instruction. These bytecode instructions require direct references to the instance that the handle keeps. The stack grows from bottom to top. Each stack is annotated with the instructions in the original program that expect the shaded handle to be a direct reference.

`getObject` which consumes the handle reference and leaves a direct reference on the stack. When the handle is one position deeper — center stack on Figure 3.22 — the post-processor injects a sequence of instructions that swap the top 2 positions, get a direct reference from the handle in the same way, and swap the top 2 positions of the stack back to their original configuration.

Dealing with method invocation instructions is more complex because instructions that invoke methods expect the method receiver to be a direct reference and to be arbitrarily deep on the stack — right-hand stack on Figure 3.22 — under all the arguments for the method. For the general case, the post-processor uses *trampolines*, which are static methods that take the handle for the receiver as the first argument and perform the original invocation using the rest of the arguments after getting a direct reference from the handle. The post-processor optimizes for invocations of methods with less than two arguments by using the same approach that it uses for field instructions, described earlier in this section.

Implementing trampolines has a small technical challenge: The post-processor needs to generate several trampolines per each method, one per invocation type that the JVM supports [LY99]. Therefore, for each method that an updatable class defines, the post-processor generates two trampolines: One for virtual invocations and another for non-virtual invocations. For each updatable interface, it generates a trampoline that performs interface invocation for each method that the interface declares. Given that interfaces in Java cannot have concrete methods, the post-processor places these trampolines in special helper classes.

3.4.4 Object Identity Semantics

To ensure handles are transparent and retain the semantics of the original program, DuSTM has to keep the original semantics of object identity. In Java, the `==` operator compares the identity of two objects. Given that the post-processor replaces all references to proxied objects by references to their handles, the `==` operator ends up being applied to handles which, naturally and correctly, considers that two objects that have the same handle as having the same identity.

The `==` operator is not the only way that a Java program can compare the identity of two objects. Each class defines two methods that are related with the identity of an object: (1) Method `equals` determines if two objects are equal, and (2) method `hashCode` returns an integer hash-code for the object. These methods have some restrictions, related with the identity of the objects they are invoked on: Method `equals` must consider as equal two objects that the `==` determines to have the same identity; method `hashCode` must return the same value for two objects that method `equals` considers to be equal. The transformed program invokes both these methods on the handle, which delegates them to the instance of the updatable type that it keeps. Handles thus retain the identity semantics of the objects they keep.

```
1  // Original
2  Square s = new Square();
3  Rectangle r = s;
4  ... r instanceof Rectangle ...
5  ... s instanceof Rectangle ...
6  ... s instanceof Square ...
```

```
1  // Post-processed
2  Handle s = new Square().handle;
3  Rectangle r = s;
4  ... r instanceof Rectangle ...
5  ... s instanceof Rectangle ...
6  ... s.object instanceof Square ...
```

Figure 3.23: Transformation of `instanceof` operator. After the program transformation, variable `s` refers to a handle. The post-processor injects bytecode to get the reference to the instance that the handle keeps prior to executing an `instanceof` operator on updatable types. The original classes in this example are shown in Figure 3.18. Class `Handle` is short for class `Rectangle__Handle`.

Updatable Code

```
1  // Original
2  void updatableMethod(Point p) {
3    Pixel px =
4      new Pixel(p);
5    nonUpdatableMethod(px);
6  }
7
8  // Post-Processed
9  void updatableMethod(Point p) {
10   RectangleComparable__Handle px =
11     new Pixel__0(p).handle;
12   nonUpdatableMethod(px);
13 }
```

Non-updatable Code

```
1  List lst;
2
3  void nonUpdatableMethod(Rectangle r) {
4    lst.add(r);
5  }
6
7  void anotherNonUpdMethod(Rectangle r) {
8    lst.sort();
9    System.out.println(lst.get(0));
10 }
```

Figure 3.24: Example showing how updatable instances can be safely passed to non-updatable code. This code uses the types introduced in the example shown in Figure 3.16. Note that the non-updatable code does not get a direct reference to the updatable instance of class `Pixel__0`, it gets instead a reference to its handle.

The `instanceof` operator in the Java language checks if the object referred to by its operand is an instance of a particular Java class. After the post-processor transforms the program, the occurrences of this operator in the updatable code would end up receiving handles as their operator, instead of a direct reference to an updatable type. For instance, consider the example shown in Figure 3.23. After post-processing, variable `s` refers to a handle. The expression on line 6 on the original program would return false and thus violate the semantics of the original program. To solve this problem, the post-processor injects code to get the instance that the handle keeps just before each time the operator `instanceof` is used to compare a reference against an updatable type. We can see an example of the transformation on the right-hand side of Figure 3.23.

3.4.5 Limitations

The separation between the updatable types and the execution platform, introduced in Section 3.3.1, means that updatable types can be instantiated only inside updatable code. The program transformations described in this section ensure that the updatable code cannot get a direct reference, it can only manipulate updatable instances through their handle. Therefore, the updatable code can only pass updatable instances through their handle to the execution platform.

For instance, consider the example shown in Figure 3.24. Note how the post-processor injects code to replace the direct reference to object `px` by its handle immediately after construction, in line 11 of the left-hand side. Line 12 on the left-hand side shows how the updatable code can only pass updatable instances through their handle to the non-updatable code.

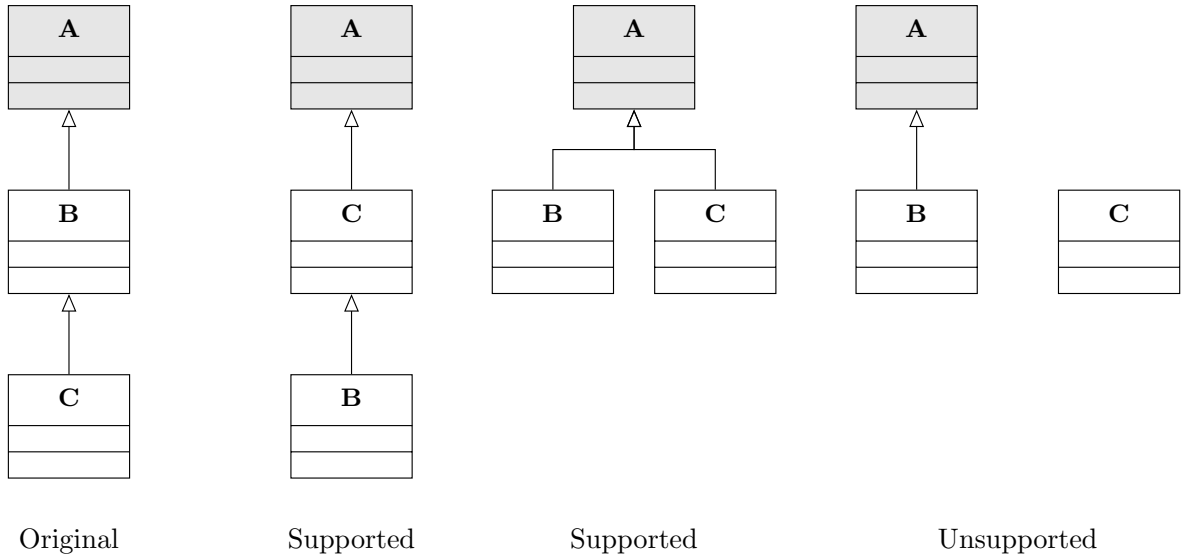


Figure 3.25: Supported and unsupported program evolutions. This figure shows 3 possible program evolutions over an original simple program composed by 3 classes: `A`, which is non-updatable; `B` and `C`, which are updatable. DuSTM can perform the first two evolutions as a DSU but not the last because it breaks the inheritance between a non-updatable type (`A`) and an updatable type (`C`).

Allowing the execution platform to interact with updatable types only through handles in this way has two shortcomings. First, the execution platform cannot instantiate any updatable type. Some Java frameworks are configured through reflection, by specifying an instance of class `java.lang.Class` as a parameter for the framework to later create instances of client code. DuSTM does not support updatable code that uses such frameworks. Reflection allows updatable instances to be created outside of post-processed code, which opens the possibility of creating direct references to updatable instances.

Second, all class updates of updatable types must have the same *non-updatable signature* as the first version, i.e., they must have the same non-updatable parent and implement the same set of non-updatable interfaces. The reason for this restriction is the fact that handles are not updatable by design and they may be kept inside instances of non-updatable types. For instance, consider the non-updatable code shown on the right-hand side of Figure 3.24. The code on the left-hand side passes a pixel through its handle to the non-updatable code, that keeps it as a rectangle. If a later DSU changes the pixel to not inherit from rectangle anymore, the non-updatable code that has references to the handle will break.

Figure 3.25 shows examples of supported and unsupported changes to a class hiearchy. The left-hand side shows the initial hierarchy. Class A is non-updatable, classes B and C updatable. Furthermore, class A is a superclass of B, which is a superclass of C. In this example, DuSTM supports any DSU in which classes B and C are subclasses of A, such as the two examples in the center of Figure 3.25. DuSTM, however, does not support performing a DSU that changes the class hierarchy as the example on the right-hand side of Figure 3.25 shows because it changes the non-updatable signature of class C, which no longer inherits from class A.

Reflection

Preserving the original program semantics when performing bytecode transformation on programs that use reflection is notoriously hard, as other authors acknowledge [TS02, GR09]. A possible approach is to create a custom reflection library that mimics the behaviour of regular Java reflection but hides handles and other program transformations (such as class name mangling). The bytecode transformation tool can then rewrite all reflection calls to calls to equivalent methods on the custom reflection library.

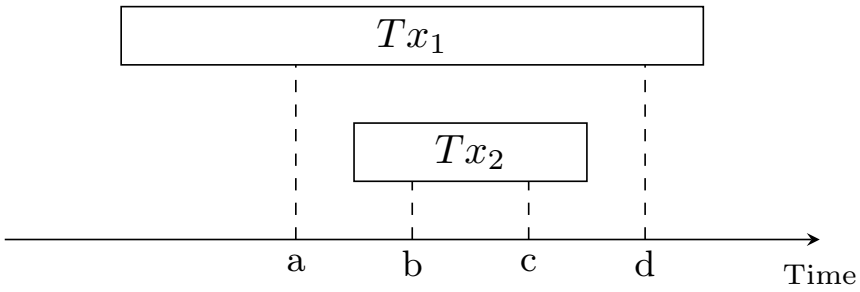


Figure 3.26: Optimization for reading the current program version. Transaction Tx_1 reads the same VBox at instants a and d. Transaction Tx_2 writes to that VBox at instant c and commits at instant d. Note that Tx_1 always reads the same value both times. Knowing that transaction Tx_1 will read, and only read, that VBox creates the opportunity to save overhead by reading the value when Tx_1 starts and caching it locally for subsequent reads. This is how DuSTM optimizes reading the VBox that keeps the current program version.

I used this approach when implementing DuSTM. Ensuring that this approach supports the full reflection API is a matter of engineering effort. DuSTM's prototype currently provides minimal reflection support, but enough to execute real-world applications as Section 3.5 shall describe. In fact, the choice of which reflection calls to support was guided by the efforts of supporting the execution of those real-world applications.

This approach to deal with reflection still poses two problems. First, to achieve full transparency, DuSTM would also have to post-process reflection invocations on methods that belong to the execution platform. Second, reflection calls made inside native code are hard to intercept and DuSTM currently does not support it.

3.4.6 Optimizations

DuSTM transforms the original code to ensure that instances of updatable classes are always manipulated through their respective handle. Every time DuSTM manipulates any updatable instances through its handle (e.g. when executing a downward method), it uses the `getObject` method shown in Figure 3.15 on page 75. This method is called frequently and thus represents an important source of performance overhead.

A closer look at this method reveals that it reads two VBoxes[12]: One to get the object-version pair (line 6) and other to get the current program version (guard of the `if` statement in line 7). Reading VBoxes is an expensive operation[13] because it involves several tasks: (1) Checking whether the VBox was previously written in the context of the current transaction, (2) ensuring that the read-set of the current transaction contains the VBox being read, and (3) iterating through the versions that the VBox keeps to find the most recent version that the transaction can read.

Task 3 ensures isolation between transactions, as Figure 3.26 shows. Transaction Tx_1 reads a VBox at instants a and d. Transaction Tx_2 writes to that VBox at instant b and then commits at instant c. For this execution to be consitent with opacity, JVSTM has to ensure that transaction Tx_1 reads the same value on instants a and d. When the transaction reads the VBox at instant d, task 3 skips over the value written by transaction Tx_2 and returns the value previously read at instant a.

There is an interesting observation in the execution that Figure 3.26 shows: Reading the same VBox v multiple times inside the same transaction T that does not write to v always results in the same value, which is the most recent value that v has when transaction T starts. In the general case, it is hard

[12] When not migrating the instance that the handle keeps, which is the common case
[13] Details can be found in Appendix A, Section A.4 (page 167).

to know in advance which VBoxes will only be read during a transaction.[14] For programs that use DuSTM, however, every transaction will only read the VBox that keeps the program version. DuSTM thus starts every transaction by reading the VBox that keeps the program version and then saves that value in a thread-local variable[15], effectively reducing the cost of reading the current program version when executing method `getObject`.

As for the other VBox that method `getObject` reads, the one that keeps the updatable object and its program version in a pair, it is possible to optimize its access by skipping tasks 1 and 2. Task 1 is not necessary because handles are written only during conversion transactions. Task 2 is also redundant because the `if` statement on line 7 performs the same validation that adding the VBox to the read-set would trigger at commit time. DuSTM can thus safely skip these tasks.

These two optimizations significantly lower the performance overhead that DuSTM imposes on steady-state execution.

3.5 Experimental Evaluation

To evaluate DuSTM, I developed a prototype that follows the implementation that Section 3.4 describes. This section describes the experimental evaluation that I performed to evaluate that prototype. Section 3.5.1 describes a series of experiments that evaluate the prototype in the context of updating an application that already uses memory transactions as its concurrency control mechanism and, in particular, a JVSTM backend as the provider of memory transactions. Section 3.5.2 evaluates how the prototype can be applied to existing real-world applications that do not use memory transactions.

3.5.1 Updating an STM-Based Application

This section describes an experimental evaluation of the DuSTM's prototype using STMBench7 [GKV07]. STMBench7 is is a synthetic benchmark for evaluating STM implementations that simulates the workload of a real-world Computer-Aided Design (CAD) application. The benchmark builds several sets of object graphs that resemble how a CAD application structures its program state. The workload consists of a series of operations, executed concurrently, that traverse and/or modify those graphs following different patterns that resemble the different ways that a CAD application mutates its state. The program state has invariants that each operation should preserve and that allow the benchmark to confirm the correctness of the underlying STM implementation.

STMBench7 is highly configurable. Among several other configuration options not interesting for this discussion, STMBench7 can be configured with the type of the workload, the number of threads to use, and the duration of the benchmark run. This section reports results obtained by configuring STMBench7 with a JVSTM backend and selecting the read-write workload with 1 and 4 threads.

All the instances that STMBench7 keeps can be reached through the transitive closure of a small number of root objects accessible via static fields. This property enables DuSTM to simulate immediate-update DSU by suspending all transactions when performing a DSU and then traversing all instances that STMBench7 keeps. After it finishes traversing (and migrating) the program-state, DuSTM resumes all transactions that it previously suspended.

The workload that STMBench7 uses was further configured with two types of operations disabled: (1) long traversals and (2) structural modifications. Long traversals access a large portion of the total available program-state, which cause the JVSTM backend to build large read-sets. In fact, these large read-sets put so much pressure on the garbage-collection mechanism that it renders any experimental

[14] Except for read-only transactions, which only read VBoxes. JVSTM optimizes for the case of read-only transactions, but not for the case that Figure 3.26 depicts.

[15] In JVSTM, there is a one-to-one correspondence between threads and transactions.

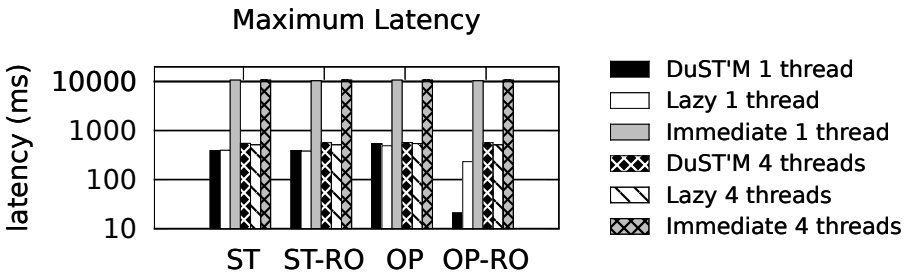


Figure 3.27: Maximum latency for each different type of operation in STMBench7 when performing lazy and immediate DSU using DuSTM. *DuSTM* refers to a benchmark execution in which there no DSU is performed. Each bar cluster represents data from a different type of operation: *ST* stands for Short Traversal, *OP* for Operations, and *RO* for Read-Only.

result unusable. Structural modifications perform random modifications (add a leaf node, delete a leaf node, delete a whole branch) at random points in the graphs that STMBench7 keeps. At some point during the benchmark execution, these operations delete a small set of branches that render a large portion of the whole program-state unreachable, which then results in trivial update times when comparing immediate versus lazy updates.

As explained in Section 3.3.6, DuSTM separates an updatable application into the updatable code and the execution platform. The updatable code, in this case, is composed by the classes that implement the JVSTM backend used for STMBench7. Semantically, this means that it would not be possible to update the behavior of STMBench7's operations, but it would be possible to replace the current JVSTM backend by a more efficient one, for instance replacing a set currently implemented using a list by a more efficient one that is implemented using a tree.

To evaluate updating STMBench7, I used a synthetic *v0v0* update which does not introduce any modification but considers that all updatable types were modified. A v0v0 update is a conservative approximation of the worst-case scenario for DSU because it migrates the complete program-state by making a deep copy of it from the old to the new program version.

The machine used to run the experiments that the remainder of this section presents is, unless noted otherwise, a system equipped with an Intel Core i5 750 processor (4 cores) and 8GB RAM, running a 64-bit Linux 2.6.36, and Java SE version 1.6.0_24 (Java HotSpot 64-Bit Server VM, build 19.1-b02, mixed mode).

Maximum Latency

STMBench7 measures the latency of each operation, measured as the time interval between issuing an operation and its completion. STMBench7 aggregates the maximum latency that it measured by type of operation. Figure 3.27 reports the maximum latency observed per operation over 15 benchmark runs, each lasting for 180 seconds and performing a DSU at 90 seconds.

As expected, immediate updates noticeably increase the maximum latency of each operation. In this case, immediate updates increase the maximum latency by a factor of almost 2 orders of magnitude over the base case. Lazy updates, on the other hand, add a modest increase on the maximum latency for each operation.

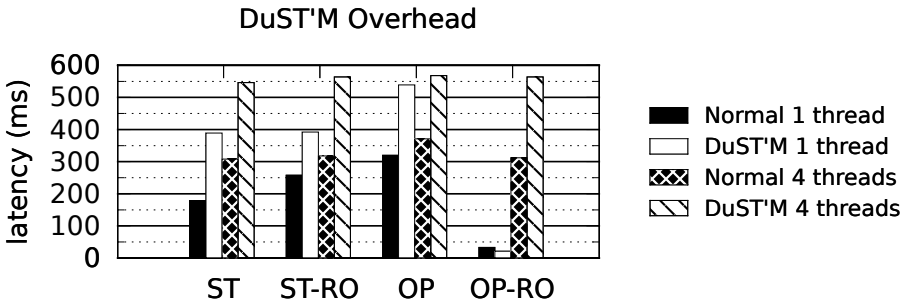


Figure 3.28: Increase of maximum latency introduced by DuSTM. *Normal* refers to the benchmark without being post-processed, *DuST'M* refers to the benchmark after being post-processed. *ST* stands for Short Traversal, *OP* for Operation, and *RO* for Read-Only.

Steady State Overhead

DuSTM introduces an extra level of indirection on the original application in the form of handles. I performed an experiment designed to measure how this extra indirection affects the maximum latency on steady state. The experiment consists of running two versions of STMBench7 in which one is left unmodified and another is post-processed using DuSTM and executed without performing any DSU during the benchmark run.

Figure 3.28 shows how the program transformation that DuSTM performs through the post-processor increases the maximum latency for every operation. The difference in performance shows the performance cost that DuSTM introduces during steady-state execution to pay for the ability to perform DSU in the future.

Throughput

STMBench7 measures the overall throughput of a benchmark run, i.e., how many operations were completed on average per second. With a simple customization, I adapted STMBench7 to measure how many operations were completed during each second of the benchmark run. The resulting measurements provide a clearer look into how the throughput varies over time.

Figure 3.29 shows the results of an experiment that measures the number of operations STMBench7 completes per second during a 180 seconds benchmark run that performs a DSU at 90 seconds. Each line reports the aggregate results of 15 benchmark runs. The noise in the original results would make all lines in Figure 3.29 unreadable.[16] Simply averaging the data from the 15 executions does not smooth the lines enough for them to be easily read. To generate this figure, each second shows instead the average of a sliding window of 5 seconds (the current and the past 4 seconds). This technique makes the plotting readable at the cost of loss of precision on a per-second basis. In particular, the time required to perform an update (while the throughput drops to zero in the plot) is inaccurate.

After a lazy update, transactions that execute the new program are interleaved with conversion transactions that migrate instances. Immediate updates, on the other hand, just execute new program transactions without any interruption. This explains why the performance after lazy updates takes some time to reach the plateau after the update. The throughput drop for 1 thread is deeper than for 4 threads. This happens because the probability of all 4 threads performing conversion transactions simultaneously after the update is lower.

[16] For instance, garbage-collection cycles influence the shape of each line at random points during the execution of the benchmark.

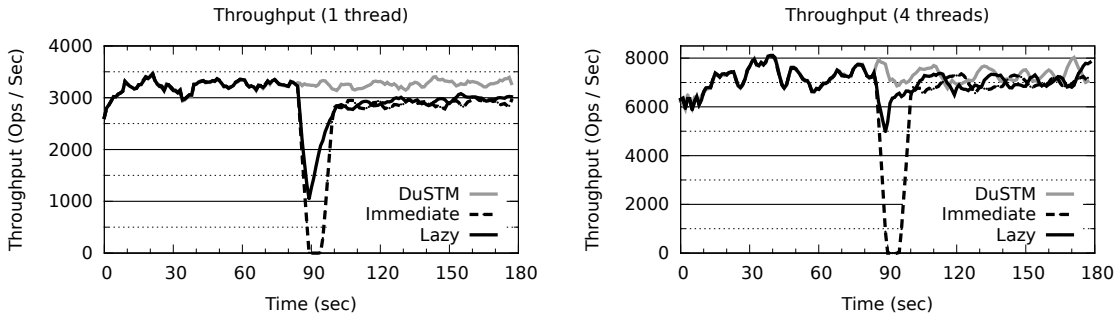


Figure 3.29: Throughput tracing at each second during a 180 second STMBench7 run. Line *DuSTM* represents a benchmark that does not perform any update, lines *immediate* and *lazy* perform an update at 90 seconds using immediate and lazy updates, respectively. Each line shows the aggregate results of 15 benchmark runs. To aggregate the results, each second reports the average of a sliding window of the previous 5 seconds.

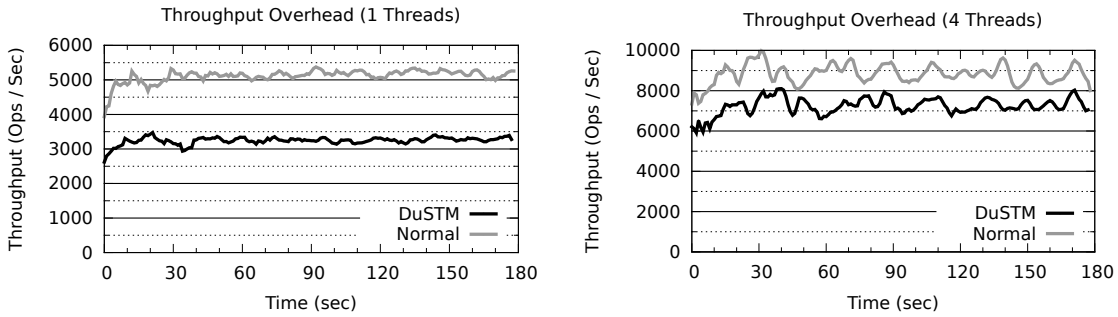


Figure 3.30: Throughput overhead during a 180 second STMBench7 run. Line *Normal* represents the unmodified benchmark and line *DuSTM* represents the benchmark post-processed by DuSTM without performing any update. Each line shows the aggregate measurements from 15 benchmark executions using a similar aggregation technique to the one used for Figure 3.29.

Immediate updates reach post-update performance in less time than lazy updates. These results show that immediate updates are more appropriate for applications that keep a small program size and that must resume maximum performance after the update as soon as possible (e.g. FTP servers, SSH servers).

The results show that, when converting the state immediately, there is a time period following the update during which the throughput is zero. This null-throughput window lasts for 10 seconds on average, which agrees with the 10-second increase of the maximum latency that Figure 3.27 shows for immediate state conversion. Updates performed using lazy state migration still result in a sharp performance drop that reaches zero but then quickly recovers. The period during which the performance is zero is, on average, less than one second.

This experiment can be adapted to measure the throughput overhead that DuSTM adds to steady-state execution in the absence of updates. Figure 3.30 shows how the throughput varies over time for two scenarios, one running STMBench7 unmodified and another running STMBench7 post-processed with DuSTM but without performing any DSU. The difference between the two lines shows the throughput cost that DuSTM introduces to steady-state execution to pay for the ability to perform DSU in the future.

The results show that DuSTM introduces, on average, 18% overhead for 4 threads and 37% for 1 thread. The cost of synchronizing threads masks the overhead for the scenario with a higher number of threads.

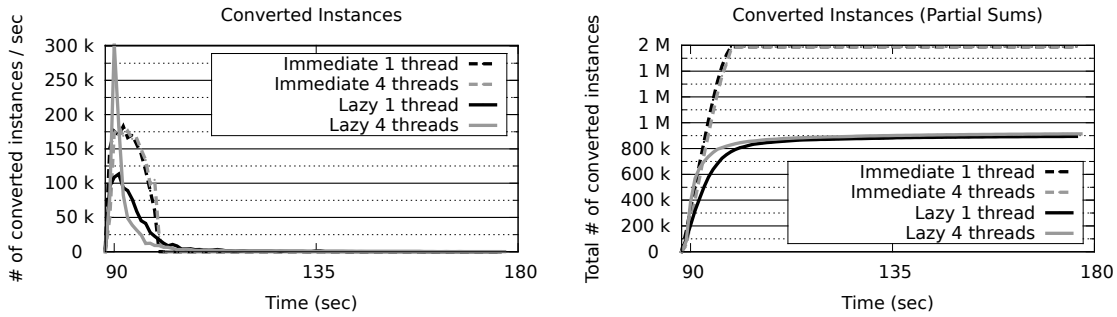


Figure 3.31: Number of instances DuSTM transforms during an update. The left-hand side plot shows how many instances DuSTM migrates at each second. The right-hand side plot shows how many instances DuSTM migrated in total up to each second. These plots were generated using the average results for 15 benchmark runs of 180 seconds each that perform a DSU at 90 seconds. The initial 90 seconds of execution do not migrate any instance because they take place before the update and are therefore omitted.

Migrated Instances

The results that this document presented so far show that lazy updates require a shorter pause in the execution of the application to perform a DSU. This empirical evidence meets the expectation that lazy updates require less time to resume execution following an update because they need to migrate fewer instances. This section describes an experiment designed to compare the number of instances migrated by each type of update, and presents its results.

To measure how many instances were migrated after an update, I modified DuSTM to count the number of instances that it migrates at each second. Figure 3.31 shows the number of instances migrated per second after performing a DSU. The results are the average of 15 benchmark runs, each lasting for 180 seconds and installing an update at 90 seconds. The left-hand side shows the number of instances migrated at each second, the right-hand side shows the total number of instances migrated up to each second.

DuSTM uses a single thread to traverse, and migrate, the program-state when performing immediate updates. This explains the similarity between the lines for the immediate updates on the single-threaded and multi-threaded versions of the benchmark: When migrating the program-state, both use a single thread to traverse it.

After a lazy update, the program executes new code in transactions that are interleaved with conversion transactions that migrate the program-state as it is accessed for the first time. This is why the lazy single-threaded update has a lower peak on the number of migrated instances per second than the immediate updates. This also explains why the multi-threaded lazy updates have the highest peak: Even though each thread has to interleave program execution with program-state migration, the combined effect of the 4 threads migrates more instances than a single thread dedicated to program-state migration.

The right-hand side shows that lazy updates migrate fewer instances, in total, than immediate updates. Moreover, both lazy update configurations reach a similar plateau of approximately $900,000$ migrated instances. This observation supports the hypothesis that lazy updates naturally identify the working-set of the application, migrate it at a high rate following an update, and migrate the rest of the program state at a much lower rate while executing the application.

I conducted another experiment to measure how the number of migrated instances and the rate of migration varies with the size of the program-state. I modified STMBench7 so that it builds a larger program-state by multiplying the total number of a common instance type in the benchmark by a factor of 5, 10, and 20. Figure 3.32 shows the total number of instances migrated up to each second after the

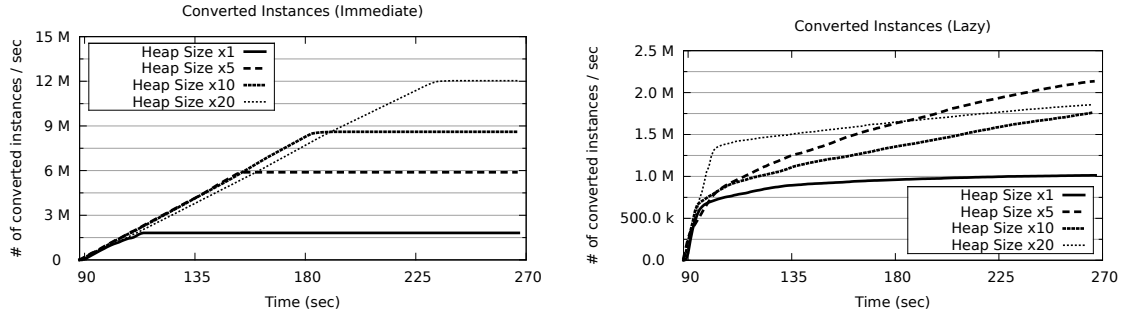


Figure 3.32: Number of instances migrated after performing an update with a varying program-state size. Each plot shows the total number of instances migrated up to each second after the update. The left-hand size plot shows the measured numbers for the immediate program-state migration and the right-hand side for the lazy program-state migration. Please note the difference between the scales on both charts, the lazy update migrates much less instances in total than the immediate update. These plots were generated using the average results for 10 benchmark runs of 270 seconds each that perform a DSU at 90 seconds. The initial 90 seconds of execution do not migrate any instance because they take place before the update and are therefore omitted.

update. Each line in the plotting is the average of 10 benchmark executions. Each benchmark execution runs for a total of 270 seconds and installs an update at 90 seconds. Due to its increased memory requirements, this experiment was conducted on a different machine, equipped with 4 AMD Opteron 6168 chips (12 cores per chip, 48 cores total) and 128GB of memory, running a 64-bit Linux 2.6.32, and Java SE version 1.6.0_22 (Java HotSpot 64-Bit Server VM, build 17.1-b03).

The total number of instances that immediate updates migrate is also the total number of instances that make up the entire program-state of an application. We can see that number in Figure 3.32 by looking at the point in which the line of each immediate update stops increasing. From this observation, we can confirm the hypothesis that the total number of instances that lazy updates migrate is much lower than the total number of instances that in the program-state. Lazy updates are thus more appropriate to perform DSU on programs that keep a large program-state.

Maximum Latency of Constant Operations

Some operations that STMBench7 executes traverse a percentage of all the existing objects in the program-state. By increasing the total size of the program-state we are also increasing the amount of work that these operations perform and, therefore, their latency.

However, there is a class of operations — *short traversals* — that traverse a fixed number of objects, independently of the total size of the program-state. To measure how the maximum latency of these operations varies with the program size, I adapted the experiment performed to measure the total number of migrated instances, which results are shown in Figure 3.32, to instead measure the maximum latency of short traversals. Figure 3.33 shows the maximum latency observed on 10 benchmark executions, each one lasting 270 seconds and installing an update at 90 seconds. The experiment was repeated for a total heap size multiplied by a factor of 5, 10, and 20.

The results confirm our expectations by showing that immediate updates increase the maximum latency of the fixed-work operations by a factor that is proportional to the total size of the program-state. On the other hand, there is no visible relation between the latency of fixed-work operations and the total size of the program-state when performing lazy updates. The results from this experiment also confirm that lazy updates are more appropriate to perform DSU when the latency of operations that perform a fixed amount of work should be minimized or otherwise kept below a certain value (e.g. multimedia streaming).

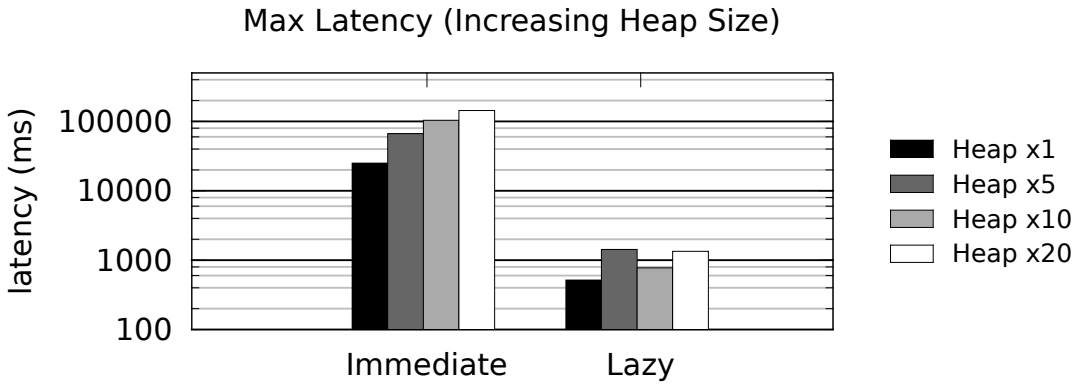


Figure 3.33: Maximum latency measured for STMBench7 operations that perform a fixed amount of work. The behavior of these operations is not dependent on the total size of the program-state. For this experiment, I executed each scenario for 270 seconds and installed an update at 90 seconds. The results show the maximum latency observed for 10 executions per scenario.

```
1  class Handle {
2    Object object;
3
4    static Object getObject(Handle h) {
5      return h.object;
6    }
7  }
```

Figure 3.34: Outline of non-transactional class `Handle`. Unlike the outline shown in Figure 3.15, on page 75, the handle does not use a transactional VBox to keep the reference to the instance of the updatable type.

3.5.2 Cost of the Handles

Section 3.4 explains how DuSTM provides support for DSU through injecting an extra level of indirection in the original program. Section 3.5.1 evaluates the cost of using DuSTM on transactional applications. This section evaluates the cost of using DuSTM on non-transactional applications, in particular, it evaluates the penalty, in terms of memory usage and performance, that the extra level of indirection adds to non-transactional real-world[17] Java applications:

To evaluate the cost that the extra level of indirection adds to existing non-transactional programs, I used a version of DuSTM's post-processor modified to generate handles that keep the reference to the instance of the updatable type using a regular Java field instead of a JVSTM VBox. This is the equivalent of generating handle types following the outline that Figure 3.34 shows, while still performing all the program transformations that Sections 3.4.2, 3.4.3, and 3.4.4 describe. This approach separates the overhead that JVSTM would introduce from the overhead that the handle injection technique introduces.

I used the modified post-processor to introduce handles to eight Java applications:

Avrora An AVR microchip simulator;

Batik A SVG renderer;

Daytrader An online stock trading benchmark;

FOP A tool that generates PS and PDF files from XSL-FO files;

H2 A SQL DBMS implemented in Java;

[17] In the sense that they are not small synthetic programs like STMBench7 is.

Benchmark	Package	# of Classes	Manually Modified	
			Files	Lines
avrora	`avrora`	1545	2	47
batik	`org.apache.batik`	2547	0	0
tradebeans	`org.apache.geronimo.samples.daytrader`	37	0	0
tradesoap				
fop	`org.apache.fop`	1314	5	12
H2	`org.h2`	471	0	0
luindex	`org.apache.lucene`	542	3	3
lusearch				
PMD	`net.sourceforge.pmd`	720	0	0
sunflow	`org.sunflow`	221	0	0

Table 3.1: Details about how DuSTM post-processed each DaCapo benchmark. Column *Package* lists the top-level package considered updatable. Column *# of Classes* lists the number of classes that DuSTM post-processed for each application. Some applications required manual changes to avoid unsupported reflection calls. The two columns under *Manually Modified* list the files and lines that were modified for each application.

Lucene A text-search engine;

PMD A Java bug finder;

Sunflow A ray-tracer.

The DaCapo benchmark suite [BGH+06] provides benchmarks with a representative workload for each of these applications. Most applications have a single benchmark and some applications, *Daytrader* and *Lucene*, have two different benchmarks. For *Daytrader*, one benchmark uses regular Java beans while other performs the operations through a SOAP layer. For *Lucene*, one benchmark indexes an existing text while another searches for keywords on an existing index.

All these applications generate deterministic output that allows the DaCapo benchmark suite to detect semantic errors by comparing the results of each benchmark with the expected result. All the benchmarks generated the expected results, which provides empirical evidence that the post-processor does not change the semantics of the original program.

Experimental Setup

Table 3.1 shows the details about how each application was post-processed. Column *package* specifies the name of the package that I considered to have all the updatable types.[18] The following column lists the total number of classes considered updatable for each benchmark.

Some applications used reflection calls that are not supported, as Section 3.4.5 explains. In those cases, I manually modified the small parts of each application that relied on reflection and that the benchmark needed to run. The two columns under *Manually Modified* on table 3.1 show how many files and lines were modified to support reflection calls.

To assess the overhead that handles introduce to the original application, I executed the benchmark suite on a quadcore system, with an Intel Core i5 750 processor (4 cores) and 12GB RAM, running a 64-bit Linux kernel version 3.2.0. For each benchmark, I ran the original application and a modified version in which I post-processed all classes considered updatable (located under the packages shown in Figure 3.1) using DuSTM. I used JVM version 1.6.0_33 (HotSpot build 20.8-b03, mixed mode), configured to use a max heap size[19] of 10GB.

[18] All the types found on the sub-packages are also considered updatable.
[19] Using the `-Xmx` option when launching the JVM

Benchmark	TTC (ms)		Heap (MB)		GC (#)	
	O	P/O	O	P/O	O	P
avrora	35032.63	1.12	8.79	1.25	36.11	28.63
batik	3431.95	1.04	42.08	1.08	1.95	2.00
tradebeans	29919.89	1.07	477.80	0.72	4.95	10.58
tradesoap	28147.42	0.99	443.98	0.99	12.11	12.21
fop	439.63	1.50	6.87	2.13	2.26	2.21
H2	45308.84	1.50	214.18	1.06	4.32	6.21
luindex	1004.37	1.30	6.82	1.36	2.16	2.16
lusearch	1992.16	1.56	133.16	0.63	20.42	27.11
PMD	4685.21	1.43	125.75	1.35	2.47	2.42
sunflow	9467.68	1.51	29.39	1.87	59.79	46.58

Table 3.2: Overhead introduced by DuSTM handles. *TTC* stands for time to complete, *GC* for garbage collection runs, *O* for original, *P* for proxified, and *P*/*O* for overhead.

I executed each benchmark using the DaCapo suite configured to perform 20 iterations for each benchmark. Therefore, every value that this section reports corresponds to the average of those 20 executions. Furthermore, I ran two different sets of 20 executions for each benchmark: One to measure the runtime overhead and another to measure the memory overhead. I monitored the memory usage by sending a QUIT signal to the JVM periodically (every second), which generates a heap summary (total and used size per generation). Obviously, this technique alone introduces non-negligible runtime overhead, which is the reason why I used a different set of executions for measuring the memory overhead.

All the benchmarks are part of the DaCapo benchmark suite version 9.12-bach. I ran all benchmarks using DaCapo's *large* workload size configuration except for *FOP* and *luindex*, which I ran with the *default* workload size because that is the largest workload that the DaCapo benchmark suite has available for these benchmarks.

Performance Overhead

Table 3.2 shows the overhead, in terms of performance and memory, that the post-processing adds. I executed each program using the DaCapo benchmark suite, which already reports the time-to-complete for each benchmark. To measure the performance overhead, I compared the time-to-complete a benchmark run using the original program with the time-to-complete using the post-processed version.

The performance overhead depends on the application and clusters around two groups: Either it is negligible (*batik*, *tradebeans*, and *tradesoap*) or significative (*fop*, *H2*, *lusearch*, *PMD*, and *sunflow*). *Avrora* and *luindex* are the exceptions that present an acceptable non-negligible overhead.

The benchmarks *tradebeans* and *tradesoap* are part of the Daytrader application and they represent a well-known and popular type of Java applications: Web applications. The overall performance of web applications depends on the persistence layer, that keeps the application data; and on the application server, that processes client requests using the web application and sends the replies back. The performance sensitive kernel of code is therefore located outside the updatable types and in the execution platform.

These results suggest that DuSTM can be applied to web applications without imposing a noticeable performance overhead. They also suggest that performance-sensitive kernels should be kept in the execution platform to retain performance (at the cost of not being updatable anymore). The challenge here is to identify and isolate those application kernels without violating the separation between updatable types and execution platform that DuSTM requires, as explained in Section 3.3.6.

Memory Overhead

Table 3.2 shows the overhead, in terms of memory usage, that the post-processing introduces. Note that the absolute value of used memory varies from benchmark to benchmark. The memory overhead reaches its highest values for benchmarks with the lowest absolute heap usage (*avrora*, *fop*, *lunidex*, and *sunflow*). This happens because the JVM minimizes garbage collection cycles to improve performance, thus allowing more garbage to accumulate for benchmarks that use less memory. This effect artificially bloats the memory usage measurements for benchmarks with low memory usage.

The memory usage measured for *tradebeans* and *lusearch* is lower for the post-processed version than for the original version. These results are inconsistent with the fact that the post-processed version uses more objects to represent the same program-state. The number of garbage collections cycles explains the measured value: The post-processed version triggers more garbage-collection cycles that free more memory than the original version. This suggests that the rate at which the post-processed versions use memory is higher than the original versions, which is expected.

The post-processed code has the potential to double the amount of objects that an application uses and thus increase the required memory by a comparable factor. The results show, however, that such a worst-case scenario does not happen in practice. In particular, the memory overhead is acceptable for benchmarks that keep larger heaps such as *tradebeans*, *tradesoap*, *H2*, *lusearch*, and *PMD*.

3.6 Discussion

This chapter presented DuSTM, a system that allows to perform DSU on a transactional Java application. Section 1.2 defines a set of goals that a practical DSU system should reach, and Section 3.1 presents a set of claims about DuSTM reaches some of those goals. This section explains how this chapter supports those claims.

Flexibility DuSTM is a flexible DSU system. It supports a wide range of modifications between updates, including any type of structural modification of updatable classes and, with some restrictions, changing the shape of the class hierarchy. DuSTM also supports custom program state migration between successive program versions, and provides tools, such as the *update class* and *old classes*, that allow the developer to describe the program state migration using regular Java code.

Before Rubah, the DSU system presented in the following chapter, DuSTM was the most flexible DSU system to the best of my knowledge. DuSTM thus fully achieves the flexibility goal.

Correctness DuSTM provides clear update semantics that can be used to reason about its correctness. Given that it supports DSU for transactional Java applications, it provides a modular framework for developers to reason about the correctness of updates in terms of transactions that take place in either the old or new program versions. *Update transactions* install the new code and migrate the program state and *migration transactions* transform each object that needs to be migrated between versions. DuSTM provides clear semantics for each of these transactions, and ensures that regular application transactions always execute in the same program version. Furthermore, DuSTM does not require the developer to change the original program so that it can support future updates.

These are good and important properties in the sense of correctness. However, DuSTM still requires the developer to write code that executes in-between program versions (to transform the program state) and does not provide any tools to ensure that an updated program will behave as expected. DuSTM could be adapted to use Tedsuto, a framework for testing updates, and thus reach the goal of correctness. However, as it stands, DuSTM does not fully achieve the goal of correctness.

Efficiency DuSTM transforms the program state lazily when it performs an update. This amortizes the large pause required to transform the whole program state over the execution of the new program version, resuming execution after an update as soon as possible.

To avoid long update-induced pauses, DuSTM introduces an extra level of indirection in the original application. This increases memory usage and adds steady-state overhead. As a result, DuSTM does not fully achieve the goal of correctness.

Effectiveness DuSTM targets a popular language — Java — and is implemented as a runtime library and a post-processor that rewrites the updatable code so that it supports DSU without requiring a custom compiler or JVM. While developing the updatable application, programmers do not need to use the post-processor and can thus use the same development time tools they would otherwise use.

DuSTM, however, can only be used with transaction Java applications. This requirement is a hard restriction on the possible programs that can benefit from DuSTM, and it limits the effectiveness of DuSTM. Therefore, DuSTM does not fully achieve the goal of effectiveness.

DuSTM is an important step towards all the goals that I defined for a practical DSU system. The fact that it requires the updatable program to be written for a transactional memory model is its most limiting factor. Unfortunately, there are not a large number of applications written for TMs. Should applications for TMs become widely available, DuSTM would be a ready-to-deploy solution for practical DSU.

Transactional memory enriches the programming model to make reasoning about concurrency modular and easier. In the process, it achieves the same goals for DSU. In the following chapter, I explore the consequences of relaxing the requirement for a transactional memory programming model, together with the rewards in terms of effectiveness and efficiency.

Chapter 4

Efficient Real-World Updates

The applications that most benefit from DSU are those that have high availability requirements. These applications typically provide service to a large number of clients that expect fast request processing and low latency between successive requests. Important on-line services are written in managed languages such as Java. For example, Twitter has moved most of its major infrastructure to Java,[1] and the Java-based Voldemort noSQL database is used by large companies such as LinkedIn.

In the previous chapter, I described a mechanism for performing DSU on Java programs that are structured around transactions. The major advantages that transactions bring to DSU are: (1) Correct update points are easy to find, and (2) the program state can be migrated lazily. For 1, the transactional system just needs to ensure that each transaction executes always on the same program version in which it started. That is, only transactions that start after the DSU execute on the new program version. For 2, the system can keep the illusion that all the program state was migrated at the time of the DSU but actually only migrate each outdated object just before it is needed after the update.

Unfortunately, the vast majority of existing applications with high-availability requirements is not structured around transactions. However, we can adapt the ideas presented in the previous chapter by considering that each program version executes in one coarse-grained transaction, instead of requiring the program to be structured around fine-grained transactions. This way, updates still take place in their own transaction that, as before, installs the new program code and migrates the program state to an equivalent state that is compatible with the new program code. The process of performing a DSU now involves committing the current coarse-grained transaction, executing the update transaction, and starting the new coarse-grained transaction for the new version. This insight allows us to apply the benefits of a transactional approach to programs that are not transactional.

In this chapter, I describe a mechanism that uses such an approach — *Rubah* — to support DSU on regular Java programs. Rubah enjoys all the advantages of DuSTM without requiring the updatable program to use transactions at all. Rubah, however, requires the developer to change the original application to support DSU in the following ways: Identify *update points*, which are program points where it is safe to perform a DSU, and add *control-flow migration* code to unroll the stack at update time and then rebuild it after the update finishes, so that the program resumes executing from the same point in which it was when the update took place.

[1] http://www.gmarwaha.com/blog/2011/04/11/twitter-moves-from-rails-to-java/

This chapter is structured as follows: Section 4.1 provides an overview of the contributions of Rubah with regards to the set of goals introduced in Section 1.2. Then, Section 4.2 explains how to modify existing applications to support DSU through Rubah, explaining the update semantics in detail. Section 4.3 presents the algorithms that Rubah uses to transform the program state between successive program versions. Section 4.4 describes a prototype implementation of Rubah, and Section 4.5 describes an experimental evaluation using that prototype and five real-world applications. Finally, Section 4.6 discusses Rubah's contributions.

4.1 Claims

This section presents claims about how Rubah reaches the goals defined in Section 1.2.1, providing an overview of the rest of this chapter guided by those goals and enumerating Rubah's main contributions.

Flexibility. Rubah is extremely flexible. It was able to handle release-level updates for five applications, described in Section 4.5.1. In fact, to the best of my knowledge, no prior system can handle the same range of updates Rubah can.

The updatable application model that Rubah supports is similar to DuSTM in the sense that Rubah also considers the updatable application to be composed of *updatable classes* and *non-updatable classes*. Rubah, however, supports a wider range of program changes than DuSTM to updatable classes. Rubah supports all the program changes that DuSTM does, and it lifts the requirement that updatable classes have to keep the same non-updatable superclass between updates. Section 4.4.1 discusses the types of updates that Rubah supports in further detail.

I claim that Rubah fully achieves the goal of flexibility as defined in Section 1.2.1 — ● following the notation introduced by Table 2.1.

Effectiveness. Rubah targets programs written in the Java programming language. It works by performing a semantics-preserving bytecode rewriting that enhances the original program with support for DSU. Rubah does not require a custom JVM to support DSU. Rubah, therefore, allows the developer to use the same development-time tools as he would otherwise use.

All these ideas were previously introduced and explored by DuSTM. Rubah's approach is, however, more effective because it rewrites the bytecode as classes are loaded by the running program. This approach eliminates the need for a post-processing stage after compilation and enables Rubah to modify non-updatable code as well as updatable code. Section 4.2.1 explains the workflow required to run an updatable version of an application and install an update at a future time.

Rubah requires the developer to modify the original programs to support DSU. In particular, developers have to add update points and control-flow migration to the updatable program. Section 4.2.2 presents an example of an updatable application and Sections 4.2.3, 4.2.4, and 4.2.5 explain in detail the changes that developers have to perform to make any program updatable, using the example to motivate each change.

Rubah places some restrictions on the control structure of updatable programs. In particular, Rubah expects updatable programs to be written around long running loops that: Take a request; process it; and return the result to the entity that performed the request, eventually keeping some state about that entity between requests. This control-structure is very common among high-available server software and Section 4.5.2 measures the effort to add support for Rubah to the five real-world server applications listed in Section 4.5.1.

I claim that Rubah fully achieves the effectiveness goal as defined in Section 1.2.4 — ● following the notation introduced by Table 2.1.

Efficiency Rubah enjoys good steady-state performance. Unlike DuSTM, Rubah imposes negligible steady-state performance overhead (-1.0–2.5%), as measured by a comprehensive experimental evaluation described in Section 4.5.4.

When performing a DSU, Rubah provides two novel program state migration algorithms: A *parallel* algorithm, that migrates the program state as fast as possible using several threads; and a *lazy* algorithm, that migrates each object as the new program version requires it for the first time after the update. Sections 4.3.2 and 4.3.3 present the parallel and lazy algorithms in detail, respectively. Section 4.4 discusses how Rubah implements each algorithm.

Both algorithms were experimentally evaluated. The results show that the parallel algorithm reduces the time required to perform an update, in Section 4.5.5. The results also show that the lazy algorithm imposes a smaller and nearly-constant update pause that does not depend on the total size of the program state, by amortizing the time required to transform the heap over the execution of the new program version, in Section 4.5.6.

I claim that Rubah fully achieves the efficiency goal as defined in Section 1.2.3 — ● following the notation introduced by Table 2.1.

Correctness. Rubah requires developers to change their applications to add support for DSU. Failure to add an update point means that the program may stop executing, not making any progress until an update is installed and not being able to install any update. An error on the control-flow migration means that the program might lose program state or even crash after an update. An error on the program-state migration can introduce a silent semantic error that might crash the program sometime after the update.

My experience, when adding support to Rubah to the existing applications listed in Section 4.5.1, is that any introduced error typically crashes the program immediately after an update. This happens even for small tests that the developer can run locally before performing a DSU on the program running in a production environment. However, when using Rubah, the developer does not have any other tools besides small trial-and-error tests to ensure the correctness of updates.

DuSTM relies on an underlying transactional memory to isolate program versions and to ensure that transactions that start after an update always execute the new program code and can only see the migrated program version. The semantics of updates are thus composable with application transactions. Rubah, however, does not rely on memory transactions. Instead, each program thread executes each program version in a conceptual coarse-grained transaction that finishes when the thread reaches an update point. After pausing all threads at update points and migrating the program state, Rubah restarts each thread in a new conceptual coarse-grained transaction to execute the new program code with the migrated program state.

Unfortunately, Rubah is further away from the goal of correctness than DuSTM. DuSTM requires a richer programming model than Rubah, and takes advantage of it to support modular and composable updates, exploring the knowledge about transactions to automatically find correct update points. Rubah, on the other hand, places fewer restrictions on the programming model but requires the developer to identify those manually.

I claim that Rubah does not fully achieve the correctness goal as defined in Section 1.2.2 — ◓ following the notation introduced by Table 2.1.

4.2 Dynamic Software Updates with Rubah

This section explains how developers can use Rubah to add support for DSU to their applications. It starts by describing the workflow for using Rubah. Then, it introduces an example, adapted from the H2 SQL database management system. Using the example, this section then explains how the developer

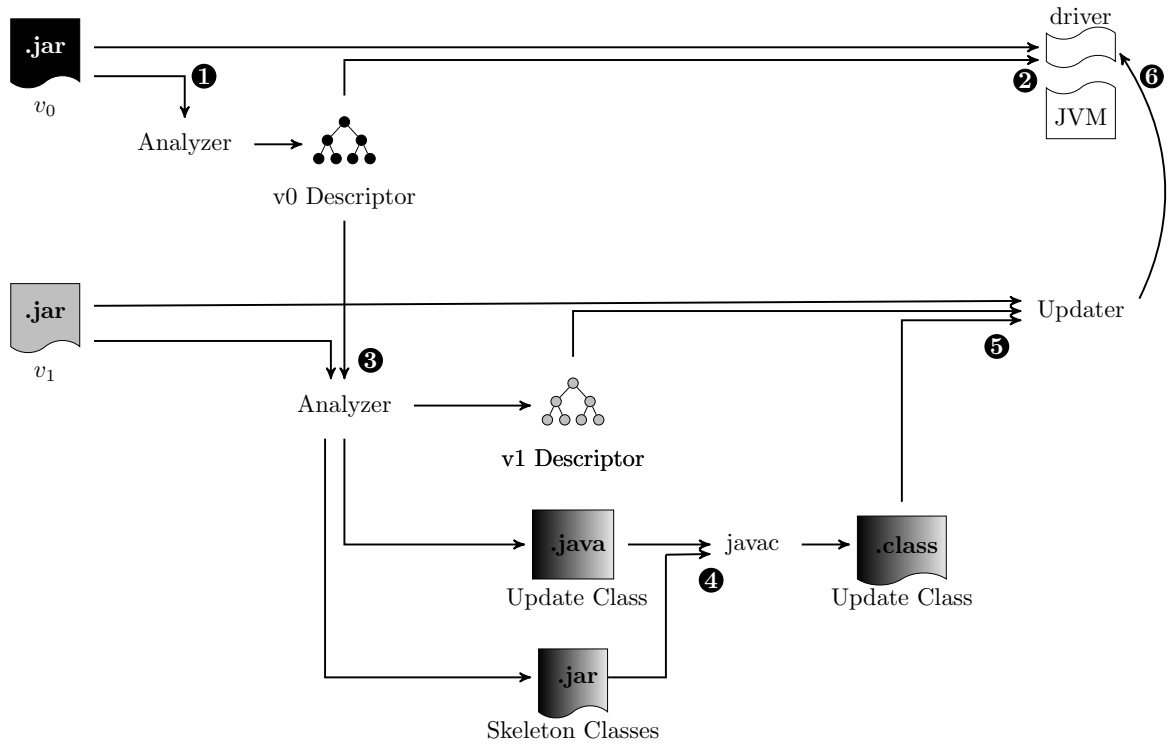


Figure 4.1: Workflow for deploying a program and preparing an update for it using Rubah. After compiling the first version of the updatable application (v_0), the developer uses the Rubah analyzer tool to generate a version descriptor (step **1**). The developer can then run the first version using Rubah driver tool (step **2**). When the developer writes the next version of the application (v_1), he passes it to the Rubah analyzer tool, together with the descriptor of the previous version (step **3**). The Rubah analyzer tool generates a version descriptor for the new version, the source code for a stub update class, and the bytecode for skeleton classes. The developer customizes the update class with the state migration logic and compiles it using the regular Java compiler and the skeleton classes (step **4**). At that point, the developer can use the Rubah updater tool with the new version bytecode, the descriptor of the new program version, and the compiled update class (step **5**), which in turn connects to the Rubah driver tool to start the update process (step **6**).

retrofits it, i.e., adapts its code to support future DSU using Rubah, by adding update-points and control-flow migration. Finally, it explains how the developer transforms the program state between versions when Rubah performs a DSU.

4.2.1 Workflow

The workflow for using Rubah is given in Figure 4.1. Prior to deploying the initial version of a program ("version 0" or v_0), that version's bytecode is given to the Rubah *analyzer* tool, which produces a *version descriptor* that contains meta-data, such as the list of all updatable classes, for that version (step **1**). The program is executed by Rubah's *driver*, which takes the application's classes and the descriptor (step **2**). The driver uses a custom classloader that intercepts each class that the application loads and performs a semantics-preserving bytecode transformation that adds support for future updates to the loaded class, most notably in support of state transformation. The state transformation algorithms and the semantics-preserving bytecode transformation are described in detail in Sections 4.3 and 4.4.3, respectively.

Once a new version of the program is available ("version 1" or v_1), the developer prepares a dynamic update by passing the new code and the v_0 descriptor to the analyzer (step **3**), which produces, along with the v_1 descriptor, an *update class* that describes how existing objects should be transformed to work

with the new code. The programmer can customize this class as needed, and then compile it using the analyzer-produced *skeleton classes* as a placeholder for the old-version classes (step **4**). The dynamic update is deployed by the *updater* (step **5**), which signals the running driver (step **6**), providing the new code and the update class. The driver then deploys the update in three stages. In the first stage, *quiescence*, the driver gets each thread to a point at which it is safe to perform the update. In the second stage, *state transformation*, the driver initiates (and may complete) the transformation of object instances whose class changed (according to the update class). In the final stage, *control-flow migration*, each thread is restarted and shepherded to a point equivalent to the one at which the update took place. At this point, the update is logically complete. Future versions repeat the bottom half of the figure (steps **3**–**6**).

This approach is extremely flexible. Rubah permits changing any class in an arbitrary manner, with few exceptions, whereas past approaches often limit which classes can be changed, and in what ways. For Rubah, the only classes that cannot be updated are the Java runtime classes and libraries (e.g., Java collections). *Updatable classes* can directly reference non-updatable classes but not the reverse, due to issues involving the bootstrap class path of a Java application [LB98]. Of course, library classes do not directly reference application classes, so this restriction poses no practical difficulty.

Rubah requires the programmer to write (or retrofit) the program so that the update process works properly. In particular, to achieve quiescence, the programmer must insert *update points* that identify safe moments to perform updates. The programmer must also add code to perform *control-flow migration*. Finally, for each new version that comes out, the programmer may also need to customize the default update class. The remainder of this section describes what must be done, using an example.

4.2.2 Updatable Application Example

Rubah's design for whole-program DSU is inspired by Kitsune's approach for C [HSD+12].[2] In particular, Rubah is well-suited for complex server applications.

The smallest full example that shows all of Rubah's features would be too large to be easily presented here. Instead, for the sake of simplicity, I present an example adapted from part of H2, a SQL database management system that I modified to support updating through Rubah. I omit all the code that is less interesting. This example will be used throughout the rest of this section.

Figure 4.2a shows the code that listens for connecting clients. The code starts by creating a socket to accept incoming connections (line 7) and initializing the database, e.g. opening the files persisted to disk (line 8). The server then enters a long running loop that handles connecting clients (lines 11–16). During each iteration, the code accepts the new connection (line 12) and creates a new thread to handle the connected client (lines 13–15) using the code shown on the right-hand side, Figure 4.2b. The server catches and logs any exception that might happen (lines 17–21). A `finally` block ensures the socket gets closed when the server exits (lines 21–23). Finally, the server stops the database, e.g. saving files back to disk and flushing its I/O buffers (line 25);

Figure 4.2b shows the code that handles each connected client. This code shows the method `run` of class `TcpServerThread` created on line 13 of Figure 4.2a. The server starts by initializing the connection and negotiating the protocol parameters with the client, e.g. version of the protocol (lines 32–37). The server then processes each command that the client issues on a long-running loop (lines 39–45) that calls method `process` to handle the each command that the client issues (line 41). Method `process` reads the next operation that the client issues, blocking until the client either issues an operation or disconnects (line 55). The code then switches on the operation (lines 56–64). The example shows the outline for

[2]The word *kitsune* means "fox" in Japanese; the word *rubah* has the same meaning in Indonesian, which the language spoken on the island of Java.

```
1  boolean stop = false;
2  ServerSocket serverSocket;
3
4  void listen() {
5
6    serverSocket =
7      NetUtils.createServerSocket(port);
8    initManagementDb();
9
10   try {
11     while (!stop) {
12       Socket s = serverSocket.accept();
13       ServerThread c = new ServerThread(s);
14       Thread thread = new Thread(c);
15       thread.start();
16     }
17   } catch (Exception e) {
18     if (!stop) {
19       TraceSystem.traceThrowable(e);
20     }
21   } finally {
22     NetUtils.close(serverSocket);
23   }
24
25   stopManagementDb();
26 }
```

(a) Code for listener thread.

```
27 Transfer transfer;
28 Socket socket;
29
30 void run() {
31   try {
32     transfer = new Transfer(socket);
33     transfer.init();
34     trace("Connect");
35     // Negotiate protocol with client
36     transfer.flush();
37     trace("Connected");
38
39     while (!stop) {
40       try {
41         process();
42       } catch (Throwable e) {
43         sendError(e);
44       }
45     }
46     trace("Disconnect");
47   } catch (Throwable e) {
48     server.traceError(e);
49   } finally {
50     transfer.close();
51   }
52 }
53
54 void process() {
55   int operation = Transfer.readOperation();
56   switch (operation) {
57     case EXECUTE_QUERY:
58       // Read query parameters from client
59       // Process query
60       // Send result set back to client
61       break;
62     case ...:
63       ...
64   }
65 }
```

(b) Code for server thread.

Figure 4.2: Small server example adapted from H2. The left-hand side shows class `TcpServer`, which listens for new clients and creates a server thread to handle each client that connects. The right-hand side shows class `TcpServerThread`, which handles each client.

the `EXECUTE_QUERY` operation case (lines 57–61): It reads the remainder of the query parameters from the client, using the `transfer` object to so do (line 58); then internally processes the query (line 59); and finally sends the response, a result-set, back to the requesting client (line 60). The code for other operations is similar and omitted from this example (lines 62–63).

This example also shows the complex handling of exceptions, typical in server code. All exceptions raised while processing each command are initially caught in the `catch` block in line 42 and passed to method `sendError` in line 43. Method `sendError` decides how to handle exceptions: It sends recoverable exceptions back to the client and re-raises unrecoverable exceptions, which break the loop, are re-caught in the `catch` block on lines 47, and logged on line 48. A finally block in lines 49–51 ensures that the connection always gets closed when method `run` exits.

Adding support for DSU through Rubah requires the developer to change the original program. Figure 4.3 shows how this example looks after introducing all the necessary modifications. The following Sections 4.2.3, 4.2.4, and 4.2.5 explain each modification in detail. I must emphasize that this example is tiny, therefore the changes required are disproportionally high. Section 4.5.2 provides empirical evidence that the changes required are much smaller than the whole size of the updatable application, for the five real-world applications listed in Section 4.5.1.

4.2.3 Quiescence and Update Points

Rubah performs dynamic updates by safely stopping the program in the old version, migrating the program state, and then restarting the program in the new version. The meaning of *safely stopping* depends on the particular semantics of each updatable program, so Rubah requires the developer to mark program locations where it is safe to stop for an update as *update points*.

A good approach is to place update points where the program is *quiescent*.[3] In the example that we are following, the listener thread, which code is shown on Figure 4.2a, becomes quiescent just before line 12. At this point, it has accepted the previous client and started a thread to handle it and it has not accepted the next one (or blocked waiting for it). The server thread, which code is shown on Figure 4.2b, becomes quiescent just before line 41. At this point, it has finished processing the last client command and has not started to process the next one. Note that both quiescence points keep minimal in-flight state, which simplifies reasoning about the update.

Developers mark update points by placing a call to method `Rubah.update`. This method takes a string as its sole argument, which is a label to identify logically different update points.[4] Figure 4.3 shows two update points, one per each thread, on lines 15 and 59.

Calling method `Rubah.update` when an update is available results in it throwing an `UpdatePointException`. This exception must ultimately reach a Rubah-provided wrapper for a thread's `run` (or `main`) method, where it is caught and dealt with. The thread wrapper is implemented in the class `RubahThread`, which is a drop-in replacement for class `java.lang.Thread` that applications must use. Line 24 on Figure 4.3a shows how an example of using class `RubahThread`.

Of course, the exception may be caught by intervening `catch` blocks in the application, so the developer may need to make changes to avoid this (lines 27–28, 63–64, and 70–71). The developer also needs to ensure that the exception does not change any state by being propagated, therefore actions within `finally` blocks must be guarded to account for possible updates (lines 35 and 76).

When an update becomes available, the program may be blocked waiting for some I/O operation. To avoid an undue delay to the update, Rubah requires the program to either: (1) Use non-blocking sockets and `select` operations, which are blocking but can be interrupted without closing the socket [HSHF12]; or (2) have each thread voluntarily wake-up from I/O calls frequently and reach an update point before blocking again. Rubah provides an API that simplifies retrofitting a program to use non-blocking I/O if needed. In the example that we are following, the listener thread may be blocked in line 12 in the original code in Figure 4.2 when an update becomes available. The example in Figure 4.3 shows how to modify the original code to use Rubah's API for nonblocking I/O operations. The programmer has to use Java's nonblocking *socket channels* (line 2), create a selector object (line 13),[5] use Rubah's API to make the blocking call (line 19), and treat a potential `UpdateRequestedException`, raised if an update becomes available while blocked (lines 20– 22). The server thread, originally shown in Figure 4.2b, can be modified in the same way, as Figure 4.3b shows.

[3] Note that our definition of quiescence differs from (and is not comparable to) that of some prior work [MR07], which defines it to mean that all updated functions are inactive, i.e., not running.

[4] Section 4.2.4 discusses the meaning of this argument in further detail when presenting control-flow migration.

[5] The programmer has to manage the lifetime of the selector object by releasing it when updating (line 34).

```
1  boolean stop = false;
2  ServerSocketChannel serverSocketChan;
3
4  void listen() {
5
6    if (!Rubah.isUpdating()) {
7      serverSocketChan =
8        NetUtils.createServerSocketChannel(port);
9      initManagementDb();
10   }
11
12   try {
13     Selector sel = Selector.open();
14     while (!stop) {
15       Rubah.update("listen");
16       SocketChannel s;
17       try {
18         s =
19         Rubah.accept(serverSocketChan, sel);
20       } catch (UpdateRequestedException e) {
21         continue;
22       }
23       ServerThread c = new ServerThread(s);
24       RubahThread thread = new RubahThread(c);
25       thread.start();
26     }
27   } catch (UpdatePointException e) {
28     throw e;
29   } catch (Exception e) {
30     if (!stop) {
31       TraceSystem.traceThrowable(e);
32     }
33   } finally {
34     sel.close();
35     if (!Rubah.isUpdateRequested())
36       NetUtils.close(serverSocketChannel);
37   }
38
39   stopManagementDb();
40 }
```

(a) Code for listener thread.

```
41 Transfer transfer;
42 SocketChannel socketChan;
43
44 void run() {
45   try {
46     if (!Rubah.isUpdating()) {
47       transfer = new Transfer( socketChan );
48       transfer.init();
49       trace("Connect");
50       // Negotiate protocol with client
51       transfer.flush();
52       trace("Connected");
53     }
54
55     Selector sel = Selector.open();
56
57     while (!stop) {
58       try {
59         Rubah.update("process");
60         process( sel );
61       } catch (UpdateRequestedException e) {
62         continue;
63       } catch (UpdatePointException e) {
64         throw e;
65       } catch (Throwable e) {
66         sendError(e);
67       }
68     }
69     trace("Disconnect");
70   } catch (UpdatePointException e) {
71     throw e;
72   } catch (Throwable e) {
73     server.traceError(e);
74   } finally {
75     sel.close();
76     if (!Rubah.isUpdateRequested())
77       transfer.close();
78   }
79 }
80
81 void process( Selector sel ) {
82   int operation = Transfer.readOperation( sel );
83   ...
84   }
85 }
```

(b) Code for server thread.

Figure 4.3: Changes required to retrofit the example shown in Figure 4.2 to use Rubah. The modified code is highlighted: Update-points have a black background (lines 15 and 59), control-flow migration has a light-grey background (lines 6–10, 24, 27–28, 35, 46–53, 61–64, 70–71, and 76), and I/O has a dark-grey background (lines 2, 13, 16–22, 34, 42, 47, 55, 60, 75, 81, and 82).

When all threads have been stopped at update points, the program is quiescent, and the update may take place. This happens in two steps: *state transformation*, which loads new and updated classes and transforms existing objects to use those new classes, and *control-flow migration*, which returns the threads to their logically correct positions in the (new) application code. I defer discussion of state transformation to Section 4.2.5 and discuss control-flow migration next, completing the explanation of the code in the example.

4.2.4 Control-flow Migration

The goal of the control-flow migration is to return each program thread to an update point in the new version that is equivalent to the point at which it stopped in the previous version. Rubah begins control-flow migration by re-starting each thread's (possibly updated) `run` method (or the `main` method for the main thread, if it is still alive). Each thread eventually reaches an update point with the same label as the update point at which the thread quiesced originally. At that point, Rubah blocks that thread. Once all threads have so blocked, control-flow migration is complete, and all threads may continue. Besides application threads and the main thread, Rubah also supports control-flow migration of thread pools.

When a thread starts for the first time, it typically performs initialization actions that should not be re-performed during control-flow migration. In the example that we are following, the original server code initializes the connections and negotiates the protocol to be used when it starts (lines 32–37 in Figure 4.2b). Executing these lines post-update would result in overwriting state that Rubah migrated and deviating from the protocol that the client is expecting, possibly resulting in the client disconnecting.

To avoid initialization code during control-flow migration, Rubah provides API calls that the developer can use to determine whether a thread is running for the first time or as a result of an update. In the example that we are following, line 46 guards the initialization code on Figure 4.3b with a call to method `Rubah.isUpdating` which returns `true` if called while performing the control-flow migration and `false` otherwise.

We can see another instance of control-flow migration on the listener portion of the example that we are following. Lines 7–8 on the original code on Figure 4.2a create the server socket and start the database management system. Figure 4.3a guards these lines with a call to method `Rubah.isUpdating` on line 6.

Note that some systems, like UpStare [MB09], attempt to perform control-flow migration automatically. Following Kitsune, Rubah prefers the manual approach, because (a) it makes the updating process manifest in the program code and thus easier for the programmer to reason about, and (b) it imposes less overhead than would full support for program-wide stack unwinding and rewinding (as in UpStare).

4.2.5 State Transformation

Prior to restarting each thread, Rubah performs state transformation to convert the existing program's objects to use the updated classes. Conceptually, this happens by visiting each object in the heap that was affected by an update and *transforming* it to work with the new version's code. In most cases, this transformation is simple; e.g., version v_0 of a class has two fields while version v_1 has three, and the newly added field is initialized with its default value. In rare cases, the transformation is more involved, and so the programmer can specify what to do in the update class.

Figure 4.4 shows an example of an update class, which specifies the transformation. This example has a single *instance conversion method* that transforms instances of class `Session` by taking an existing instance `o0` that belongs to version v_0 and using it to initialize the equivalent new instance `o1` that shall take `o0`'s place in v_1.

```
1  class Session {
2    User user;
3    // Added in version 1
4    String userName;
5  }
```

```
6  class UpdateClass {
7    void convert(v0.Session o0, v1.Session o1) {
8      // Automatically generated
9      o1.user = o0.user;
10     o1.userName = null;
11     // Customized
12     o1.userName = o0.user.name;
13   }
14 }
```

Figure 4.4: Example, adapted from CrossFTP, of an update class. From version 0 to version 1, class `Session` gained field `userName`, as the left-hand side shows (line 4). The right-hand side shows how the stub update class that Rubah's analyzer tool generates copies all the fields that retain the same name and signature between versions (line 9), and generates a default initialization to the new/modified field (line 10). The developer customizes the stub class with the logic needed to migrate the program state between versions (line 12).

Update classes have one instance conversion method for each class that has a different set of fields from version v_0 to version v_1. Even if the set of fields is the same, with regards to name and type, the developer can define instance conversion methods to account for fields whose semantics changes. If a field has changed neither name nor type, then Rubah copies its value from the old to new version by default; the developer can override this behavior by assigning to the field in the conversion method. Update classes may also define *static conversion methods* to transform static fields.

In Figure 4.4, adapted from the CrossFTP server, field `user` in v_0 keeps information about the current user. The FTP protocol authenticates users through a sequence of USER/PASS messages with the right arguments. Version v_0 drops the connection if the password is incorrect or the user name is not known; version v_1 allows retrying using the same connection. It saves the user name supplied by the last USER command on the new field `userName` to support multiple commands PASS in sequence. The custom migration code just copies the value of the known user name to the new field, in case the update is installed between an USER/PASS interaction.

The arguments of the conversion method in Figure 4.4 are *skeleton classes*, which as the name implies, have been stripped of a lot of the original's contents: All methods are removed, and all fields are made public (so as to be accessible to the update class's code). Each class is placed in a distinct namespace, depending on its version, allowing the developer to refer to version v_0 or v_1 unambiguously and still use the regular Java compiler to compile the update class.

Rubah's analyzer generates a default update class that the programmer may customize. The analyzer compares v_0 and v_1 and matches fields by owner class name, field name, and field type. It generates a conversion method for each class with unmatched/changed fields that copies over the fields that match between versions and initializes the fields that do not match to a default value (`0`, `false`, or `null`). The developer then "fills in the blanks."

Rubah's state transformation algorithms are responsible for finding outdated instances and updating them via the update class. Rubah has two algorithms, a *parallel* one and a *lazy* one, which have different trade-offs and are discussed in detail in the next section.

Note that Rubah does not migrate state kept in local variables. This happens because the stack of each thread is unwound during quiescence by propagating the `UpdatePointException`. There are no active stack frames (and, therefore, no active local variables) at update-time.

My experience with Rubah, and the experience of the authors of the similar Kitsune [HSD+12] DSU system for C, is that values in local variables at update-time are rarely needed. If they are needed, the developer can store them away temporarily just before the update, and retrieve them after the update.

```
1  int inflightOperation;
2
3  void process() {
4    int operation;
5    if (Rubah.isUpdating())
6      operation = this.inflightOperation;
7    else
8      operation = Transfer.readOperation();
9
10   try {
11     switch (operation) {
12       ...
13     }
14   } catch (UpdatePointException e) {
15     this.inflightOperation = operation;
16     throw e;
17   }
18 }
```

Figure 4.5: Example showing how to save local variables during an update. In this example, one of the operations has an update point in the code omitted on line 12. The code that processes each operation, originally shown between lines 55–64 in Figure 4.2, must be changed to support the extra update point. The required changes are highlighted in light gray (lines 1, 10, and 14–17). After the update, the server needs to resume the operation it was performing before the update through control-flow migration. The changes required to perform control-flow migration are highlighted in dark gray (lines 5–7). Any other state kept in local variables that the updatable operation keeps must be saved using a similar program modification.

For instance, in the example that we are following from Figure 4.2, consider that one of the operations takes a long time to complete. The developer can add an update point to the slow operation, so that updates are readily installed when they become available. However, to do so, he has to save, at least, the `operation` local variable that method `process` declares in line 55.

Figure 4.5 shows the manual program modification required to save local variables at update time. This example saves local variable `operation` by creating an instance variable `inFlightOperation` (line 1), which Rubah migrates between versions. When propagating the update point exception, the developer needs to save the local variables he is interested in (`catch` block on lines 14–17). After the update, the developer can use the control-flow migration to reset the saved local variables (lines 5–7).

4.3 State Transformation Algorithms

To perform a Dynamic Software Update, Rubah stops every application thread at an update point. When all threads stop this way, and the application thus becomes quiescent, Rubah transforms the existing program state to an equivalent version that is compatible with the new code. Section 4.2.5 showed how the developer specifies the program state transformation logic. This section explains how Rubah uses that logic to transform the program state between program versions.

Rubah supports two novel state transformation algorithms. The first, *parallel* algorithm transforms all outdated objects eagerly, using multiple threads, while the program is stopped. The second, *lazy* algorithm transforms each outdated object as late as possible, just before the program attempts to use the object after the update takes place. This section describes each algorithm in detail.

4.3.1 Notation

This section uses Java-like pseudocode to present each state transformation algorithm. However, there are some subtle differences to Java code, made for conciseness and readability:

- Brackets are omitted, and indentation determines scope;
- A map `visited` keeps track of visited objects. The map associates outdated objects with either their transformed versions or with themselves, if they have not changed. Expression `visited[key] = val` associates `key` with `val`. Expression `visited[key]` retrieves the current mapping for `key`. If no such mapping (yet) exists, `visited[key]` yields $\perp$;
- Visiting each field in an object, used to compute the transitive closure of the object graph, is written using notation: `for (Field f : obj)` ... `obj.f` ....
- We use atomic *compare and swap* (CAS) to ensure safe concurrency. The expression `CAS(lval,expectVal,setVal)` atomically sets the *l-value* `lval` to `setVal` assuming that `lval`'s contents are currently `expectVal`, in which case `setVal` is returned, otherwise the current contents are. Thus, if `obj.f=0`, then the expression `CAS(obj.f,0,1)` sets `obj.f` to be `1`, and returns `1`, at which point the expression `CAS(obj.f,0,2)` would make no change to `obj.f` and return `1`. We assume the map supports atomic semantics so that `map[key]` can be used as an l-value, i.e., `CAS(map[key],expect, newKey)` denotes an atomic map insertion.

The rest of this section explains how to actually implement these notational conveniences.

4.3.2 Parallel State Transformation Algorithm

The simplest way to transform the program state is to do so *eagerly*, while the program is stopped. A single thread can, starting from the root references, follow each object reference transitively until all the program state is visited and transformed. This is very similar to a *stop-the-world* tracing garbage collection algorithm [JHM11] and *object graph serialization* techniques [VF02,LDS92]. In fact, many DSU systems follow this approach [HSD+12, SHM09, WWS10]. We improve on this basic idea by performing tracing in parallel, using multiple threads.

For the purposes of state transformation, note that Rubah only needs to consider the root references to be the static fields in all loaded classes and the fields in all stopped `RubahThread` objects. It does not need to consider local variables to be root references because the stack of each thread is unwound during quiescence by propagating the `UpdatePointException`.

Figure 4.6 shows the parallel state transformation algorithm as well as a single-threaded variant, for comparison. The main code is in the `migrate` method.

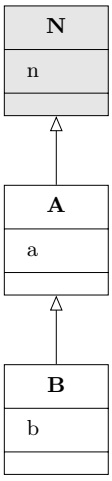
The algorithm calls `migrate(o)` for each root object `o`. This method starts by looking up the object in the map (line 5). If not present, it proceeds to map the old class to the new one by calling method `Rubah.mapClass` (line 8) which, for argument class `c`, returns either class `c` if the update does not modify `c`; or the updated version of outdated class `c`. For outdated objects, the algorithm creates an instance of the new class, and transforms the object (lines 11 and 12).

Method `Rubah.convert(o0,o1)` transfers the state from outdated object `o0` to the updated version `o1` performing the state transformation logic described by the programmer, as described in Section 4.2.5. Note that instance conversion methods are called in a hierarchical way similar to how Java calls constructors [GJS96]. Let us consider the example that Figure 4.7 shows, in which classes `A` and `B` are updatable, class `B` extends `A`, class `N` is non-updatable, and class `A` extends `N`. In this case, to transform instances of class `B`, Rubah: (1) copies all fields inherited from class `N` (line 4), (2) copies all unchanged fields from

```
1  Map visited;
2  TaskQueue queue;
3
4  migrate (Object obj) =
5    if (visited[obj])
6      return visited[obj];
7    Class c = obj.getClass();
8    Class newC = Rubah.mapClass(c);
9    Object newObj;
10   if (newC != c)
11     newObj = Rubah.new(newC);
12     Rubah.convert(obj, newObj);
13   else
14     newObj = obj;
15   Object mapped = map(obj, newObj);
16   if (mapped != newObj)
17     return mapped;
18   traverse(newObj);
19   return newObj;
```

```
20 ST:traverse (Object obj) =
21   for (Field f : obj)
22     obj.f = migrate(obj.f);
23
24 ST:map(Object pre, Object post) =
25   visited[pre] = post;
26   return post;
27
28 MT:traverse (Object obj) =
29   for (Field f : obj)
30     Task t = new Task()
31       { obj.f = migrate(obj.f); }
32     queue.add(t);
33
34 MT:map(Object pre, Object post) =
35   return CAS(visited[pre], ⊥, post);
```

Figure 4.6: Parallel state transformation algorithm. The `traverse` (line 18) and `map` (line 15) methods differ for the single- and multi-threaded variants. Their code is prefixed with `ST` and `MT`, respectively, on the right-hand side. Method `MT:traverse` uses a `TaskQueue`. The notation for creating tasks (lines 30–31) uses braces which form the boundary of a closure: `obj` and `f` are free variables inside task `t` resolved to those in the lexical scope (i.e., the variables defined in lines 28 and 29, respectively). The implementation of three methods is omitted: `Rubah.mapClass` (line 8) takes a single argument `c` and returns either class `c`, if the update does not modify `c`, or the updated version of outdated class `c`; method `Rubah.new` (line 11) creates a new object of the given class without running any constructor; and method `Rubah.convert` (line 12) transfers the state from an outdated object to its updated version, performing the state transformation logic described in the update class.




```
1  B o0; // Existing outdated instance
2  B o1; // Blank new instance
3
4  o1.n = o0.n;
5  o1.a = o0.a;
6  A.convert(o0,o1);
7  o1.b = o0.b;
8  B.convert(o0,o1);
```

Figure 4.7: Order in which Rubah calls conversion methods along the class hierarchy. The left-hand side shows the class hierarchy: Class `N` is non-updatable, classes `A` and `B` are updatable. Class `A` extends `N` and class `B` extends `A`. The right-hand side shows pseudo-code that explains the order in which Rubah copies over the unchanged state and calls the custom conversion methods defined for classes `A` and `B` in the update class.

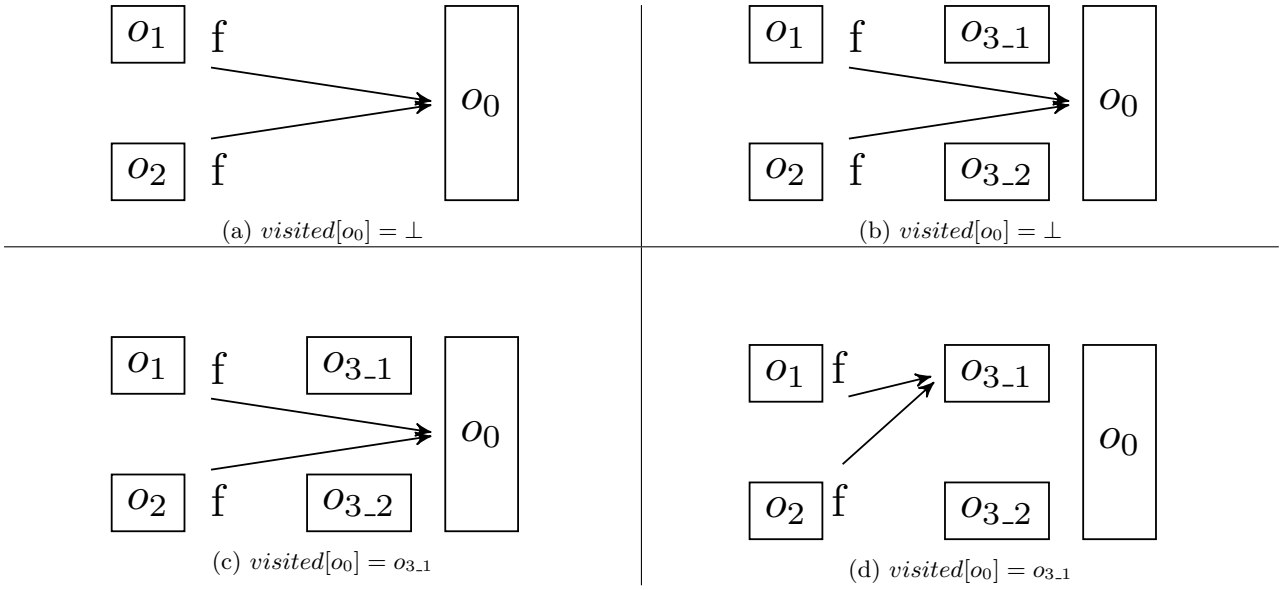


Figure 4.8: Two tasks concurrently try to transform the same object. (a) Task t_1 finds o_0 through $o_1.f$, task t_2 through $o_2.f$. (b) Each task migrates the same object, generating o_{3_1} for t_1 and o_{3_2} for t_2. (c) Task t_1 succeeds at mapping o_0 to o_{3_1}, thread t_2 fails. (d) Both tasks set their respective reference to o_{3_1}.

class `A` (line 5), (3) calls `A`'s conversion method to transform `A`'s updated fields (line 6), (4) copies all unchanged fields from class `B` (line 7), and (5) calls `B`'s conversion method to transform `B`'s updated fields (line 8).

After transforming the object, the parallel state transformation algorithm shown in Figure 4.6 marks the object as visited (lines 15 to 17) and traverses the transformed object (line 18). In the single-threaded variant of the algorithm, traversal is done by the method `ST:traverse`, which simply calls `migrate` for every field that the object has. In this variant, `ST:map` simply updates an object in the map, so the condition in line 16 is always false.

The multi-threaded algorithm uses a `TaskQueue` to coordinate state transformation among multiple threads. The multi-threaded object traversal (method `MT:traverse`) creates tasks to do object transformation for each field (line 32). Each task, itself, creates further tasks and the algorithm finishes when the task threads complete with an empty queue.

There is a possibility of races given the presence of multiple threads concurrently transforming the heap. For instance, consider the case where two threads find the same outdated object o_0 through different referring objects o_1 and o_2, as shown in Figure 4.8a. Both threads find the map empty in line 5 and proceed to convert o_0 (Figure 4.8b).

Allowing both threads to continue at this point would result in field $o_1.f$ referring to o_{3_1} and field o_2 referring to o_{3_2}. This is a clear semantic error. The parallel algorithm avoids it by requiring both threads to synchronize when they try to map the converted object by calling method `MT:map` in line 15, which is implemented using an atomic map update in line 35. Only one thread succeeds at performing the atomic update (Figure 4.8c).

Due to the possibility of races, as described in the previous paragraph, the instance conversion methods presented in Section 4.2.5 must be *idempotent* when using the parallel program state migration algorithm. We shall see that the same restriction applies to the lazy program state algorithm, presented in Section 4.3.3.

Once the parallel algorithm registers a mapping from outdated object o_0 to its updated version o_{3_1}, future threads will find it in line 5 and will not attempt to convert it. The thread that lost the race simply returns the converted object in line 17, while the thread that won traverses the object in line 18 before returning it on the following line. Finally, when returning from method `migrate` in line 31, each thread sets its referring field $o_1.f$ and $o_2.f$ to the converted object o_{3_1} (Figure 4.8d).

4.3.3 Lazy State Transformation Algorithm

Lazy state transformation takes place while the program is running. The goal is to postpone the transformation of each object to the last possible moment. Laziness avoids the significant pause that would otherwise occur for large heaps.

To implement lazy transformation, Rubah uses proxies to intercept control when the program is about to dereference an object (i.e., read/write one of its fields or call a method) that is outdated. The proxy does the necessary work to bring the object up-to-date before allowing the program to continue. To simplify the presentation, I present the algorithm as if every object can behave like a proxy to itself by setting a flag, rather than using a separate proxy object. The start of each method is modified to check the proxy flag, and perform the necessary work if the flag is set, before executing the original method body. Section 4.4 discusses the actual implementation.

Correctness Conditions

Any state transformation algorithm is correct if, once the program threads are restarted after reaching quiescence, they only run up-to-date code and access up-to-date state. This is trivially true for the parallel algorithm.

For the lazy algorithm to guarantee correctness, it must ensure that the restarted threads will only ever use objects that are *safe to access*. In the following text, I explain what it means for an object to be safe to access by presenting two invariants that the lazy algorithms keeps during the execution of the new program version.

Invariant 1: After the update, the program only uses objects that are safe to access.

An object `o` is safe to access if and only if: (1) `o`'s class is not outdated, and (2) either `o` is a proxy (i.e., its proxy flag is set) or all of its fields are safe to access. By ensuring this invariant, Rubah ensures that whenever the program uses an object, the object is either up-to-date or a proxy. In the latter case, the algorithm updates the proxied object's fields so that they are safe to access and uninstalls the proxy before letting the program use the (now up-to-date) object. This approach causes the object graph to have a clear *frontier* between the up-to-date and partially updated program state that is composed of proxies. This frontier starts at the root references and expands outward as more proxies get dereferenced.

In addition to this core invariant, the lazy algorithm maintains another important invariant, which is that all objects mapped to by the `visited` map are safe to access.

Invariant 2: If `visited[o] = p` then `p` is safe to access.

As we shall see shortly, this invariant helps ensure that after converting a proxy object to one that is up to date, the latter is safe to access.

```
1  Map visited;
2
3  LAZYmigrate (Object obj)
4    LAZYtraverse(obj);
5    obj.isProxy = false;
6    visited[obj] = obj;
7
8  LAZYtraverse (Object obj)
9    for (Field f : obj)
10     Object ref = obj.f;
11     if (ref.isProxy)
12       continue;
13     else if (visited[ref])
14       CAS(obj.f, ref, visited[ref]);
15       continue;
16     else
17       Class c = ref.getClass();
18       Class newC = Rubah.mapClass(c);
19       if (c != newC)
20         Object p = Rubah.new(newC);
21         Rubah.convert(ref, p);
22         p.isProxy = true;
23         p = CAS(visited[ref], ⊥, p);
24         CAS(obj.f, ref, p);
25       else
26         ref.isProxy = true;
27
28 Object method(Object ... args)
29   if (this.isProxy)
30     LAZYmigrate(this);
31   // Rest of original method
```

Figure 4.9: Lazy state transformation algorithm. For the sake of simplicity, the pseudo-code assumes that field `isProxy` was injected to every class. Section 4.4 explains how to actually implement proxies. The implementation of three methods is omitted: `Rubah.mapClass` (line 18) takes a single argument `c` and returns either class `c` if the update does not modify `c` or the updated version of outdated class `c`; method `Rubah.new` (line 20) creates a new object of the given class without running any constructor; and method `Rubah.convert` (line 21) transfers the state from an outdated object to its updated version, performing the state transformation logic described in the update class.

Algorithm

Figure 4.9 shows the lazy state transformation algorithm. The algorithm first handles the roots by running the loop on line 9, where each field `obj.f` considered is a root reference.[6] Lines 17 to 19 test if each referred object needs to be transformed. If not, the algorithm simply proxies the object (line 26). Otherwise, the algorithm creates an object of the new class without running any constructors (line 20), runs the conversion code to initialize the new object using the state of the outdated object (line 21), proxies the new object (line 22), marks the old object as visited (line 23), and sets the original reference to point to the new object (lines 24).[7] Note that assigning `visited[ref]` to `p` on line 23 satisfies invariant 2 because `p` is not outdated, and is a proxy. Objects are transformed only once: aliased proxies are skipped (lines 11–12) and aliased objects are set to the correct, safe-to-access object (lines 13–15). For now, assume that the `CAS` operations on lines 14, 23, and 24 always succeed; their role shall become evident later on this section.

At this point all root references refer to proxies. Invariant 1 is therefore true and Rubah can safely start running each paused thread's `run`/`main` method at the new version, beginning the process of control-flow migration. Assuming that all accesses to objects are via method calls, then the next method

[6] Similarly to the parallel algorithm, Rubah only needs to consider the root references to be the static fields in all loaded classes and the fields in all stopped `RubahThread` objects.

[7] Recall from Section 4.3.1 that the first argument of `CAS` is treated as an l-value, not an r-value.

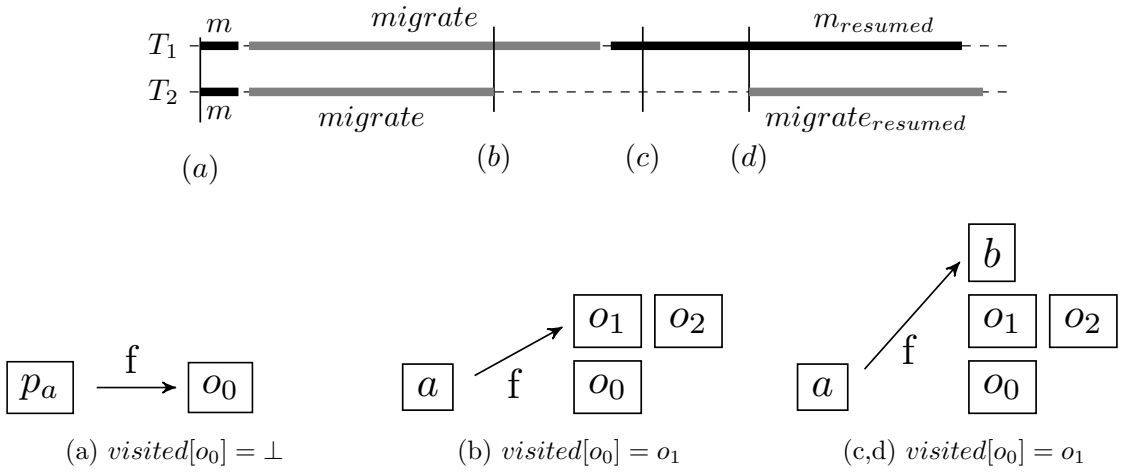


Figure 4.10: Race between lazy state transformation code and application code. The top half shows a timeline of events composed of two threads, T_1 and T_2. The bottom half shows a relevant portion of the heap, composed by proxy p_0, converted object o_1, and objects a and b. Thread T_2 is scheduled out of execution between instants (b) and (d). Both threads start by invoking method $p_a.m$, which triggers the transformation of outdated object o_0 through method *migrate* (short for `LAZYMigrate`).

call on an object will be to a proxy. We assume all methods have been modified according to the bottom of the figure: the program calls method `LAZYmigrate` (line 30), which traverses the proxy (line 4) using `LAZYtraverse`.

As explained above, method `LAZYtraverse` ensures all of the proxy's fields are safe to access: Each field is already a proxy (line 11), is made into a proxy (lines 22 and 26), or was previously visited (line 13), in which case Rubah updates it with the new version from the map. Invariant 2 ensures that this new version is safe to access.

Once a proxied object is traversed, `LAZYmigrate` uninstalls the proxy (line 5), and marks the object as visited (line 6) by mapping the object to itself. Doing so satisfies invariant 2 since the object's fields are all safe to access. This fact also ensures invariant 1 when, at line 31, Rubah resumes running the object's code.

Let us now revisit the uses of `CAS` in the algorithm. Figure 4.10 shows an example that makes clear the need for all uses. At instant (a), proxy p_a refers to outdated object o_0 through field f. Two threads, T_1 and T_2, execute method $p_a.m()$ concurrently, which triggers the execution of method `LAZYmigrate`. Each thread converts object o_0, yielding o_1 for T_1 and o_2 for T_2, and race to register their converted object in the `visited` map. They do so using the atomic update operation on line 23, which ensures that only one thread wins the race and all the threads use the same transformed object. Note that the lazy transformation algorithm may convert the same object more than once, as this example show. As a consequence, the conversion code used with the lazy migration algorithm must be *idempotent*, similarly to the parallel algorithm.

At instant (b), thread T_1 wins the race and T_2 is scheduled out of execution until instant (d). After executing method `LAZYmigrate` to completion, thread T_1 resumes executing the new program method $a.m()$ which performs $a.f = b$ at instant (c). Thread T_2 resumes executing at instant (d) and executes line 24. Note that thread T_2 cannot be allowed to simply perform $a.f = o_1$ because that would overwrite b and thus change the program's semantics and introduce an error. That is why line 24 has a `CAS` operation, `CAS(`$a.f$`,`o_1`,`p_0`)`, which in this case (correctly) fails for T_2. This race is also the reason for the `CAS` operation in line 14.

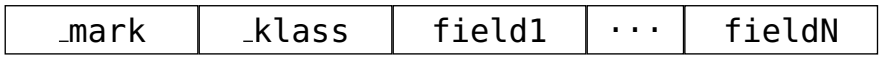


Figure 4.11: Memory layout of a Java object. This object has fields `1` through `N`. The object header uses two words before the first field: `_mark` contains the hash code, lock state, and GC information about this obect; `_klass` refers to a structure that contains information about the class of the object and its vtable.

Assuming that the `visited` map is wait-free, this algorithm is trivially *wait-free*: All operations are guaranteed to finish in a bounded number of steps because there are no loops. The following section explains why the implementation of this algorithm is also wait-free.

4.4 Implementing Efficient Updates

Rubah is the first DSU system for Java that is both full-featured (flexibly handling release-level updates) and VM-independent. This section details the implementation of Rubah's prototype, in particular how the driver (1) rewrites the application, adding support for DSU whilst preserving the original semantics, and (2) performs a dynamic update once one becomes available. Rubah's prototype is implemented in roughly 9000 LOC (lines of code) of Java and uses the ASM bytecode rewriting tool [BLC02, Kul07].

4.4.1 Name Mangling and Class Replacement

Rubah renames updatable classes to distinguish those of different (past and future) versions. A class named `AppClass` gets renamed to `AppClass__0` in version v_0 and `AppClass__1` in version v_1. For brevity, in the following text, I write C_0 for `C__0` and C_1 for `C__1`. Changing the name of a class may break reflection calls, such as `Class.forName`. Rubah rewrites all invocations of these methods to call Rubah's API instead (e.g. `Rubah.classForName`), which provides the same semantics and accounts for name mangling.

When the updater signals that an update is ready (step **6** in Figure 4.1), the driver will load the new classes. Rubah updates classes by generating a new class C_1 for each class C in the new version of the program. Given that there is no relation between the two versions C_1 and C_0 of the same class C, Rubah can support any changes made between versions without requiring any modification to how the underlying JVM represents classes internally.

Rubah generates a new class C_1 for each updatable class C even if class C is unchanged from the previous version. As a consequence, Rubah must transform all instances of C_0 to instances of class C_1. But executing the state transformation algorithms for objects of classes that did not change would be inefficient. Rubah takes advantage of how the JVM lays out objects in memory to avoid such transformations.

Figure 4.11 shows how an object looks in memory. Note that the object's fields are preceded by two words, `_mark` and `_klass`, that contain meta-data about the object. HotSpot finds the class to which any object belongs by looking for that object's reference to its `_klass` structure (Jikes and OpenJDK have a similar object layout). If two classes A and B define the same fields in the same order, we can turn an instance of A into an instance of B by simply modifying the `_klass` reference. The `_mark` word also keeps meta-data about the object. In particular, this word stores the identity hash code of the object, its lock status, and GC information about it.[8]

Rubah uses the unsafe operations available in class `sun.misc.Unsafe` to manipulate `_klass` and `_mark` words. If the structure of a class C does not change between versions, Rubah turns all instances

[8] Assuming the object is not locked. Typical JVM implementations use thin locks [BKMS98, Dic01] that get inflated when an object is locked, effectively moving the hash code to the lock structure. Rubah can deal with either case.

of class C_0 it finds into instances of class C_1 by setting the `_klass` reference. When transforming an object, Rubah migrates the identity hash code using the unsafe API to overwrite the one saved in the `_mark` structure.

Note that, when Rubah installs a new `_klass` reference, it also changes the vtable of the object. Given that the bodies of the methods might change between versions, this effectively installs the new code. This technique is analogous to using HotSwap [Orab, Dmi01] to install new code for loaded classes. It has, however, two important advantages over HotSwap: (1) It does not require the JVM to run in debug mode, which adversely affects JIT performance; and (2) it supports changing the set of methods that a class defines, which HotSwap does not. The downside is that this technique requires a heap traversal to find (and fix) all outdated instances, whereas HotSwap changes the internal representation directly on the class that all instances use.

Changing the `_klass` reference of an object is potentially unsafe because the code that the JIT compiler emits (e.g. when inlining) assumes that the `_klass` reference does not change. As such, changing the `_klass` could crash the JVM. I developed Rubah carefully to ensure that this violation only happens in methods inside of Rubah that never get inlined in the program's code and that such methods never perform any virtual method invocation that might reach the vtable, which is only accessible through the `_klass` structure.

The garbage-collector also uses the `_klass` structure. However, it assumes far less than the JIT compiler about it and just looks for the relevant metadata at a fixed offset for all objects. Given that both the old and the new `_klass` structures agree on this metadata, this optimization does not cause the garbage-collector to crash the JVM.

4.4.2 State Transformation

Rubah's implementation of state transformation largely follows the algorithms given in Section 4.3, with two exceptions: (1) the `visited` map is often implemented as an added field rather than entirely as a separate data structure, and (2) the `isProxy` field is actually implemented by manipulating the `_klass` reference to refer to a proxy class.

Visited Map

The `visited` map from Section 4.3 marks objects as visited and maps outdated v_0 object instances to their v_1 equivalents as they are transformed. Rather than implementing the map entirely as a separate data structure, Rubah adds an extra instance field to updatable and non-updatable classes called `$forward` that points to an object's updated version. This approach adds a small per-object memory overhead, but avoids adding the extra memory pressure at update-time that a separate data structure would impose. It also permits more fine-grained concurrency control: reading or writing the forwarding pointer can be done with a regular compare-and-swap operation.

Unfortunately, not all classes can be changed to add this new field. For instance, the JVM directly accesses the fields in all `java.lang.Reference` subclasses by a fixed index. Adding a field changes the index and cause the JVM to crash. Also, arrays cannot have extra fields. In these cases, Rubah uses an adaptation of `java.util.concurrent.ConcurrentHashMap` that provides the same semantics as `java.util.IdentityMap` and supports an atomic update operation that checks if a key is present, otherwise inserting a mapping in a single step.

```
1  method(Object... args)
2    if (this.isProxy)
3      LAZYmigrate(this);
4    // Rest of method
```

(a) Pseudo-code

```
5  method(Object... args)
6
7    LAZYmigrate(this);
8    this._klass = C1._klass;
9    this.method(args);
```

(b) Real code

```
10 method(Object... args)
11   while (this._klass != C1._klass)
12     LAZYmigrate(this);
13     this._klass = C1._klass;
14   this.method(args);
```

(c) Implicit loop

Figure 4.12: Implementation of proxies. Proxies were originally described as implemented by adding a prefix to every method that checks field `isProxy` (a). In fact, Rubah generates separate proxy classes (b) with methods that convert/traverse the proxied object (line 7), turn the proxy into a regular object by adjusting the `_klass` reference (line 8), and call the method on the converted object (line 9). It is possible that other threads concurrently turn the object back into a proxy after line 8, which makes Java's virtual method invocation on line line 9 call the proxy code again and thus act as an implicit `while` loop (c).

Lazy Proxies.

Section 4.3.3 suggests that a proxy is just an object whose added `isProxy` flag is set, where the flag changes how methods work. Checking this flag would degrade performance at the entrance of every method. Furthermore, given that the JVM JIT optimizing compiler aggressively inlines small methods, the flag check would increase the code size of all methods, making the JIT compiler miss inline opportunities for small methods and thus generate slower code. It would also require an extra field in every class.

Instead, Rubah generates a proxy class to hold the proxy code and turns regular objects into proxies, and proxies back into regular objects, by manipulating the reference to the `_klass` structure through unsafe operations in the same way we describe in Section 4.4.1.

Rubah generates a proxy class C_P for each class C that it loads. C_P extends C and overrides all of C's methods, redirecting the control flow to Rubah's API.[9] Proxies thus inherit the fields of the classes they extend, having the same layout as the object they proxy, with the only difference between an object and a proxy is the vtable it keeps.

Changing the vtable through the `_klass` pointer makes proxies intercept virtual method invocations. However, besides those, proxy classes must also intercept other ways that the proxied object might be manipulated, which are field accesses and non-virtual method calls.[10]

Rubah rewrites all field accesses so that they are made through accessor methods, which can be overriden and intercepted by proxy objects. When such accessors are called from within the class's own methods, the JIT safely optimizes the call away by inlining; the only overhead will be due to accesses from outside the class.

For non-virtual calls, there is no issue if the call is made via `this` or `super`, since the current object cannot be a proxy. The only time the receiver of a non-virtual call can be a proxy is when invoking a private method of a different object (having the same class). Rubah emits a check before the invocation to ensure that the other object is not a proxy; if it is, it must be transformed.

[9] Rubah removes all `final` modifiers from classes and methods (but not fields) it loads to ensure that every class and method can be proxied. There are some classes in the `java.lang` package that do not support this, such as `java.lang.String`, but these classes are never proxied.

[10] In Java, calls to private methods are non-virtual, as are calls to methods via `super`.

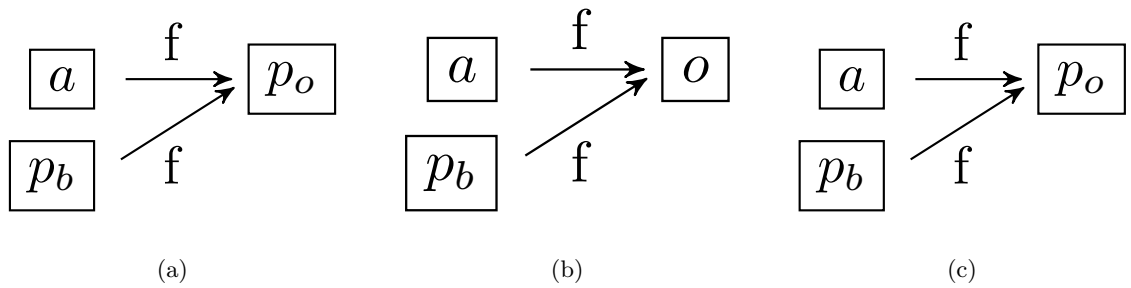


Figure 4.13: Progress during lazy program state transformation. In this example, two threads are executing the code on Figure 4.9 : T_1 is running method `LAZYmigrate(`p_o`)` and T_2 is running method `LAZYtraverse(`p_b`)`. At (a), T_1 is at line 5 and T_2 is at line 10 (`a.f=`p_o). At (b), T_1 executes line 5 (`a.f=`o). At (c), T_2 executes until line 26, including it (`a.f=`p_o, again). Note that $visited[o] = \perp$ throughout the whole execution. Object a is the reason why two threads can get to the initial configuration.

Both cases described in the last two paragraphs are very rare in typical Java programs, and the extra calls should not add any measurable overhead in practice.

Wait Freedom

The pseudocode of the algorithm given in Figure 4.12a (copied from Figure 4.9) checks the `isProxy` flag on line 2 to determine if an object is a proxy, and calls `LAZYmigrate` if so, before continuing with the body of the original method. In Rubah's actual implementation, shown in Figure 4.12b, the `_klass` reference has been modified to refer to a proxy class whose methods consist simply of a call to `LAZYmigrate`, followed by a call to `this.method(args)`. Because `LAZYmigrate` resets the `_klass` reference to that of the original object (equivalent to resetting the `isProxy` flag on line 5 in the `LAZYmigrate` method shown in Figure 4.9), this call executes the correct method.

However, there is a possibility that another thread could re-proxy the object after line 8. When this happens, line 9 calls the same proxied method. There is an implicit loop, made explicit in Figure 4.12c, that executes method `LAZYmigrate` while the `_klass` pointer is set to the proxy class. Now we must be concerned: Is it possible that a thread will be stuck in the while loop forever, thus violating wait freedom? Fortunately, the answer is '*no*'.

Consider the following scenario, shown in Figure 4.13: An object o is aliased by two objects such that $a.\texttt{f} = b.\texttt{f} = o$. Furthermore, object b is proxied (denoted by p_b). Two threads, T_1 and T_2, are executing the lazy transformation algorithm, shown in Figure 4.9, concurrently. Thread T_1 traverses $a.\texttt{f}$, proxies o (e.g., in line 26), and then calls a method on proxied p_o (Figure 4.13a). Because p_o is a proxy, this prompts Rubah to traverse it, eventually executing line 5 (Figure 4.13b). But before T_1 can execute line 6, suppose thread T_2 traverses $p_b.\texttt{f}$ and, because o is not marked as visited yet, the test on line 13 fails and T_2 re-proxies o (Figure 4.13c).

The similarity between Figures 4.13a and 4.13c suggests that the lazy migration algorithm can enter a loop where no progress is made. In fact, at this point, when thread T_1 returns from method `LAZYmigrate`, the guard in the explicit while loop on Figure 4.12c is true and `LAZYmigrate` is called again. However, notice that thread T_1 marked o as visited in line 6. The next time a thread finds o while traversing an object, the conditional in line 13 is true, so the object is not re-proxied again. Therefore, once T_1 executes line 5 the second time, the object will be de-proxied permanently.

The worst case scenario is if half of the threads in the application behave as T_1 and the other half as T_2, alternately, as in Figure 4.13. However, because there is a bounded number of threads, there is a bounded number of times that a proxy can be installed and uninstalled in sequence for the same object.

Assuming that the map `visited` is wait-free, it follows that there is a bound on the number of steps required for each proxy to break out of its while loop. Therefore, I can state that the implementation of the lazy state transformation algorithm remains *wait-free*.

4.4.3 Bytecode Rewriting

Rubah rewrites the original program at class load time. The rewritten program preserves the original program semantics and supports DSU. All the bytecode transformations have already been described, scattered throughout the preceding sections. This section enumerates all transformations, summarizing each one and specifying where each transformation is described in detail.

Class Renaming. Rubah renames the updatable classes to avoid name collisions between multiple versions of the same class. Class `C` becomes class `C__0` in version 0 and `C__1` in version 1. Section 4.4.1 describes class renaming in full detail.

Field Redirection. Rubah's lazy algorithm is based on the assumption that Rubah can intercept all possible ways of manipulating objects. Rubah intercepts method invocation by generating proxies that extend the proxied classes and override all methods. Rubah intercepts field manipulations by rewriting them into calls to accessor methods, that Rubah adds to each class. Any other accesses to objects (e.g. bytecode instruction `instanceof`) are always preceded by a method call or a field manipulation to load the object to the operand stack. Section 4.4.2 describes field redirection in full detail.

Class/Method Protection. To generate proxies, Rubah needs to be able to inherit from all classes and override all methods. When a class or a method is annotated with the modifier `final`, it is not possible to extend it or override it, respectively. Rubah removes the `final` modifier from every class it loads. Section 4.4.2 describes field redirection in full detail.

Note that this transformation has no impact on performance because the JIT compiler does not use this information. It instead optimizes on a per call-site basis. The performance evaluation on Section 4.5.4 provides empirical evidence to this claim.

Hash code. The identity hash code is located in the header of the object on field `_mark`, near field `_klass`. Rubah uses unsafe low-level operations to copy the identity hash code for the same objects between different versions. Section 4.4.1 describes field redirection in full detail.

This approach does not work for all classes. When performing an update, the JVM creates objects relevant to the new program version and stores them before Rubah has a chance to migrate the identity hash code. This happens for objects of class `java.lang.Class` that represent classes of the new program version. In these cases, Rubah injects an integer code to keep the hash code and overrides method `hashCode` to return the extra field. These cases are very rare and do not add any noticeable performance overhead.

Reflection/Unsafe. The updatable program may rely on reflection and on the same unsafe API that Rubah uses. Rubah rewrites all sensitive reflection calls that might expose the bytecode transformation (e.g. `java.lang.Class.forName` or `java.lang.Class.getName`) to call an equivalent Rubah API instead that behaves as the original program. Rubah also rewrites calls to the unsafe API that can access proxies directly and thus override how Rubah intercepts their manipulation. Section 4.4.1 describes field redirection in full detail.

4.4.4 Portability Among JVMs

Rubah was tested on Oracle's HotSpot JVM. It does not modify any part of it, but it relies on a number of assumptions about it. In particular, Rubah (1) uses "unsafe operations" to read fields directly, circumventing access checks and bounds checks, and to compare-and-swap on arbitrary memory locations; and (2) assumes the JVM lays out fields in the same order along the class hierarchy, and places each object's vtable in a fixed location accessible to unsafe operations. Besides Oracle's HotSpot, IBM's Jikes and OpenJDK also satisfy these assumptions.

4.5 Evaluation

This section reports an experimental evaluation of Rubah along three axes:

Programmer effort How difficult is it to retrofit an application to use Rubah? How difficult is it to write an update class (which describes how to transform the application's state)? How flexible is Rubah (which types of changes does it support)?

Steady-state overhead How much slower is the normal operation of the Rubah-retrofitted version of an application than its unmodified version?

Per-update overhead How is the performance of an application negatively affected while the update is being installed? That is, how long is the application paused and/or its performance degraded?

To provide empirical data for each of these questions, I designed a series of experiments that use Rubah to run existing server applications in one process while another process benchmarks the server. Some experiments involve updating the server process during a benchmark run.

4.5.1 Updatable Applications

Rubah was used to dynamically update the following five applications:

1. **H2**, an SQL DBMS written in Java;
2. **Voldemort**, a key-value store used by LinkedIn;
3. **CrossFTP**, an FTP server;
4. **Jake2**, a Java port of the shooter game Quake 2;
5. **JavaEmailServer**, a POP3/SMTP mail server.

All these applications are long-running, and maintain important in-memory state (the database/store contents, the game state, and/or the protocol's state for each client) that would be lost on restart.

Rubah was able to install application releases as dynamic updates. The first three columns on Table 4.1 list the application versions, their size, and how they changed between releases. H2 changed considerably in the considered releases: Among other changes, developers implemented support for new SQL commands/idioms and full-text search, and improved the performance of H2's page store.

Voldemort did not change as much: the new release fixes a race and improves throttling when cleaning up data after rebalancing a server cluster.

CrossFTP added support for new configuration options for the PASV command and the international character encoding for directory lists. JavaEmailServer added support for limiting the maximum size for incoming messages, maximum delivery attempts before dropping a message, and relaying messages based on the recepient's address.

Version	Release Code *(#lines / #files)*	Release Changes *(#classes / #methods / #fields)*	Retrofit Modifications *(#lines / #files)*	Update Class *(#stub / #mod) LOC*
H2				
1.2.121	40119 / 98	-	267 / 9	-
1.2.122	40566 / 98	63 / 149 / 12	Same	106 / 45
1.2.123	40655 / 99	44 / 86 / 3	Same	40 / 30
Voldemort				
1.5.3	87516 / 517	-	175 / 7	-
1.5.4	87539 / 517	8 / 12 / 2	Same	12 / 2
Jake2				
0.9.5	85408 / 256	-	29 / 2	-
CrossFTP				
1.07	18221 / 161	-	224 / 8	-
1.08	18108 / 161	9 / 20 / 1	Same	16 / 1
1.09	18173 / 160	30 / 58 / 4	+4 / Same	47 / 2
1.11	18435 / 161	10 / 34 / 11	Same	51 / 23
JavaEmailServer				
1.3.3	2368 / 20	-	183 / 6	-
1.3.4	2447 / 20	5 / 11 / 1	Same	26 / 2
1.4	2529 / 20	7 / 17 / 3	Same	55 / 9

Table 4.1: Changes between releases and programmer effort to support Rubah. Column *release code* shows the total lines of code, excluding comments and blank lines, and number of files on the original application. Column *release changes* shows the code changes between the previous release, in terms of modified classes, methods, and fields. Column *retrofit modifications* shows how many lines of code I added/modified to support Rubah and how many files were changed. Column *update class* shows the lines of code (LOC) of the automatically generated update class file and the number of its lines I added/modified.

4.5.2 Programmer Effort

Table 4.1 assesses the programming effort to use Rubah on each supported application. In total, I retrofitted four versions of CrossFTP, three versions of H2 and JavaEmailServer, two of Voldemort, and one of Jake2 (as other versions lack sufficiently different functionality). The fourth column counts the number of files and lines affected by retrofiting each application to use Rubah. For all five, I added update points to long-running loops and control-flow migration as described in Sections 4.2.3 and 4.2.4, respectively.

To be updatable through Rubah, applications must reach update points shortly after an update is available. In the following paragraphs, I describe how I placed update points so that each application can always reach them.

When idle, all applications wait either for new clients to connect, or for new requests from connected clients. I retrofitted each application so that it can be interrupted while waiting. For H2, CrossFTP, and JavaEmailServer, I changed I/O calls to use Rubah's equivalent interruptible calls that use non-blocking I/O[11] (accounting for 134 and 49 LOC, respectively); Voldemort already uses non-blocking I/O; and Jake2 polls I/O frequently rather than blocking.

[11] Rubah's I/O library does not support SSL at this point, so I commented out CrossFTP's code that uses SSL. Supporting SSL is just a matter of engineering effort.

When active, each application processes requests from clients: H2 processes SQL commands, Voldemort processes read/store operations, Jake2 processes network frames, CrossFTP processes FTP commands, and JavaEmailServer processes POP3/SMTP commands. I retrofitted each application so that it finishes processing the current requests it already started processing at the time an update became available, and reaches an update point before starting to process any new requests.

Some applications might take a long time processing some requests. For instance, CrossFTP RETR/STOR commands involve sending/receiving an arbitrarily large file over the network. To avoid large periods of quiescence, while the server is not accepting new clients/requests because an update is available but it cannot start the update process because it is executing such a command, I took advantage of the presence of a transfer buffer that CrossFTP fills before sending/receiving data and added an update point reached when the buffer gets filled.

The numbers that table 4.1 reports are consistent with the numbers reported for dozens of updates to six C applications using Kitsune [HSD+12]. The number of required changes is relatively small and not strongly correlated with program size, but rather with its control structure—notice that Jake2 required only 29 lines changed compared to 183 for Java Email Server, but is actually larger, 85K LOC versus 2K LOC. Moreover, as indicated by the table, no new changes were required for subsequent versions of H2, Voldemort, and JavaEmailServer. I expect that retrofitting an application to support Rubah is, like Kitsune, a modest, largely one-time cost.

For H2, Voldemort, CrossFTP, and JavaEmailServer, I developed update classes to implement state transformation between the supported versions; the fifth column provides some data about these classes. We can see that stub update classes eased the burden placed on the developer: The maximum number of lines that I had to modify was 45. These updates were tested by running standard benchmarks (described in Section 4.5.3), updating while they were underway, and confirming the integrity of the final results.

4.5.3 Experimental Setup

The experimental evaluation described in this section measures Rubah's influence on an application's steady state performance, and its performance at update time. Measurements were carried out on a machine equipped with two Intel Xeon E5520 processors (8 physical cores, 16 logical) and 24GB of RAM running Ubuntu 10.04 (Linux kernel 2.6.32). I used the Oracle JVM version 1.7.0_25 with HotSpot 64-Bit Server VM (build 23.25-b01) configured to use a maximum heap size of 16GB for the server and 2GB for the client.

All of the experiments start the application server process and then launch a separate client process that executes a performance benchmark that interacts with the server and measures its performance. To measure *steady-state overhead* I compare the performance of the unmodified server with that of the Rubah-enabled one—no updates are performed. To assess *per-update overhead* I update the Rubah-enabled server in the middle of the benchmark run and measure the performance impact of doing so. In addition to performing a real update from one version to the next (which I call a *v0v1* update), I also consider a *v0v0* update, which installs the same version that the program is running, but considers all classes incompatible and transforms all the updatable program state, copying (and not simply adjusting the `_klass` pointer, as described in Section 4.4.1) all instances while the program state traversal takes place. This is a good approximation of a worst case scenario. Given that I lack a usable second release for Jake2, v0v0 updates were the only way to test updating it.

To measure H2's performance, I used the TPC-C benchmark available in the DaCapo benchmark suite [BGH+06] as the client process. We can configure the TPC-C benchmark with the number of transactions to run and the size of the database to create before running the workload. The database size is expressed in terms of a *scale factor* with which TPC-C multiplies the number of rows in several tables it creates. I configured the H2 server to keep all data in memory.

Voldemort ships with a performance benchmark that I used as the client process. The benchmark has several configurable parameters. The most interesting are: The number of operations to perform, number of key-value pairs created before running the workload, the size (in bytes) of each stored key, and the ratio of read and write operations performed by the workload. Besides these parameters, I extended the benchmark with support to run the workload for a fixed period of time (as opposed to a fixed number of operations). I configured Voldemort's server in a single node setting, with all the data in memory. The benchmark executes a realistic mix of 95% read and 5% write operations [BAC+13].

To evaluate CrossFTP, I implemented an FTP benchmark. Existing FTP benchmarks focus on measuring the bandwidth of file transfer, typically downloading/uploading the same file as many times as possible over the duration of the benchmark. This workload does not exercise the parts of a server that deal with other FTP commands, e.g. browsing the file structure. The benchmark I implemented connects to a remote FTP server, randomly browses the directory structure in a depth-first manner, and downloads the first file it finds. The benchmark spawns multiple threads, each one representing one client. I configured CrossFTP to serve a directory tree that is $D = 2$ levels deep, with each non-leaf folder containing $W = 10$ sub-folders, and each leaf folder containing a file. Files have random contents with sizes in the range 2MB–300MB following an exponential distribution with a mean size of 50MB. I chose these parameters so that the workload resembles a repository of binary software packages used by current GNU/Linux distributions, which are typically made available through an FTP server.

Jake2 and JavaEmailServer lack an automated performance benchmark. Thus, I only measure the pause time resulting from applying an update to an idle process; for Jake2, this is a v0v0 update after loading the game state, and for JavaEmailServer it is v0v1 update after startup.

4.5.4 Steady-State Overhead

To measure steady-state overhead, I compare the performance of the unmodified server, referred to as *vanilla*, with that of a Rubah-enabled one that does not perform any update.

I ran the following experiments for each application:

- **H2** Measure the time it takes to run 256K transactions on a database with a scale factor of 32;
- **Voldemort** Measure the time it takes to run 25M operations over a key-value store populated with 5M entries of size 128 bits;
- **CrossFTP** Measure the bandwidth used during a 5 minute run with 8 client threads.

I repeated each experiment 10 times. Table 4.2 reports the median and semi-interquartile range of 10 benchmark runs. We can see that the imposed overhead is between −1.0% and 2.5%. This range is well within the experimental noise on modern systems [MDHS09]. I thus claim that *Rubah imposes no measurable overhead in normal execution.*

Version	Vanilla	Rubah	Overhead
H2 Elapsed time (seconds)			
1.2.121	350.5 ± 6.4	351.4 ± 3.9	0.3%
1.2.122	348.8 ± 7.0	350.0 ± 3.5	0.8%
1.2.123	347.1 ± 7.0	350.0 ± 3.5	0.8%
Voldemort Elapsed time (seconds)			
1.5.3	469.1 ± 3.1	471.6 ± 1.3	0.5%
1.5.4	469.1 ± 2.5	473.7 ± 3.5	1.0%
CrossFTP Bandwidth (Mbps)			
1.07	829.9 ± 5.5	811.1 ± 15.6	2.3%
1.08	827.8 ± 3.7	813.7 ± 6.0	2.5%
1.09	801.8 ± 4.9	809.7 ± 5.5	-1.0%
1.11	803.5 ± 14.1	809.1 ± 3.4	-0.7%

Table 4.2: Results of benchmark runs, with and without Rubah, thus reporting steady-state performance. Reported values are the median and semi-interquartile range of 10 benchmark runs. Overhead is computed by $(Rubah/Vanilla) - 1$. Size is measured in scale factor, for H2 (explained in detail in Section 4.5.3); and in number of key-value pairs, for Voldemort.

4.5.5 Parallelizing State Transformation

When eagerly transforming the program state, the application does not execute while Rubah threads traverse and transform the heap. To measure the benefits of parallelizing eager state transformation, I configured each benchmark to install an update 10 seconds after populating the server with test data, and measured the time Rubah took to perform parallel transformation.

I repeated the experiment for both *v0v0* and *v0v1* updates (1.2.121 to 1.2.122 in H2's case) and with a varying number of transformation threads (1, 2, 4, 8, 12, and 16). The H2 benchmark used a database with a scale factor of 32 and the Voldemort benchmark used a key-value store with 5M entries. CrossFTP, Jake2, and JavaEmailServer keep such a modest amount of program state that increasing the number of threads does not influence the state transformation time, and are thus excluded from this experiment. Table 4.3 reports the average and standard-deviation of 10 executions of each setting.

Comparing to single-threaded transformation, Rubah achieves speedups using up to 16 threads on H2 and Voldemort, despite the fact that the test machine has only 8 physical CPUs. The *v0v0* case has more work to do per object, therefore sees a higher speedup than the *v0v1* case. In Voldemort's case, changing from 1 to 2 threads yields little or no speedup, and sometimes even slows down. This happens because 2 threads create a much larger number of in-flight conversions, thus creating a larger task queue and triggering more garbage collections. Adding more threads amortizes this added memory pressure.

4.5.6 Performing Updates

Installing an update temporarily pauses the application while waiting for the threads to quiesce, and then while loading the new classes. On top of that, when transforming the program state eagerly, the application remains paused while Rubah threads traverse and transform the heap. The next experiment measures the pause introduced for both the eager and lazy state transformation algorithms, as well as the impact on post-update performance.

To approximate the pause due to an update, this experiment measures the maximum server latency that the client ever experiences during the benchmark run. I expect that the pause induced by an update will dwarf the normal latency a client would experience, especially in this experimental setting that has negligible network latency.

Num. Threads	v0v0		v0v1	
	Time (sec)	Speedup	Time (sec)	Speedup
	H2			
1	31.2 ± 0.7	1	18.8 ± 0.7	1
2	19.0 ± 0.5	1.7	12.3 ± 0.5	1.5
4	12.6 ± 0.3	2.5	9.2 ± 0.2	2.0
8	10.0 ± 0.2	3.1	8.2 ± 0.4	2.3
12	9.3 ± 0.3	3.3	8.1 ± 0.3	2.3
16	9.2 ± 0.2	3.4	7.8 ± 0.2	2.4
	Voldemort			
1	42.5 ± 1.0	1	29.2 ± 0.9	1
2	39.5 ± 1.0	1.1	30.4 ± 1.1	0.9
4	22.0 ± 0.8	1.9	18.0 ± 0.9	1.6
8	13.7 ± 0.7	3.0	12.1 ± 1.0	2.5
12	12.6 ± 0.7	3.3	10.6 ± 0.5	2.8
16	12.0 ± 0.4	3.5	10.7 ± 0.4	2.7

Table 4.3: Elapsed time (in seconds) of parallel state transformation. The first column under each benchmark is the median time and semi-interquartile range, in seconds, required to transform the program state. The second column is the speedup relative to one thread. Reported values are the average and standard-deviation of 10 benchmark runs. The H2 benchmark used a database with a scale factor of 32 and the Voldemort benchmark used a key-value store with 5M entries.

For each application, the experiment consists of:

- **H2** Executing the benchmark for 256K transactions; with a database scale factor of 32, 64, and 128; and performing an update from version 1.2.121 to 1.2.122 at T=60 seconds; while measuring the maximum time each successful[12] SQL command takes to execute;

- **Voldemort** Executing the benchmark for 20 minutes; with a store size of 1M, 5M, 10M and 15M key-value pairs; and performing an update from version 1.5.3 to 1.5.4 at T=300 seconds (5 minutes); while measuring the maximum time each store read/write takes to complete;

- **CrossFTP** Executing the benchmark for 5 minutes and performing an update from version 1.08 to 1.09 at T=10 seconds while measuring the time the server takes to reply to each CWD/LIST command and the time it takes to fill a 4MB transfer buffer when downloading a file;

- **Jake2 and JavaEmailServer** Perform an update, from version 1.3.4 to 1.4 for the JavaEmailServer and v0v0 for Jake2, after loading and initializing the servers, while they are idle (since I had no good automated performance benchmark to use).

Besides considering a real update, I also performed each experiment with a v0v0 update (for Jake2, that was the only experiment possible). I repeated each experiment 10 times for the parallel (using 16 threads) and lazy state transformation algorithms. Table 4.4 reports the measured pause times.

For Jake2, both the parallel and lazy algorithms induce short pauses. I confirmed that the update was non-disruptive by playing several matches of Quake2 while performing the update;[13] the Quake2 client already tolerates network latency and the Jake2 server keeps a very small program state.

[12] The TPC-C benchmark issues commands that timeout due to table locking made by other commands. We discard such unsuccessful commands.

[13] I thank Piotr Mardziel and James Parker for having helped me evaluate Jake2 by playing several matches during regular work hours.

Size	v0v0		v0v1	
	Parallel	Lazy	Parallel	Lazy
H2				
32	11.0 ± 0.3	3.3 ± 0.2	9.0 ± 0.1	3.1 ± 0.1
64	20.9 ± 0.8	3.7 ± 0.4	15.3 ± 0.6	3.7 ± 0.1
128	71.0 ± 1.2	4.0 ± 0.5	30.9 ± 0.9	3.7 ± 0.3
Voldemort				
1M	4.9 ± 0.3	1.5 ± 0.3	4.4 ± 0.4	1.9 ± 0.4
5M	13.5 ± 1.0	1.6 ± 0.6	10.7 ± 0.8	2.2 ± 0.5
10M	24.7 ± 1.8	1.6 ± 0.5	19.1 ± 2.1	2.2 ± 0.5
15M	158.2 ± 7.1	1.8 ± 0.5	107.4 ± 0.8	2.4 ± 0.4
Jake2				
	1.5 ± 0.1	1.2 ± 0.1	-	-
CrossFTP				
	0.33 ± 0.04	0.35 ± 0.08	0.35 ± 0.07	0.44 ± 0.06
JavaEmailServer				
	0.11 ± 0.01	0.09 ± 0.01	0.10 ± 0.01	0.09 ± 0.02

Table 4.4: Pause time (in seconds) required to install each update under various heap sizes. Reported values are the median and semi-interquartile range of 10 benchmark runs. The first column is the size that each benchmark used to populate the server with test data (scale factor for H2 and number of key-value pairs for Voldemort). The parallel transformation used 16 threads.

CrossFTP and JavaEmailServer also have small pause times. These programs keep a modest amount of program state—consisting of meta-data about each connected client (e.g. current working directory, transfer mode, permissions for CrossFTP; authentication state, list of messages, message being composed for JavaEmailServer)—so Rubah takes little time to traverse and transform their state. The parallel algorithm has slightly better results than the lazy algorithm. I interacted manually with JavaEmailServer through a telnet client connected to the POP3/SMTP port while an update was taking place to ensure that the server does not drop connections or any session data due to the update process.

Table 4.4 shows the update-time pause for a variety of heap sizes for H2 and Voldemort. The pause times for the parallel algorithm, shown in the second and fourth columns, grow with the heap size, as expected. For larger heaps, the update pause causes a pronounced increase in the maximum latency.[14] For the lazy algorithm, we can see that the update pause is constant regardless of the total heap size, and is quite small compared to the parallel algorithm.

Besides measuring the maximum time taken to complete each operation, the experiments previously described in this section also keep track of how many operations the benchmark completes per second, for H2 and Voldemort. Figure 4.14 shows the performance data measured during the experiment. The figure presents plots where the x-axis is elapsed time, and the y-axis is the throughput in transactions/operations per second.

The left column of the charts in the Figure 4.14 shows the results when using parallel state transformation. These charts show a sharp performance drop when performing the update, followed by a rapid rise back toward the pre-update peak. There are two things to notice: First, the update pause increases with the heap size, particularly for *v0v0*, which must traverse all of the heap. Second, we see that performance does not completely return to its pre-update level. We observed a similar drop in steady-state performance on an experiment that traverses the whole heap starting from the root references without

[14]Note that the results that Table 4.4 reports are not directly comparable to those of Table 4.3; e.g., the numbers in Table 4.3 for Voldemort are measured for an update taken at T=10 seconds, but for Table 4.4 the update is at T=300 seconds. For the latter, I measured a heap transformation time of 10.3 seconds for Voldemort v0v1 at 5M (out of the 10.7 second pause reported in Table 4.4), which is slightly less than the 10.7 seconds reported in Table 4.3, but within the reported error range.

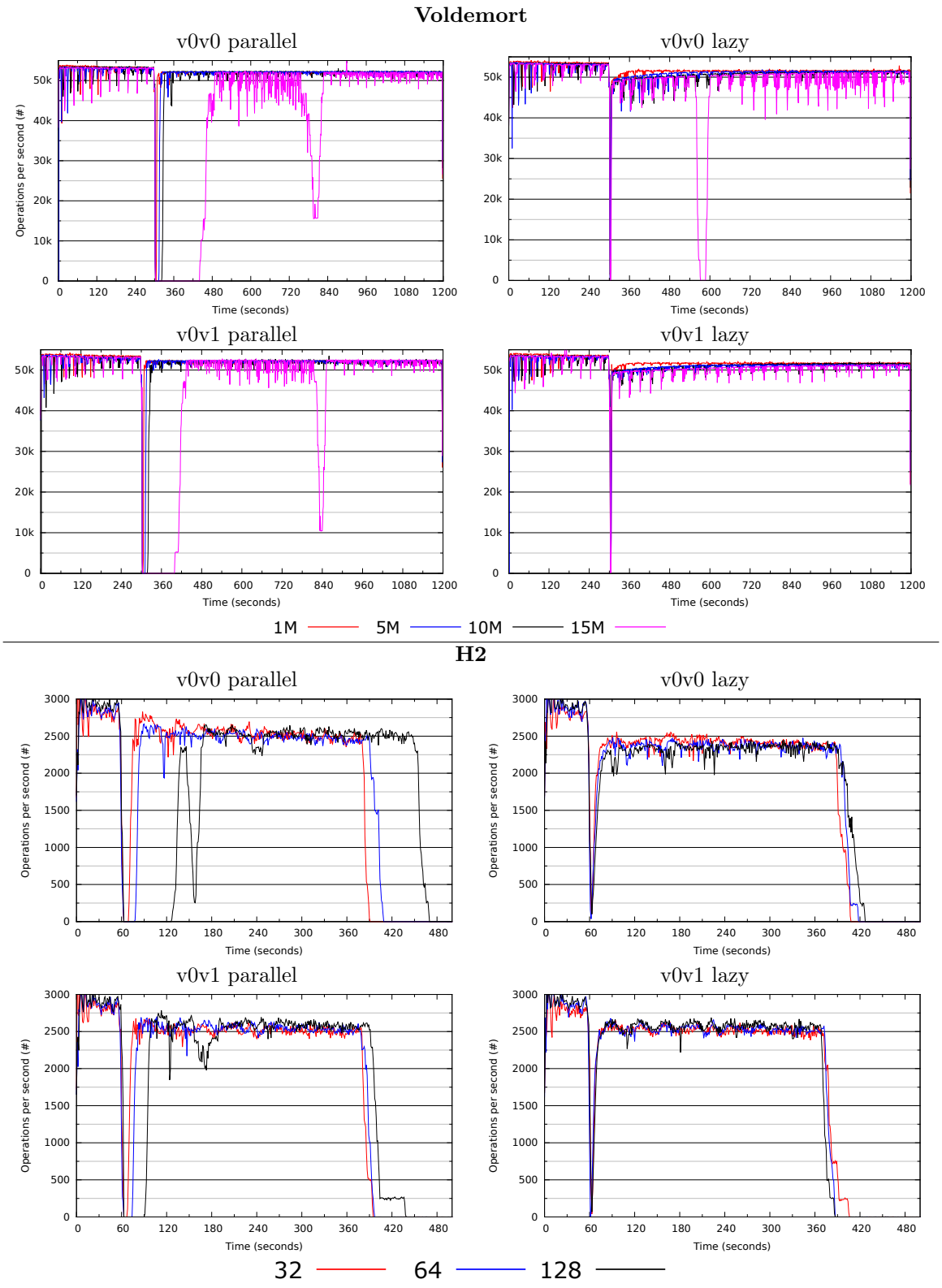


Figure 4.14: Plotting of Rubah's performance while installing an update under varying heap sizes. Each line shows the performance for a different-sized database (the line label is the scale factor for H2 and the number of key-value pairs for Voldemort). Each line reports the average of 10 benchmark runs. The occasional performance dips are due to garbage collections; their number and magnitude indicate the level of memory pressure.

making any changes. I thus believe that the performance drop is due to a change in some internal JVM state triggered by the state traversal. On a separate experiment, I installed a second update for the parallel v0v0 case and confirmed that the performance after the second update reached the same levels as the performance after the first update.

The right column of the charts in Figure 4.14 plots performance when using lazy transformation. The key feature is the far smaller drop in performance at update-time, after which performance slowly rises, depending on the heap size. Table 4.4 shows that the update pause is constant, regardless of the total heap size, and is quite small compared to the parallel algorithm.

Returning briefly to Figure 4.14, we note that the experiment also shows that lazy transformation does disproportionately better than the parallel transformation with larger heaps. For Voldemort, the 15M case consumes nearly the entire heap. The parallel transformation makes the GC thrash, triggering numerous full-GC cycles that are not able to free much memory. The lazy algorithm performs much better. It still triggers one full-GC cycle, but that one cycle actually frees enough memory to keep the GC from ever thrashing. We can see a similar thrashing pattern for H2's 128 *v0v0*, even though it happens after the update is installed.

Figure 4.14 also shows that the lazy algorithm converges very quickly to steady-state performance after the update. Several reasons contribute to this behavior: Converting an object lazily is fast, thus imposing a small performance penalty; the working set of each application is small and constant despite the heap size; and the rate of lazy object conversion drops quickly after the update.

4.5.7 Post-update performance

The results on Figure 4.14 show that the post-update drop of peak performance is larger for the lazy case, compared to the parallel one. This happens due to decisions that the JIT compiler takes when optimizing the code immediately after an update, when proxies are present and used frequently. To diagnose this behavior, I ran Oracle's HotSpot JVM with debug flags that log the optimization decisions the JIT compiler makes.[15] The compilation logs show differences in how the JIT optimizes virtual method invocation after a lazy update. The following text explains those differences, and why the peak performance is lower.

The JVM supports method dispatch based on the runtime type of the object in which the method is executed — the *receiver*. The JVM uses the object's vtable to choose the most specific concrete method to execute when performing a virtual method invocation. However, looking up the vtable at each invocation is costly and the JVM optimizes for the common cases, as follows:

1. **Single method always invoked:** The JIT compiler inlines the method at the call site, protected with a trap that checks the receiver object type and ensures that the inlined code is correct for each call;

2. **Two methods always invoked:** Similar to 1, the JIT compiler inlines both methods after a conditional branch that jumps to the right method. A trap is also used to ensure correctness;

3. **More than two methods invoked** The JIT inlines the two most frequently invoked methods, as in 2. The JIT emits code to perform a *slow* virtual method call that consults the object's vtable for all the other concrete methods.

After a lazy update, some virtual method invocations are resolved to proxy methods. This affects the JIT compiler's optimization decisions, turning (a) case 1 into 2, (b) 2 into 3, or (c) changing the two methods inlined in case 3. For (a), the size of the optimized code increases due to the extra inlined method.

[15] -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation

This might prevent the optimized code from being itself inlined elsewhere, resulting in a performance penalty. For (b) and (c), the JIT might decide to inline proxy code. As the program executes after an update, the number of proxies found by the code drops because they get transformed. This optimization thus increases the number of slow virtual method invocations that use the vtable and bypass the inlined code, which in turn yields lower steady-state performance.

A possible solution to this problem is to implement a mechanism to reset the JIT compiler on demand, dropping all the emitted code and collected performance metrics. Another alternative is to implement a flag to keep the JIT compiler from inlining code belonging to specific Java classes, or an anotation that prevents some methods from being inlined.[16] Both options require JVM changes. However, the changes required are minimal and completely backwards compatible.

4.6 Discussion

This chapter presented Rubah, a system that supports DSU in regular Java applications. Rubah is a step forward from DuSTM, presented in Chapter 3, in the direction of a practical system that supports Dynamic Software Updates. Section 1.2 defines a set of goals that a practical DSU system should reach, and Section 4.1 presents a set of claims about Rubah reaches some of those goals. This section explains how this chapter supports those claims, comparing Rubah with DuSTM.

4.6.1 Flexibility

Rubah is a flexible DSU system. Rubah loads each class update as a completely separate class, without any relation to the previous version. This approach is extremely flexible. Rubah permits changing any class in an arbitrary manner, with few exceptions, whereas past approaches often limit which classes can be changed, and in what ways. For Rubah, the only classes that cannot be updated are the Java runtime classes and libraries (e.g., Java collections). *Updatable classes* can directly reference non-updatable classes but not the reverse, due to issues involving the bootstrap class path of a Java application [LB98]. Of course, library classes do not directly reference application classes, so this restriction poses no practical difficulty.

Rubah is more flexible than DuSTM. DuSTM uses a post-processor to rewrite the updatable classes so that they have the same behavior but support DSU. Rubah also uses a bytecode rewriting mechanism. An important difference, however, is that Rubah uses a modified classloader to rewrite the bytecode as it is loaded. This approach allows Rubah to rewrite *all* the bytecode, updatable and non-updatable, that a program ever loads.

As a result, Rubah supports more types of changes between versions than DuSTM. In particular, DuSTM does not support the case where non-updatable code creates updatable instances through reflection. Rubah supports this case without problems.

DuSTM also requires future updates to a class to preserve the same non-updatable root class. For instance, if class N is non-updatable and class A extends N, then all future versions of A have to extend N. Future versions of A can extend any other non-updatable class, as long as they still extend N transitively.

[16] Microsoft's C# runtime environment provides support for method annotations that disable inlining of that method.

```
1  class WebThread extends Thread {
2  }
3
4  // Launching an web thread
5  new WebThread().start();
```

Version 1.2.122

```
1  class WebThread implements Runnable {
2  }
3
4  // Launching an web thread
5  new Thread(new WebThread()).start();
```

Version 1.2.123

Figure 4.15: Program change that Rubah supports but DuSTM does not. This change happens between versions 1.2.122 and 1.2.123 of H2. Class `org.h2.server.web.WebThread` inherits from `java.lang.Thread` in version 1.2.122. It implements interface `java.lang.Runnable` in version 1.2.123.

Rubah lifts this restriction. A particular reason is that Rubah can traverse the program state and find all instances to outdated types, even if they are kept inside non-updatable classes, and transform them accordingly. For instance, consider the change that class `WebThread` underwent between H2 versions 1.2.122–1.2.123, depicted in Figure 4.15. The root non-updatable parent of class `WebThread` changed from `Thread` to `Object`. DuSTM does not support this update, but Rubah does. Rubah is thus the most flexible DSU system for Java to date.

4.6.2 Efficiency

Rubah imposes no measurable performance overhead when executing in steady-state. That is, a program that supports DSU through Rubah gets that feature for free in terms of performance overhead when not actually performing a DSU. This is an important difference that separates Rubah from DuSTM, which introduces non-trivial performance overhead of up to 50% to support DSU.

When performing a DSU, Rubah pauses the application to transform the program state to a version that is compatible with the new code. Rubah provides two algorithms to perform such transformation: A parallel one, that uses several threads to speed up the transformation and thus minimize the pause; and a lazy one, that simply transforms root references and restarts the program as soon as possible, leaving behind a trail of proxies that transform the remainder of the program state as the new version access it for the first time after the update.

The proxies that Rubah uses are very different from the proxies that DuSTM uses in the form of handles. Each proxy that Rubah uses only exists for a limited period, whereas handles in DuSTM are permanent. As a result, Rubah proxies effectively amortize the time required to migrate the program state over the execution of the updated program version. Furthermore, each proxy removes itself from the object graph after being used, effectively removing all proxy related overhead.

The experimental evaluation showed that the parallel algorithm scales well by increasing the number of threads used to transform the program state, reducing the pause time. It also shows that the lazy transformation requires a constant pause, independent of the total size of the heap, confirming the similar results obtained for DuSTM.

Existing DSU systems for Java either support lazy program state transformation (JDrums [RA00], DVM [MPG+00]) or do not impose any performance overhead on steady-state execution (Jvolve [SHM09], DCE-VM [WWS10]); bot not both. I argue that the ability to perform lazy program state transformation is key for an efficient DSU system. Together with no performance overhead on steady-state execution, these features make Rubah the most efficient DSU system for Java to date.

4.6.3 Effectiveness

Rubah targets a popular language — Java — and is implemented without requiring a custom compiler or a custom JVM. Rubah is directly applicable to programs that have a typical control-flow structure where each thread executes a long-running loop that reads requests from clients, processes them, and sends back the response.

Rubah can be used on a broader class of application than DuSTM, which requires updatable applications to be transactional. Even though Rubah still requires a particular structure on the updatable applications, the experimental evaluation described how to adapt five existing applications to use Rubah.

Even though Rubah does not require a custom JVM, its implementation is tightly coupled with the particular JVM it uses. DuSTM puts much less restrictions on the underlying JVM and can be readily used on a different JVM. Rubah needs to be adapted before it works on a different JVM.

However, Rubah is a step forward in effectiveness when compared to other DSU systems with comparable performance, JVolve [SHM09] and the DCE-VM [WWS10]. These two systems are implemented *inside* the JVM through considerable modifications to the garbage-collector. As such, these two systems are tied to a specific garbage collection implementation. Porting these systems to a different JVM effectively means rewriting them almost from scratch. Rubah is much easier to port to a different JVM that provides an escape hatch to perform low-level memory access inside the JVM addressing space, being able to take immediate advantage of future improvements in garbage collectors. Even though Rubah's prototype is implemented for the Oracle HotSpot JVM, other popular JVMs provide the required escape hatches. For instance, OpenJDK supports `sun.misc.Unsafe` and IBM Jikes has a similar class `VM-Magic`. It is possible to port Rubah to these systems by rewriting a small portion of it, and without making any change to the JVM.

JDrums [RA00] and DVM [MPG+00] require a special JVM, not just a custom garbage-collection algorithm. These systems are thus impossible to port to another JVM. JavAdaptor [PGS+11] and JRebel [KV12] require HotSwap [Dmi01, Orab], which is a debugging feature available in modern standard JVMs. Although these systems are directly applicable to JVMs that provide HotSwap with zero porting effort, they are development time tools that cannot be used in production, which limits their effectiveness.

Therefore, Rubah is the most effective DSU system for Java to date.

4.6.4 Correctness

When performing an update, Rubah needs to perform three tasks: (1) Stop the program safely, (2) transform the program state, so that it is compatible with the new code, and (3) migrate the control-flow to start executing the program in the new version.

All these tasks require manual annotations, which take the form of update points for task 1, update classes for task 2, and control-flow migration for task 3. On top of that, the developer needs to retrofit the application to ensure that it stops safely after reaching an update point.

Rubah automates the state transformation to a great extent. It compares two program versions and generates a stub update class that highlights what changed between versions. The developer has to initialize all the fields that Rubah could not match between versions.

All other DSU systems for Java rely on timing restrictions to detect correct update points. DUSC [ORH02], JDrums [RA00], DVM [MPG+00], and JVolve [SHM09] require active code to be quiescent at the time of an update. Unfortunately, as I discuss in Section 2.1.1, quiescence can lead to false positives, which in turn lead to update-induced crashes. The remaining DSU systems for Java (HotSwap [Dmi01, Orab], JRebel [KV12], JavAdaptor [PGS+11], and the DCE-VM [WWS10]) are development time tools that perform updates as soon as they are ready without any restriction, which is clearly an unsafe choice that may lead to incorrect updates.

JVolve goes one step further and allows developers to list program points at which the update cannot happen. Rubah requires the developer to manually specify which program points allow updates to take place. Whilst Rubah still allows the developer to list incorrect update points, the ability to manually identify those points is a key step to correct updates.

Errors triggered during an update can crash the application, make it hang forever, or otherwise deviate from its expected behavior. These errors defeat the whole purpose of DSU, which is to improve the availability of running software. Worse still, these errors are hard to detect because they might not happen on either version in isolation, just when the old version is updated to the new one.

Without any assurance of correctness, no DSU system can ever be claimed as practical. The next chapter presents Tedsuto, a testing framework specifically designed to test the update process systematically, thus detecting and reproducing errors introduced by the update itself. Together with Tedsuto, Rubah is the first practical system that supports Dynamic Software Updates.

Chapter 5

Correct Updates

The main goal of Dynamic Software Updating (DSU) is to improve availability by removing the downtime required to update a program by stopping it. DSU is not a panacea, of course; it must be done with care. The program code assumes the execution state adheres to a certain format and invariants. Changing the code at run-time requires corresponding changes to the state that the program keeps, so that the new program state is compatible with the new program code.

It is often the responsability of the programmer to define how to transform the program state between successive versions. One particular challenge is *timing*: The transformation code may assume certain invariants, and these must hold under all the circumstances in which an update might take place. There is also the challenge that the semantics of the program may change due to the update, and this change may only make sense at certain points in execution.

Manual intervention in the dynamic update process improves the flexibility of the update, i.e. what type of program changes can be performed dynamically. But it has the possibility of error. Performing an erroneous update may cause the updated program to crash, hang, silently corrupt its state, or otherwise misbehave. This defeats the main goal of DSU. As such, we need a reliable way to *test* that dynamic updates are correct before we attempt to deploy them on live systems.

In this chapter, I present *Tedsuto*,[1] a framework for systematic testing of DSU. To use Tedsuto, the programmer annotates existing *system tests*, which check the end-to-end behavior of a program. Tedsuto uses the annotated tests to assert the correctness of an update by checking if the program's behavior before, during, and after an update makes sense. The basic idea is to systematically repeat each system test automatically, performing updates at different points in each re-execution. Tedsuto is implemented for the Rubah Java DSU system (presented on the previous Chapter), but can be adapted to other state-of-the-art DSU systems, such as DuSTM (presented on Chapter 3), Kitsune [HSD+12], JVolve [SHM09], or the DCE-VM [WWS10].

This chapter is structured as follows: Section 5.1 lists all of the contributions that Tedsuto makes in the form of claims that shall be supported by the remainder of the chapter. Section 5.2 explains how updates can fail, providing some examples of failures with the respective consequences. Section 5.3 describes Tedsuto in detail, in particular its architecture and how it explores different update opportunities for each test through different testing strategies. Section 5.4 describes how a prototype implementation of Tedsuto targeting Rubah and how Tedsuto can be implemented for other state-of-the-art DSU systems. Section 5.5 describes the experimental evaluation that validates Tedsuto's contributions, lists 8 new bugs that Tedsuto found, and discusses Tedsuto's limitations and general applicability. Finally, Section 5.6 discusses how Tedsuto reaches the goals defined in Section 5.1.

[1] The name Tedsuto is a play on the name of the University of Maryland mascot, Testudo.

5.1 Claims

Tedsuto is not a full solution for DSU so I cannot use the goals that I defined in Section 1.2 to describe Tedsuto's contributions. Regarding those goals, I can only claim that Tedsuto can be implemented for a DSU system so that such DSU system reaches the correctness goal.

However, there are a number of claims that I can make about Tedsuto that provide a better overview of its contributions. The rest of this section describes each claim in detail. The name of each claim is a property of Tedsuto.

Portable. Tedsuto can be implemented for any DSU system. Section 5.3.1 shows Tedsuto's architecture and, in particular, the type of support that Tedsuto requires from the underlying DSU system. Section 5.4.1 describes a prototype implementation of Tedsuto for Rubah, and Section 5.4.2 describes how to implement Tedsuto for other DSU systems. I claim that Tedsuto can be implemented for other DSU state-of-the-art DSU systems.

Low Effort. Tedsuto requires the developer to adapt existing system tests. I claim that Tedsuto requires a small amount of manual effort to adapt those tests so that they can be used to check the behavior of the program in the presence of DSU. Section 5.5.2 measures the effort required to add support for Tedsuto to existing testing test suites.

Practical. Tedsuto finds bugs by executing each system test several times. Furthermore, the support that Tedsuto requires from the underlying DSU systems slows down test execution. The total time that Tedsuto needs for each system test is thus a factor of the total number of re-executions times the (increased) time per test execution. I claim that the total time per test is feasible. Sections 5.3.2–5.3.6 describe the different strategies that Tedsuto uses to repeat each system test, and thus explore different update opportunities; and Section 5.5.3 measures the overhead that Tedsuto adds to normal execution and the number of re-executions per test.

Effective. I claim that Tedsuto can find timing-sensitive update bugs, even if those bugs require several threads to trigger. Tedsuto does so by executing each system test several times and performing updates at different times during each re-execution. Sections 5.3.2–5.3.6 describe the different strategies that Tedsuto uses to repeat each system test, and Section 5.5.4 describes 8 new bugs that Tedsuto found and that were previously unknown, including bugs that are only visible on multi-threaded executions.

5.2 The Need for DSU Testing

Dynamic Software Updating changes a process in place, patching the existing code and transforming the existing in-memory execution state. This way, DSU preserves active, long-running connections (e.g., to databases, or media streaming, FTP and SSH servers), which can immediately benefit from important program updates (e.g., security fixes). It also preserves in-memory server state, which is valuable for in-memory databases, gaming servers and other systems that rely on the relatively low expense and high performance of commodity RAM to maintain large in-memory datasets.

```
1  Object global;
2
3  m() {
4    process();
5
6    cleanup();
7  }
8
9  process() {
10   ...
11
12 }
13
14 cleanup() {
15   global = new Object();
16   ...
17   global.hashCode();
18 }
```

(a) Version 0.

```
1
2
3
4
5
6
7
8
9  process() {
10   global = new Object();
11   ...
12 }
13
14 cleanup() {
15
16   ...
17   global.hashCode();
18 }
```

(b) Version 1.

Figure 5.1: Example illustrating why the timing at which an update takes place can influence the correctness of the update. The left-hand side shows an initial program in Java-like pseudo-code. Method `cleanup` initializes and uses an object referred to by a global reference (e.g. static field) `global`. The right-hand side shows all the code that an update modified: The initialization code of variable `global` moved to method `process`, from line 15 to line 10. Both methods are quiescent at line 5. However, performing an update at this point results in executing the initial version of method `process` and the updated version of method `cleanup`, which in turn crashes at line 17 when the new version access `global` without either version initializing it.

Whilst DSU is an important tool to improve the availability of existing programs providing important and infrastructural services, it is not a panacea. DSU must be done with care. Program code assumes the execution state adheres to a certain format, so changing the code at run-time requires corresponding changes to the execution state, both *control* (i.e., thread call stacks and program counters), and *data* (i.e., contents and format of heap objects). As described in Chapter 2, it is often the programmer's responsibility to define such state changes, e.g., by providing *migrations* that map between the old representation of an object and a new one, and between an old control state and a new one.

One challenge with writing migrations is *timing*: The migration code may assume certain invariants that must hold for all the circumstances under which an update could take place. Mistakes in migrations or their timing can result in crashes, corruption, and other misbehavior. For instance, consider the program example introduced in Figure 5.1.[2] Version 1 moves the initialization code from line 15 to line 10. Suppose, while running version 0, we dynamically update the program after method `process` returns, on line 5. We specify no data or control migration because we observe that no data formats or running methods (i.e., `m`) have changed. However, once the program resumes, `m` will call version 1's `cleanup`, which crashes on line 17 because variable `global` was not initialized by version 0's `process`.

5.2.1 Dynamic Software Updating Failures

Dynamic software updating systems work by accomplishing two tasks. First, they *load code* into a running program to add to and/or replace existing classes, methods, etc. Second, they *migrate* the running program's *execution state* to an equivalent form that is compatible with the newly loaded code. This state consists of *data*, like linked tree and list structures that store an in-memory database, and

[2]This example was introduced originally in Figure 14, on page 14. It is repeated here for convenience.

```
1  class Session {
2    User user;
3
4  }
```

(a) Version 0.

```
1  class Session {
2    User user;
3    String userName;
4  }
```

(b) Version 1.

Figure 5.2: Example illustrating a class update. The left-hand side shows the initial version of a Java class. The right-hand side shows an update that adds field `userName`, which should be the same as field `user.name` during user authentication, i.e., `userName == user.name`. The new field is not used after connecting users authenticate themselves. This example was adapted from a real update for CrossFTP.

control, like the execution stacks of active threads. Depending on how the code changed, a migration may need to convert data representations, e.g., when the new code expects a hash table where the old expected a tree, and control states, e.g., when the new code refactors an old function into two.

As Chapter 2 discusses in detail, data and control migrations are typically specified by the programmer, perhaps with some automated assistance [NHSO06, HSD+12, PVH14, MB09]. For instance, in Rubah, described in detail in Chapter 4 starting on page 105, control migrations are specified indirectly by changing the way the program resumes after an update — Sections 4.2.3 and 4.2.4 — and data migrations are specified using a special "update class" — Section 4.2.5.

An important element of DSU is *timing*: The effect of a dynamic update's data and control migrations may differ depending on when the update is performed. To reduce the chances of update-related crashes, DSU systems often restrict the moments during a program's execution at which an update can take place. We call these moments *update opportunities*. Many DSU systems limit update opportunities to program points at which changed code is not *active*, i.e., not referenced from any thread's callstack [WWS10, SHM09, CYC+07, RA00]. This limitation is not safe, as explained earlier with example 5.1, because it still allows programs to crash due to the update process. Other DSU systems, including Rubah, limit opportunities to moments when each thread's execution has reached a programmer-specified point [NHSO06, HSD+12, PVH14, MB09].

Unfortunately, programmers will make mistakes when changing their program to support updating, and/or when writing data or control migrations for particular updates. These mistakes can manifest as a hang, crash, or other misbehavior when a dynamic update is applied.

Consider the example shown in Figure 5.2, adapted from the CrossFTP server.[3] The updated program reads the name of a user trying to authenticate from the field `userName`, whereas the old program used field `user.name`. A dynamic update must transform the running version's data from the old representation of class `Session` to the new; in particular, it should copy the value of `user.name` into the new `userName` field, to establish the invariant `userName == user.name`. Suppose the programmer forgets to specify such a migration. The new code will behave incorrectly when accessing field `userName`; e.g., it will crash with a null-pointer exception, deny access to valid users, or allow access to invalid users.

[3] This example was introduced earlier in Figure 4.4 on page 108. It is repeated here for convenience.

```
1  // Set-up
2  FTPClient c = new FTPClient();
3
4  // Test
5  c.sendVersion(0);
6  c.USER("user");
7  c.PASS("wrong");
8  c.PASS("right");
9  assert(c.isConnected());
10 assert(c.isLoggedIn());
11
12 // Tear-down
13 c.QUIT();
```

Figure 5.3: Example system test that checks the implementation of user authentication on an FTP server. This test sends to the server a wrong password first, followed by the correct password. An FTP server passes this test if the FTP client that the test uses ends the test connected and authenticated.

As another example, recall the update shown in Figure 5.1. Here, a control migration must be used to map the program counter of the taken update opportunity to the appropriate location in the new version. The mapping is trivial for function `f`: Lines 4, 5, and 6 in version 0 map to lines 4, 5 and 6 in version 1, respectively. For function `process`, line 10 in version 0 maps to line 11 in version 1, with the caveat that global variable `global` also needs to be (data-)migrated for the update to be correct. However, suppose the programmer specifies a mapping from line 15 in version 0 to line 10 is version 1 (as that is the line of code that moved). Doing so results in incorrect behavior because the program counter will be moved to version 1's `process` method, but the (version 0) `process` method was already executed. As a result, messages may get duplicated.

In short, the behavior of a dynamic update depends on the control and data migrations specified by the programmer, and the update opportunity at which these migrations take effect. Without care, a dynamic update may produce incorrect results.

5.2.2 Testing Dynamic Updates

How can we avoid update failures? Gupta et al. have shown that, in general, establishing that an update is correct is undecidable [GJB96]. Normal program properties are undecidable too (e.g., termination), so typical software development uses tests to ensure that a software system will behave as it should when deployed. Likewise, we can use testing to give us confidence that a dynamic update, when applied in deployment, will behave as expected. We can test the correctness of a dynamic update by running a test and checking the behaviour of the program before, during, and after an update. Given that DSU is a whole-program operation, we can perform DSU tests with *system tests*, which check the end-to-end behavior of a program. For instance, consider the test in Figure 5.3, and suppose that both the old and new versions (when run without performing an update) pass the test. It is possible that, due to bugs in data/control migrations or other program changes, dynamically updating from the old to new version during the test run will cause the test to fail. For instance, consider the update to the `Session` class of Figure 5.2, and suppose that the data migration fails to initialize the `userName` field. Dynamically updating the server on any of the update opportunities generated by lines 6 or 7 of the test will cause it to fail,[4] because these commands cause the FTP server to use the `userName` field. Updating the server at other opportunities during the test run will pass because the new code either initializes the field (when handling the request at line 5) or never refers to it again (when handling requests at lines 8 and 13).

[4]Each test-server interaction may generate several update opportunities.

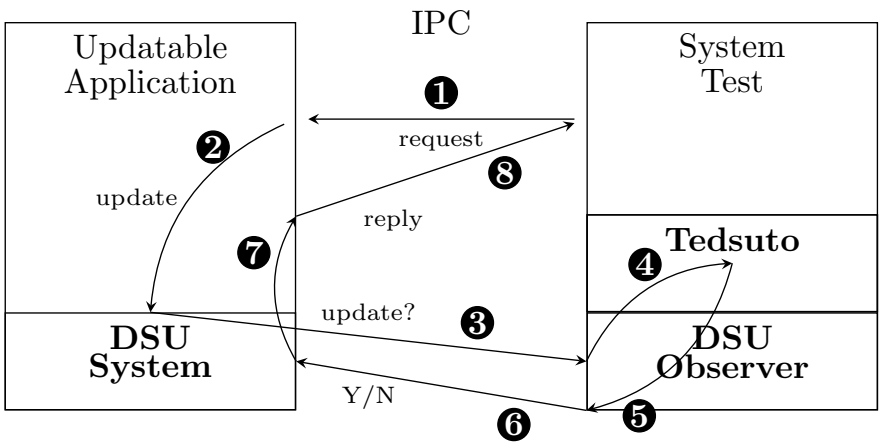


Figure 5.4: Architecture of Tedsuto. Straight arrows (**1**, **3**, **6**, and **8**) mean inter-process communication (IPC) between the test and the updatable application. Curved arrows (**2**, **4**, **5**, and **7**) mean control-flow transfer through method calls within each process. Following the arrows, we can understand the sequence of calls that Tedsuto makes when processing each request that the system test makes.

In summary, it is possible that a test passes when the target application is updated at some update opportunities, but not others. As such, a testing strategy for dynamic updates should explore the behavior of different update opportunities systematically. This idea provides the basis for the design of Tedsuto.

5.3 Tedsuto

Tedsuto is a framework for systematic testing of Dynamic Software Updates. Tedsuto repeats each system test automatically, performing an update at different points in each re-execution. Tedsuto easily tests backwards-compatible behaviors, i.e. those externally visible behaviors that the update does not change, and new behaviors, e.g. that a bug is fixed correctly or that a newly added feature functions properly. It is thus a complete solution for testing DSU. Even though I implemented Tedsuto using Rubah, the basic concepts can be adapted to other DSU systems such as DuSTM, described in Chapter 3; Kitsune [HSD+12]; JVolve [SHM09]; or the DCE-VM [WWS10]. This section discusses the architecture of Tedsuto and the types of testing methods it enables.

5.3.1 Architecture

Figure 5.4 shows the architecture of Tedsuto. The updatable application and the system test run in separate processes that communicate through Inter-Process Communication (IPC). During its execution, the system test will interact with the updatable application, e.g., by sending it service requests (1). While processing each request, the execution of the updatable application triggers several update opportunities (2). Tedsuto assumes that the DSU system generates a small number of update opportunities per interaction. This may not be true for all DSU systems; I discuss in Section 5.4.2 how Tedsuto can still be used with such systems. For now, let us assume that each interaction generates a small number of opportunities.

```
1  class Tedsuto {
2    static void allowUpdates();
3    static void disallowUpdates();
4    static void ensureUpdated();
5    static void operation(Object threadID, String label);
6  }
```

Figure 5.5: Tedsuto's API for adapting system tests. Methods `allowUpdates` and `disallowUpdates` allow Tedsuto to skip uninteresting update opportunities. Retrofitting existing system tests with them is completely optional. Method `ensureUpdated` can be used to adapt tests that only pass in the updated program version (e.g. ensure that new features or bug fixes work as expected after the update). Method `operation` identifies high-level operations, through the `label` argument, made concurrently by different threads, through the `threadID` argument. It allows Tedsuto to explore update opportunities in multithreaded tests.

A novel part of Tedsuto is that it employs an *update observer* that is queried at every update opportunity to decide whether to perform an update (3). The update observer, located in the same process as the system test, then notifies Tedsuto about the opportunity to perform an update (4). Based on the information that Tedsuto has about the system test in execution, it tells the observer whether to take the current opportunity, and what sort of update to perform (5). The observer then sends the decision back for the DSU system (6) before returning back to the updatable application (7), which in turn sends the reply back to the system test (8), which can then continue.

The remainder of this section explains the types of update tests that this architecture enables: exhaustive tests (Section 5.3.2), update-specific (Section 5.3.3), operation-oriented (Section 5.3.4), update-point synchronization (Section 5.3.5), and control-flow reboots (Section 5.3.6). They are differentiated by the decision process used at update opportunities (e.g., exhaustive or operation-specific), the kind of system test being performed (e.g., backwards-compatible or version-specific), and whether to perform a full update when the opportunity arises, or a more localized test (either an update synchronization or a control-flow reboot).

5.3.2 Exhaustive Tests

During a deterministic test, update opportunities can be matched between different re-executions by the order in which they happen. For instance, consider two executions of the test shown in Figure 5.3 on a server that performs updates only after FTP commands. Regardless of the update opportunity that Tedsuto explores, the second update opportunity will happen on line 6 in both executions. This allows Tedsuto to re-execute the test and systematically explore all update opportunities.

Of course, not all update opportunities need to be explored. In particular, in the example we are following, the update opportunities triggered while setting-up the test (lines 1–2) and tearing-down (lines 12–13) are not relevant to the test and may be skipped. Other FTP tests may require an authenticated user (e.g. to test file permissions) and move lines 5, 6, and 8 to the test set-up. Developers can adapt existing tests to skip exploring uninteresting opportunities by retrofitting the tests with Tedsuto's API, shown in Figure 5.5. In particular, they can surround lines 4–10 in Figure 5.3 with calls to `allowUpdates` and `disallowUpdates` to explore opportunities only during the main body of the test. Tedsuto automatically supports tests written in the popular JUnit framework and skips all opportunities that happen during the `setUp` and `tearDown` methods. Section 5.5 describes how I evaluated Tedsuto using two JUnit test suites without any effort to integrate them with Tedsuto's API.

```
1  AtomicInteger count;
2  int MAX; // Maximum clients allowed
3
4  // Launch several threads that do:
5  FTPClient c; // Thread-local
6  void run() {
7    while (!stop) {
8      Tedsuto.operation(c, "USER");
9      c.USER("user");
10     Tedsuto.operation(c, "PASS");
11     c.PASS("right");
12     assert(count.incAndGet() < MAX);
13     Tedsuto.operation(c, "QUIT");
14     C.QUIT();
15     count.dec();
16   }
17 }
```

Figure 5.6: Example of a multithreaded test that checks the maximum number of connected clients to an FTP server. The test fails if, at any point during its execution, there are more clients connected than the maximum. This test can be integrated with Tedsuto by adding the highlighted code to represent each high-level operation that the test performs.

5.3.3 Update-Specific Tests

So far, I have assumed that system tests should pass both program versions when not performing an update. However, this may not always be the case. For example, the new version may add support for new features and fix bugs. As such some tests for the new version may not pass on the old version. For instance, consider an example taken from version 1.0.7 (or, v_0 for short) to version 1.0.8 (v_1) of CrossFTP. When a client fails authentication in v_0, CrossFTP closes the connection at the first wrong password attempt. In v_1, CrossFTP allows the client to retry another PASS command reusing the same connection. The system test shown in Figure 5.3 passes in v_1 but fails in v_0. This is an important test because it checks that a new feature works as expected; we would like to adapt it to confirm that the dynamically updated program supports this new functionality.

Let us assume that the test generates one update opportunity per line on lines 5–8. Updating at either of the first two update opportunities, after lines 5 and 6, passes the test. Waiting for any update opportunity afterwards results in the connection being closed at line 7 and failing the test: Tedsuto performs the update too late.

Developers can state the latest point in the test execution at which an update can be performed using method `ensureUpdated` from Tedsuto's API, shown in Figure 5.5. In this case, developers would add a call to this method after line 6. I call this annotated test an *update-specific test*.

Oftentimes, update-specific tests start as existing tests written for the new version, which developers then adapt by adding annotations or small code changes. Section 5.5 reports the effort required to write 17 update-specific tests for 2 updatable applications; mostly adding one single line of code per test.

5.3.4 Operation-oriented Testing

Exhaustive tests and update-specific tests match update opportunities between different executions, which means they only apply to deterministic tests. Tedsuto supports multi-threaded and nondeterminstic tests by employing a notion of coverage that is based on the high-level operations carried out by the test. Figure 5.6 shows an example of a nondeterministic test; let us ignore the highlighted code for now. This test uses multiple threads to check whether the server respects the limit on the number of authenticated clients. The test is composed of three high-level operations: (1) Sending the USER command; (2) sending the PASS command, thus becoming authenticated; and (3) sending the QUIT command to reset

the connection for another iteration of the test. The developer annotates the test with information about the operations, exposing them to Tedsuto through method `operation` on its API. In this case, it means adding lines 8, 10, and 13, highlighted in Figure 5.6.

Method `operation` takes two arguments, as shown in Figure 5.5. The second one is simply a label that distinguishes the operation from all others. The first is some object that identifies the thread in the test. In this case, each thread uses a dedicated `FTPClient` object to interact with the server; such an object identifies each thread unambiguously. With this information, Tedsuto can reason about *combinations of opportunities* during multi-threaded testing. In this case, for 2 threads, Tedsuto can explore the following combinations: USER/USER, USER/PASS, USER/QUIT, PASS/PASS, PASS/QUIT, and QUIT/QUIT. Tedsuto can now rerun the same test repeatedly until all combinations are explored. Currently, Tedsuto does not force any particular thread scheduling. This is a direction in which Tedsuto can be expanded by future work.

Operation annotations are useful even for single threaded tests: Essentially, they suggest to Tedsuto to take one update opportunity per annotated operation, even if many more are available. This gives the tester a way to reduce the testing space, taking advantage of domain knowledge.

5.3.5 Update-point Synchronization

Most DSU systems require all threads to *synchronize* before performing an update [NHSO06, HSD+12, WWS10, SHM09, PVH14, MB09]. That is, each thread should reach some known program point (e.g., a function call, a thread/GC safe point, or a manually annotated point). When all threads reach such *update points*, the DSU system performs the update, resuming all threads afterwards.

For instance, consider the code in Figure 5.6. Consider that there are several threads executing the loop and that the FTP server internally uses a lock to implement a maximum number of clients. Consider also that one thread acquires the lock and blocks for an update. In the meantime, another thread blocks waiting for the lock. At this point, the program is in deadlock: No thread makes progress and the update never gets performed. And this happened because an update became available; regular program operation would never result in such a deadlock.

Tedsuto finds this class of update-related bugs by simply requiring all threads to synchronize at all update opportunities during a system test, just to release them immediately after. I call this technique *update-point synchronization*. It allows Tedsuto to explore all the update opportunities in a single test execution, without performing an actual update. Update-point synchronization proved very effective: It found 3 of the 8 new update-related errors that I discovered using Tedsuto.

5.3.6 Control-flow Reboots

In Section , I discussed an example of control migration, performing the update that Figure 5.1 shows while modified code is active. Some DSU systems simply forbid such updates [NHSO06, SHM09], providing a simpler update model at the cost of flexibility. More recent DSU systems—UpStare [MB09], Kitsune [HSD+12], and Rubah—overcome this limitation and provide support for updating active code. These systems require developers to migrate the control state of the program between versions, effectively mapping PC positions and stack frames, which is an error-prone proposition.

Kitsune and Rubah implement control migration by stopping all active threads and then restarting them in the new-version code. In Rubah, as I discussed in Section 4.2.4 on page 107, the control migration code is effectively embedded in the startup code of each thread. This code, as it executes, regenerates the stack by taking an alternative path, provided by the developer, to the one it would normally take during startup. An erroneous alternative startup code can leave the updated program in an incorrect control state.

It is possible to test the control-flow migration code without performing a complete update. In particular, Tedsuto can initiate a *null update* by performing an update synchronization, and then "rebooting" each thread, causing it to restart. But, instead of restarting on the new version, Tedsuto restarts the "updated" program on the same version, and thus tests this alternative startup path. If the control migration code is correct, the program will return to the state it was in before the null update. This technique requires just a single test run to reboot at all update opportunities, rather than having to run the whole test many times, once per opportunity. Control-flow reboots are very effective: I found 2 of the 8 new update-related errors using it. Section 5.5 presents all the details.

5.4 Implementation

I have implemented Tedsuto for Rubah. Chapter 4 explains Rubah in detail. In this section, I explain how the concepts of Rubah map to Tedsuto.

5.4.1 Tedsuto for Rubah

Tedsuto requires the target DSU system to provide support for an update observer, as we explained in Section 5.3.1. Then, at each update opportunity, Tedsuto interacts with the observer to decide if it should perform an update or not. In Rubah, update opportunities map directly to calls to method `Rubah.update`, I modified Rubah to redirects these calls to the observer process. The observer process performs a full update for exhaustive and update-specific tests. For update synchronization tests, the observer initiates quiescence, but then allows threads to continue in the same version without throwing an `UpdatePointException`. For control-flow reboots, the observer initiates quiescence and then restarts each thread in updating mode (without updating the code) to test that the control migration code does not incorrectly corrupt the program's state.

5.4.2 Tedsuto for other DSU Systems

Although the current implementation of Tedsuto targets Rubah, Tedsuto can be applied to other update systems with explicit update points, which are among the most practical systems yet built and evaluated. These include Kitsune [HSD+12], Ekiden [HSHF11], UpStare [MB09], and Ginseng [NHSO06]. Exhaustive tests, update-specific tests, and update synchronization tests would all apply; for Kitsune, control-flow reboots would as well.

Some systems define update points *implicitly*, rather than explicitly [WWS10,SHM09,CYC+07,RA00]. For example, updates may be permitted at any point as long as a changed function or method is not active. All testing techniques would also apply to these systems (with some engineering effort), with the exception of control-flow reboots. In particular, for tractability, Tedsuto should not test updates at all (implicit) opportunities, but at a representative set of opportunities. It is possible to compute this set without loss of effectiveness with a technique from Hayden et al. [HSH+12] which determines, through program analysis, which update opportunities would have provably the same outcome for a given update, i.e., the updated program would execute the same code. Hayden et al. found that for typical updates, the number of opportunities can be reduced by between 87% and 95%. To work with Tedsuto, DSU systems with implicit update opportunities need to be modified so that each opportunity manifests in the program, and then coordinate them with the Tedsuto observer.

5.5 Experimental Evaluation

I evaluated Tedsuto by testing two programs for which I previously added support for DSU through Rubah (as described in Section 4.5): *H2*, an SQL database; and *CrossFTP*, an FTP server. The experimental evaluation shows that Tedsuto requires *low effort* to use existing tests: I used 3 test suites, comprising a total of 446 backwards compatible tests and 17 new update-specific tests that I adapted from existing tests by changing 5% of the code on 5 files. I also used a complex benchmark for H2 as a multi-threaded system test, adapted to use Tedsuto by changing 0.5% of its code. The experimental evaluation also shows that Tedsuto is *effective*: I report 8 new update-related bugs that I found whilst evaluating Tedsuto. Finally, the experimental evaluation shows that Tedsuto is *efficient*: Update-point synchronization and control-flow reboots can be used as development time tools; exhaustive testing requires more time to complete each test suite but can still be used to ensure that a new release can be correctly deployed as a DSU. This section describes the experiments in detail.

5.5.1 Experimental Configuration

All experiments that I describe in this section were run on a machine equipped with an an Intel Xeon E31280 machine, 3.5 GHz CPU (8 logical cores, 4 physical), 16GB of RAM, with GNU/Linux Ubuntu 14.04 (kernel 3.13.0-39). All tests were conducted with Oracle's JVM version 1.7.0_79-b15 (HotSpot version 24.79-b02).

I performed the experimental evaluation using two applications previously adapted to support DSU through Rubah: H2, which is a mature, SQL DBMS written in about 40K lines of Java; and CrossFTP, which is an FTP server written in about 18K lines of Java. Section 4.5 describes in detail how each application was retrofitted with support for Rubah. As a short summary, I updated H2 from version 1.2.121 to version 1.2.123, spanning version 1.2.122; and CrossFTP from version 1.07 to version 1.11, spanning versions 1.08 and 1.09. For the experiments that this section describes, I collapsed manually all releases into a single update.

I evaluated Tedsuto with a total of 3 test suites and 1 performance benchmark. I used 2 test suites for H2: (1) The tests that ship with it, called *H2-test* (14K LoC, 39 tests); and (2) a JUnit test suite taken from another Java SQL database called *HSQLDB* [5] (15K LoC, 300 tests). I adapted a JUnit test suite for Apache's MINA FTP server [6] to use with CrossFTP, called *FTP-test* (2.7K LoC, 107 tests). Finally, I used the *TPC-C* performance benchmark (8K LoC) shipped with the DaCapo benchmark suite as a multi-threaded system test for H2, given that TPC-C verifies the invariants of the benchmark at the database level after its completion. This is the same TPC-C performance benchmark that I used to evaluate the performance of Rubah, described in Section 4.5. All values reported are the average of 3 executions.

Adapting each test suite to run with only a subset of its tests (e.g., to ignore unit tests) independently required some effort, but this is not directly related with Tedsuto.

5.5.2 Manual Effort

Backward-compatible system tests (i.e., those that pass the old and new software versions) can be used with Tedsuto with no extra effort. Update-specific tests do require some manual adjustment (e.g., to indicate the latest point at which an update can be applied). Operation-oriented testing also requires annotating the operations.

[5] http://hsqldb.org/
[6] https://mina.apache.org/ftpserver-project/index.html

Test	Program	Extracted		Modified
		LOC	Tests	LOC
Login	CrossFTP	17	2	2
MD5	CrossFTP	128	10	10
DROP admin	H2	36	1	1
SELECT	H2	32	1	1
SCOPE_ID	H2	108	3	3
Benchmark	**Program**	**Extracted**		**Modified**
		LOC	Operations	LOC
TPC-C	H2	8 231	8	42

Table 5.1: Effort required to write update-specific tests. Each test was first extracted from its original suite—I report number of tests per extracted file. TPC-C is a benchmark that performs 8 different operations, which I manually annotated.

Suite	Original time	Tedsuto time		# Opportunities
		Baseline	UP-S + CF-R	
HSQLDB	4	6	216	1 905
H2-test	12	19	2 541	23 651
FTP-test	6	13	29	257

Table 5.2: Time (seconds) required to run each test suite to completion under several testing configurations. *Baseline* means that Tedsuto does not perform any update. *UP-S* means update-point synchronization and *CF-R* means control-flow reboots.

Table 5.1 shows the effort required to extract tests that fail on the old version, and adding annotations to turn them into update-specific tests. It also tabulates the effort required to identify operations performed by the TPC-C benchmark. As we can see, the effort is very low: I modified under 0.5% of the total number of lines.

Each update-specific test checks whether a feature, introduced by the new version, works as expected after the update: *Login* checks that the same connection supports several login attempts without dropping; *MD5* checks the MD5 command; *DROP admin* checks that administrators can delete themselves; *SELECT* checks a particular idiom of select queries with a select sub-query; and *SCOPE_ID* checks IDs automatically generated by triggers.

5.5.3 Performance

Moving the decision whether to explore updates to another process requires an IPC at every update opportunity, which adds performance overhead. To measure that overhead, I first executed all test suites to completion without Tedsuto. Then I computed a *baseline* overhead which uses Tedsuto, but without performing any updates. I also measured the overhead when performing an update synchronization and control-flow update at every possible update opportunity during a test run. Table 5.2 shows the results together with the number of update opportunities per test suite.

Table 5.3 reports the time to perform exhaustive testing on the HSQLDB and FTP-test suites. For these runs, I only performed updates at opportunities that occurred after the set-up phase, and before the start of the clean-up phase for each JUnit test. The table shows how many total opportunities were explored, how many were avoided, and the overall time to complete each suite.

Suite	Time	# Opportunities	
		Total	Explored
HSQLDB	16 374	77 521	2 431
FTP-test	1 654	696	408

Table 5.3: Time (seconds) required to exhaustively test each JUnit test suite and number of update opportunities generated and explored. Each explored opportunity requires an individual test run.

The H2-test suite generates an enormous number of update opportunities and has multi-threaded tests. I thus did not perform exhaustive testing with it. I did, however, perform operation-oriented testing with the H2-test suite. Operation-oriented testing can be configured with a budget of update opportunities to explore per test, which allows developers to control how long tests take. However, I discovered that the other testing techniques found many more bugs, more quickly, as I discuss further in Section 5.5.5.

Given a test suite with small, modular, and deterministic tests, such as HSQLDB and FTP-test, the low cost of update-point synchronization and control-flow reboots allows developers to use these two techniques to quickly find bugs during development of a new version. Exhaustive testing has a higher cost that forbids using it during development, but still low enough to be used for every version deployed as a DSU. Performing update-point synchronization and control-flow reboots can also be used with larger, non-deterministic tests, such as H2-test, with similar costs to exhaustive testing.

When performing exhaustive testing, each re-execution required restarting the target program from scratch, including launching a new JVM instance. This happens because Rubah does not support reverting updated code back to its old version. As a result, each re-execution of a test on the FTP-test suite took around 4.5 seconds; 3 of which just launching a JVM and starting the CrossFTP server; and around 1 second performing the update. The test itself took the remaining 0.5 seconds. Time ratios for the HSQLDB suite are similar.

A possible solution for this problem is to launch several tests on the same program version, interacting with the same target server; stop all at the nth opportunity; perform an update when the last test reaches the nth opportunity; allow all to complete; and repeat for the n+1th opportunity. This would require a single execution for all tests for each opportunity explored. I implemented a prototype of this idea and noticed that the overall time to perform exhaustive testing reduced drastically. However, it required extensive changes to the original test suite to avoid tests running concurrently from interacting (I never got it to work correctly). This is one way in which Tedsuto could be expanded in future work.

5.5.4 Bugs found

When developing dynamic updates for H2 and CrossFTP to use Rubah, I performed extensive manual testing and debugging. Despite this, applying Tedsuto to these programs revealed 8 new bugs, all of which would have serious consequences (discussed below) if they manifested during deployment.

Table 5.4 reports the name of each bug together with the test suite and the technique that found it most quickly. In several cases, exhaustive and operation-oriented testing found the bug, too. I discuss why each technique found each error at the end of this sub-section.

In the following, I describe each bug in detail:

Internal Data Races in Rubah Rubah had internal data races that would manifest only in rare circumstances. For instance, when launching a thread just before an update took place, that thread would not stop for the update but keep executing while Rubah performed program-state transformation. The update would eventually crash due to old code accessing transformed data. Another example is

	H2-Test	HSQLDB	FTP-test
Update-Point Sync	a), d) , e)	a)	a)
Control-Flow Reboots	b)	b)	b), c), h)
Update-Specific	—	—	g)
Exhaustive	—	f)	—

Table 5.4: Bugs found, grouped by technique and test suite. *a)* and *b)* group several bugs in Rubah; I list them multiple times.

```
1  if (Rubah.isUpdating())
2    xferredOffset = saved.xferredOffset;
3
4  Rubah.update("transfer");
5
6  while (xferredOffset != file.size()) {
7    // transfer a block and increase the offset
8    try {
9      Rubah.update("transfer");
10   } catch (UpdateRequestedException e) {
11     saved.xferredOffset = xferredOffset;
12     throw e;
13   }
14 }
```

Figure 5.7: Example adapted from CrossFTP that shows a badly placed update point on line 4. This update point should be reachable only when restarting after an update took place, yet it is possible that it initiates an update. As a result, variable `saved.xferredOffset` is not set properly, and the program fails to resume the file transfer after the update takes place.

when performing two updates in tight sequence, the second update could start before all threads finished control migration for the first update. In this case, Rubah would fail to stop all threads for the second update; the left-out threads would then crash due to accessing wrong-version data.

Resource Leak Updatable programs should use interruptible I/O so that an update can be readily applied even when the program is waiting for I/O. Rubah provides a drop-in API for interruptible Java I/O calls[7] that requires the developer to provide, and manage the lifetime of, selector objects. In Figure 4.3 (on page 106), method `readOperation` throws an `UpdateRequestedException` when interrupted by an update. This exception is caught on line 61 and the loop soon reaches the update point on line 59. When I added support for Rubah to H2 and CrossFTP, I re-opened each selector between updates without closing the one used in the previous version, i.e. I did not add line 75. After some updates, the program reached the maximum number of selectors and terminated.

Wrong Update Point When retrofitting CrossFTP with Rubah, I added support for updates to happen while transferring files. Figure 5.7 shows how. After the update, the control migration restores the offset already transferred (line 2), which was saved before the update (line 11), and then reaches an update point to complete the control migration (line 4). An update that takes place after starting the transfer but before sending any data could reach the update point on line 4 without setting the state. That update point should be guarded by line 1.

Lock Timeout H2 implements transaction isolation through row locks. When attempting to grab an already locked row, threads spin until the lock becomes available or the operation times-out. If an update happens at this point, the thread that holds the lock reaches an update point, and thus

[7]In Java, interrupting a socket operation closes the socket; Rubah's API is compatible with the socket API but can be safely interrupted for updates.

stops executing, while other threads are waiting for the lock. The other threads will eventually fail the operation after the time-out expires and only then reach an update point. This bug then manifests itself as transactions aborting due to either (1) mis-detecting concurrent modifications or (2) triggering a lock timeout, depending on the SQL statement waiting for the lock.

Exclusive Mode The H2 database supports a feature called *exclusive mode* in which a single client has exclusive access to a particular database. All other clients that try to connect to that database have to wait until the connected client exits exclusive mode. Performing an update in this setting leads to a deadlock: The threads belonging to the exclusive client reach an update point, and thus stop executing until the update starts; while the other threads keep waiting for the exclusive mode to be released without ever reaching an update point, and thus preventing the update from starting.

Function ID Transformation Internally, H2 represents prepared *CALL* statements, which invoke built-in functions on the database, using a distinct integer for each possible function. One version of H2 added a new built-in function *SCOPE_ID* to retrieve the IDs generated through database triggers for each statement. However, this new function had, in the new version, the same ID — 154 — as another function *AUTOCOMMIT* in the old version, which checks whether the auto-commit flag is set for the current session; *AUTOCOMMIT* was given ID 155 in the new version. When an update is performed after preparing a *CALL AUTOCOMMIT* statement, a prepared statement could invoke the wrong function *SCOPE_ID* after the update. I was able to observe a similar bug for function *READONLY*, that checks whether the current session is read-only, on other test case.

New FTP Command — MD5 CrossFTP adds support for the MD5/MMD5 commands in one of the versions I retrofitted. However, the server failed to detect that the command was available after the update because an in-memory map structure of available commands was not updated during the update to contain the new command.

Batch FTP commands I retrofitted CrossFTP in a way that did not support receiving FTP commands in a batch. When several commands were included in a single message, CrossFTP would process the first command and then wait for more commands from the client, instead of checking if the received message had any commands left. The original code used a buffered stream, which only performs a socket read when empty. The retrofitted code also uses a buffer but it always reads commands from the socket, thus missing commands left in the buffer by a previous read.

Discussion

Update-point synchronization and control-flow reboots perform a large number of updates in tight sequence, thus revealing data races on corner-cases inside the DSU system itself (a) and erroneous thread interleaving on multi-threaded tests (d and e). The sheer number of updates that these two techniques perform also reveals resource leaks (b). Control-flow reboots would find the same errors as update-point synchronization, but at a slightly higher cost. Exhaustive and operation-oriented testing performs a single update per execution and would not find these errors. Some errors are simply caused by taking a rare and erroneous update opportunity (c and h). Exhaustive testing would also find these two bugs because it explores all possible update opportunities. Finally, some errors are due to incorrect data migration between versions and would only be found by exhaustive testing (f) or update-specific testing (g), depending if the error is on modified backwards-compatible code or new features.

5.5.5 Operation-Oriented Testing in Practice

I used operation-oriented testing in the adapted TPC-C benchmark suite. Operation-oriented testing requires specifying a budget of updates to explore per combination, and which combinations to consider. I applied this technique to H2 and TPC-C, exploring 20 combinations per operation on the most common, least common, and randomly selected combinations (we measured the number of update opportunities per combination in a pre-run). This technique also discovered bug (d), but found no additional bugs. One issue is that it is fairly inefficient: updating on uncommon combinations required several re-runs until the target combination would finally happen. I conjecture that a more efficient scheduling scheme (e.g., along the lines of an explicit state model checker like CHESS [**?**]) might make operation-oriented testing more efficient and effective. This is one direction in which Tedsuto could be expanded by future work.

5.6 Discussion

This chapter presented Tedsuto, a DSU testing framework that tests the correctness of updates through systematic testing. In Section 5.1, I made a series of claims about Tedsuto. The rest of this chapter provided evidence for those claims. In this section, I summarize the evidence that supports each claim.

Tedsuto is *portable*. It can be implemented for any DSU system, given that such DSU system supports delegating the decision about whether to take an update opportunity to an external process, as described in Section 5.3.1. Even though I only implemented Tedsuto for Rubah, it is possible to also implement it for other DSU systems described in Chapter 2 and for DuSTM, described in Chapter 3. In the latter case, update opportunities would not be reaching update points, but instead accessing object kept inside transactional handles.

Tedsuto requires *low effort*. Tedsuto assumes the existence of a collection of system tests for the updatable program. System tests, in this context, test the back-to-back behavior of the whole program. Tedsuto requires the developer to annotate those tests with information about what the test is doing. I added support for Tedsuto to two system test frameworks and two performance benchmarks by modifying up to 5% of the original code, as Section 5.5.2 reports.

Tedsuto is *practical*. The numbers that Section 5.5.3 reports show that Tedsuto introduces a non-trivial amount of overhead while executing the tests, due to the number of re-executions needed for exhaustive and update-specific tests, and due to the number of update opportunities taken for update-point synchronization and control-flow reboot tests. Even with this overhead, Tedsuto is still feasible and allows tests to be executed within an acceptable amount of time. Using a test suite written for the popular JUnit framework (or, alternatively, annotating tests with methods `allowUpdate` and `disallowUpdate`) keeps the number of re-executions per test within a feasible bound,.

Tedsuto is *effective*. It found manually injected bugs, even if those bugs require multiple threads to trigger, as Section 5.5.4 reports. Furthermore, it found new, previously unknown, bugs on both applications and on Rubah itself.

Tedsuto allows the developer to use existing system tests to reason about update correctness. With Tedsuto, the developer can write tests that check both backwards-compatibility of features that an update does not change, and the correct behavior of features that an update modifies/introduces.

As a systematic testing technique, the level of assurance that Tedsuto provides has limits. Tedsuto can only find bugs on the code and program states that the system tests execute. The results that the Section 5.5 reports, however, show that Tedsuto performs well and can effectively find update-related bugs. Together with Tedsuto, Rubah reaches the goal of *correctness* that I introduced in Section 1.2.2.

Chapter 6

Conclusion

In this dissertation, I show that it is possible to design and implement a *practical* solution to solve the problem of *Dynamic Software Updating (DSU)*. To demonstrate this thesis, I have described the design and implementation of two DSU systems — DuSTM and Rubah — and a systematic testing framework for DSU — Tedsuto — that reach all the goals for a practical DSU system:

- **Effectiveness.** Both DSU systems that I present in this dissertation target programs written in the Java programming language. DuSTM requires Java programs to be written in a transactional style; Rubah requires Java programs to be structured around a long-running event-processing loop, which is very common in server programs. Neither system limits the development-time tools that developers can use while writing updatable programs.

- **Flexibility.** Both DSU systems that I present in this dissertation allow almost any change of a Java program between two versions. Both systems support changing the structure of existing classes without any limitation. DuSTM supports changing the class hierarchy with some restrictions and Rubah supports changing the class hierarchy without any restriction, as long the updated program is still type-safe according to Java's type system and the developer can write state transformation logic that migrates the program-state to a version that is compatible with the new program.

- **Efficiency.** Both DSU systems that I present in this dissertation minimize the pause in execution required to perform an update by migrating the program state lazily, i.e., object by object as the natural control-flow of the updated program reaches it for the first time after the update. Rubah imposes negligible steady-state performance overhead, i.e., while the program is executing normally and not performing an update.

- **Correctness.** In this dissertation, I present a systematic testing framework — *Tedsuto*. Tedsuto allows developers to reuse system tests and write new ones, to ensure that the updated program behaves as expected. This dissertation describes how to implement Tedsuto for Rubah.

Throughout this dissertation, I support these claims by describing the design, implementation, and performance profile of DuSTM, Rubah, and Tedsuto. In addition, I used Rubah to add DSU support to 13 versions of 5 existing Java programs, originally developed without any type of support for DSU. I use Tedsuto to ensure the correctness of 2 of these programs using a collection of existing system tests for each program.

This work represents an important advance in the state of the art. In particular, no existing DSU system can be considered practical by the goals that I set in this dissertation. These goals were chosen carefully to ensure that state-of-the-art DSU systems can be readily applied to a large number of existing programs. Rubah is the first of such practical DSU systems. Moreover, Tedsuto allows developers to test the process of performing a DSU in the same way they test the end-to-end correctness of their system, ensuring that the updated parts of the program behave as expected and the non-updated parts of the program behave as before, regardless of the timing when the update was performed.

6.1 Contributions

1. A technique to atomically update a running transactional program without stopping it, using a versioned STM to expose a simple update semantics to the developer;

2. A semantics-preserving transformation to add support for DSU to the bytecode of a transactional program that does not require a custom JVM to execute the transformed DSU-enabled program;

3. The design, implementation, and evaluation of DuSTM, an update system that supports a wide range of modifications between successive program versions and that can migrate the program state between versions atomically using modular program-state transformation code;

4. A semantics-preserving transformation to add support for DSU to existing Java programs that imposes no steady-state overhead while executing on steady-state;

5. Two algorithms to transform the program-state while performing an update: A *parallel* algorithm that uses several threads to minimize the time required to transform all the program-state; and a *lazy* algorithm that uses *proxy objects* to transform each outdated object as late as possible after the update takes place and thus minimizes the pause required to start executing the new program after an update;

6. The design, implementation, and evaluation of Rubah, an update system that: Supports release-level updates to Java programs not originally designed with DSU support, does not require a custom JVM, and provides good performance while executing the program in steady-state and while performing an update;

7. Three techniques to systematically test the behavior of a program undergoing DSU: Control-flow reboots, intensive update testing, and extensive update testing. All techniques re-run system tests to explore updating at different timings and thus find errors or provide empirical guarantees about their absence; each technique focuses on a different type of update error.

8. The design, implementation, and evaluation of Tedsuto, a complete solution to testing DSU that can check both backwards-compatible behaviors, i.e., those externally visible behaviors that the update does not change; and new behavior, i.e., that a bug is fixed correctly or that a new feature works as expected.

6.2 Future Work

The work presented in this document motivates a number of directions for future work. In particular, the following topics represent important unanswered questions that can further improve the state-of-the-art on DSU in the direction that this dissertation points:

- **DuSTM performance.** Improve the steady-state performance of DuSTM. Steady-state performance is the biggest problem with DuSTM; 50% overhead on steady-state is not an acceptable cost for the ability to perform DSU. One option is to explore the feasibility of using *on-demand proxies*, similar to the proxies that Rubah uses on the lazy state transformation algorithm;
- **DuSTM portability.** Research if it is possible to port DuSTM to other STM models besides the multi-versioned STM currently used;
- **Rubah portability.** Port Rubah to other JVM and refactor it to contain all implementation-related parts in a separate, pluggable, module;
- **Java unsafe API.** Refactor the current JVM unsafe features that Rubah uses into the Java API to make them safe, usable by future DSU systems, and to make Rubah easier to port to other JVMs;
- **Tedsuto portability.** Implement Tedsuto for other Java DSU systems (DuSTM, DCE-VM, JVolve) and other non-Java DSU systems (Kitsune, UpStare, Ginseng);
- **Static DSU optimization.** Use an offline conservative static analysis on both versions of the program, the current in execution and its new version, to avoid traversing portions of the object graph that are guaranteed to not have any outdated object or any reference to an outdated object. This improves the performance of both state transformation algorithms that Rubah uses;
- **Usability.** Perform a controlled study using programmers developing and then updating an application using DuSTM, Rubah, and Tedsuto to assess how easy to use they are and improve their overall usability.
- **Tedsuto scope.** Research how Tedsuto can be applied to other fields, e.g. testing checkpoint/restore algorithms, deterministic replay systems, testing DSU in distributed systems;
- **DuSTM/Rubah scope.** Research how DuSTM and Rubah can improve the state-of-the-art of DSU for distributed systems.

Appendices

Appendix A

Transactional Memory

The purpose of this appendix is to give an overview of the design and implementation of Transactional Memory in general and JVSTM in particular so that the reader can follow the discussion about DuSTM on Chapter 3. Throughout this appendix, I re-use the same notation that I introduced in Section 3.2. This appendix is based on the introductory chapters of the book by Harris et al. on Transactional Memory [HLR10] which describes the state of the art on the topic in greater detail.

A.1 Concurrent Application Example

To help guiding the discussion about Transactional Memory, this section introduces a simple running example of a concurrent application. Let us consider a simply linked ordered list of integers that supports adding new elements, removing the lowest element, and computing the sum of all the elements on the list. For the sake of simplicity, I shall refer to the list that contains elements 0, 1, and 2 using the notation (0, 1, 2).

Before reasoning about concurrency, let us consider a list that implements its specification for sequential executions. In the following, I define the semantics of the sequential list in terms of pre and post conditions of each operation:

add n

> **Pre-condition** List is (A, i, B) such that $n <= i$, and A and B denote zero or more elements a and b such that $a < n$ and $b > n$, respectively;
>
> **Post-condition** List is (A, n, i, B).

remove

> **Pre-condition** List is $(i, rest)$, where $rest$ denotes zero or more elements a such that $a >= i$;
>
> **Post-condition** List is $(rest)$;
>
> **Result** i

sum

> **Pre-condition** List is $(a_1, \ldots, a_n)$
>
> **Post-condition** List is $(a_1, \ldots, a_n)$
>
> **Result** $a_1 + \cdots + a_n$

```
1  class SeqSortedList {
2
3      final Node head = new Node(Integer.MIN_VALUE);
4
5      void add(int el) {
6          Node n = new Node(el);
7
8          Node p = head;
9
10         while (p.next != null && p.element < el)
11             p = p.next;
12
13         n.next = p.next;
14         p.next = n;
15     }
16
17     int remove() {
18         Node n = head.next;
19         if (n == null) throw new NoSuchElementException();
20         head.next = n.next;
21         return n.element;
22     }
23
24     int sum() {
25         int sum = 0;
26
27         for (Node p = head.next ; p != null ; p = p.next)
28             sum += p.element;
29
30         return sum;
31     }
32 }
33
34 class Node {
35     final int element;
36     Node next;
37
38     Node(int el) {
39         element = el;
40     }
41 }
42
43 class NoSuchElementException extends RuntimeException { }
```

Figure A.1: Possible implementation of a sequential sorted simply linked list.

Figure A.1 shows a possible implementation for the sorted list in Java that is not not *thread safe*. The lack of thread safety means that, in the presence of multiple concurrently executing threads, the interleaving of the various threads can leave the list in a state that is inconsistent with the operations that each thread performed on the list. For instance, consider the list (0, 20). Consider also two concurrent threads that add 5 and 6. Figure A.2 shows a possible interleaving that leads to the resulting list (0, 5, 20) instead of the expected (0, 5, 6, 20).

In Java, we can turn the sequential list defined in Figure A.1 into a thread-safe list by adding the modifier `synchronized` to all methods that read or modify the list. When a thread executes a synchronized method on a particular object, it acquires the monitor of that object before starting to execute the method and releases it just before returning. While that thread is executing the method, and thus holding the monitor, no other thread is able to acquire the same monitor. As a result, the execution of a synchronized method on one object by one thread cannot be interleaved with the execution of another synchronized method on the same object by a different thread. Figure A.3 shows an alternative thread-safe list that is implemented using synchronized list methods.

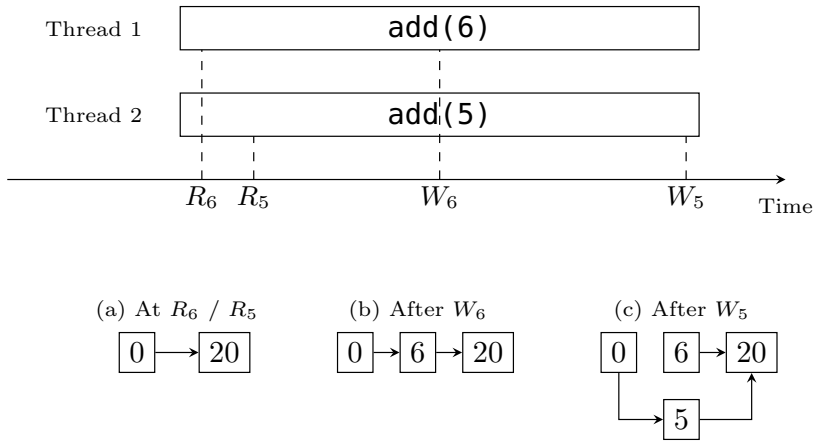


Figure A.2: Thread interleaving that results in an inconsistent execution of the code that Figure A.1 shows. The top half a possible concurrent execution of two threads over time. Thread 1 executes lines 8 and 14 at instants R_6 and W_6, respectively; thread 2 executes the same lines at instants R_5 and W_5. The bottom half shows how the list looks after each time instant.

```
1  class SyncList extends SeqSortedList {
2      synchronized void add(int el) {
3          super.add(el);
4      }
5
6      synchronized int remove() {
7          return super.remove();
8      }
9
10     synchronized int sum() {
11         return super.sum();
12     }
13 }
```

Figure A.3: Implementation of a thread-safe sorted simply linked list. The super class `SeqSortedList` is defined in Figure A.1.

By requiring all list methods to acquire a monitor during their execution, the code that Figure A.3 shows effectively rules out any thread interleaving that leaves the list in an inconsistent state. This list implementation is this thread-safe. For instance, if we consider the interleaving shown in Figure A.2 that left the sequential list in an inconsistent state, Figure A.4 shows how that same interleaving leaves the synchronized list in a consistent state.

Java's `synchronized` modifier can turn sequential objects into a thread-safe objects, but that comes with a cost. In particular, the synchronized list does not support any concurrent operation. For instance, consider list $(0, 20)$ and concurrent operations `add(5)` and `add(30)`. The synchronized list only executes one operation at the time, even though these two operations change different parts of the list and could make progress in parallel while still leaving the list in a consistent state. Worse still, the synchronized list does not allow methods `add` and `sum` to progress in parallel, or even concurrent `sum` methods over the same list.

Suppose that we implement the sorted list using a more sophisticated locking strategy that allows methods to make progress in parallel. Let's call this implementation a *fine-grained list*, hinting that methods should lock each element at a time instead of locking the list as a whole. Unfortunately, even the fine-grained list is not without problems. In particular, it is not *composable*. Suppose that we want to implement an operation `swap` that takes two lists as argument and swaps the first elements of the two lists. Figure A.5 shows a straightforward implementation that works for sequential code.

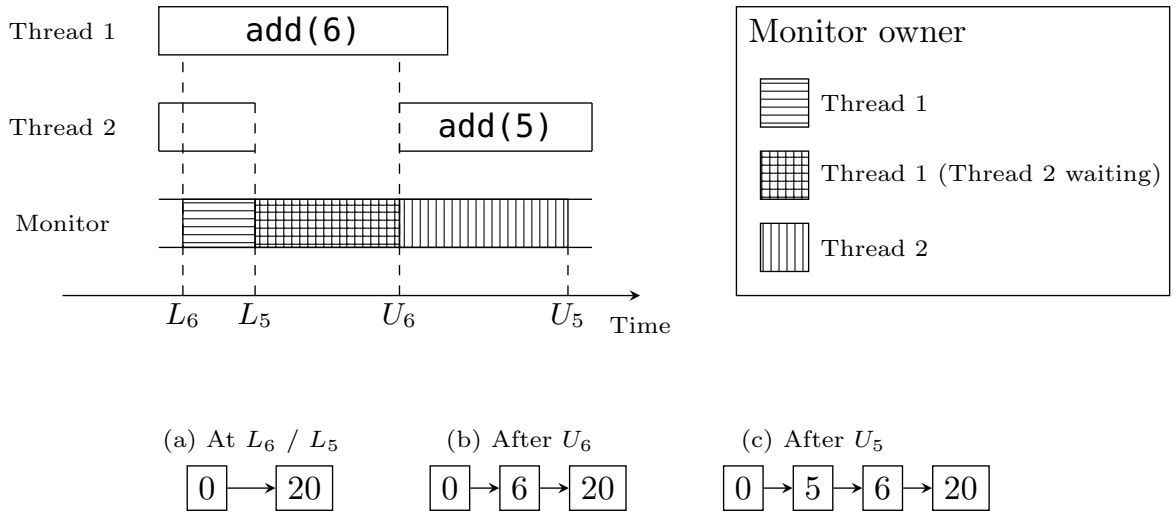


Figure A.4: Thread interleaving that results in a consistent execution of the code that Figure A.3 shows. The top part shows how thread operations interleave over time, together with the status of the monitor that guards the list. Time instants L_6/L_5 and U_6/U_5 refer to operations that try to acquire/release the monitor, respectively. The bottom part shows how the list looks after each time instant.

```
1  class SeqSortedListWithSwap extends SeqSortedList {
2      static void swap(SeqSortedList l1, SeqSortedList l2) {
3          int tmp = l2.remove();
4          l2.add(l1.remove());
5          l1.add(tmp);
6      }
7  }
```

Figure A.5: Implementation of method `swap`. Class `SeqSortedList` is defined in Figure A.1.

The code that Figure A.5 is not thread-safe. Suppose that method `swap` executes on lists $l1 = (1, 2)$ and $l2 = (3, 4)$ concurrently with method `sum(l2)`. The two possible results of method `sum` are either 7 or 5, because list `l2` is either (3,4) or (1,4). Figure A.6 shows a possible interleaving in which method `sum` returns 4, which is incorrect.

Figure A.7 shows a possible implementation of method `swap` using synchronized blocks. Unfortunately, this implementation is not correct. Consider an execution in which two threads T_1 and T_2 execute methods `swap(a,b)` and `swap(b,a)` in parallel. Consider also, that thread T_1 acquires the monitor for object `a` and then thread T_2 for object `b`. From this point on, each thread will try to acquire the lock that the other thread holds. Both threads are thus in a deadlock.

Any implementation of method `swap` that uses locks/monitors as a concurrency control mechanism *breaks abstraction* because it cannot compose existing simpler and correct operations `add` and `remove` to implement a larger more complex operation and still remain correct.

A.2 Transactions as Composable Concurrency Control

A memory transaction is an abstraction similar to database transactions but available at the programming language level. Database transactions ensure the consistency of data shared among several concurrent operations while preserving composability. In this context, a database transaction is a sequence of actions that appear indivisible and instantaneous to the system as a whole. A database transaction provides **A**tomicity, **C**onsistency, **I**solation, and **D**urability. This set of properties is typically referred to as *ACID*. Atomic transactions take place as a whole, either making all constituent actions visible on one

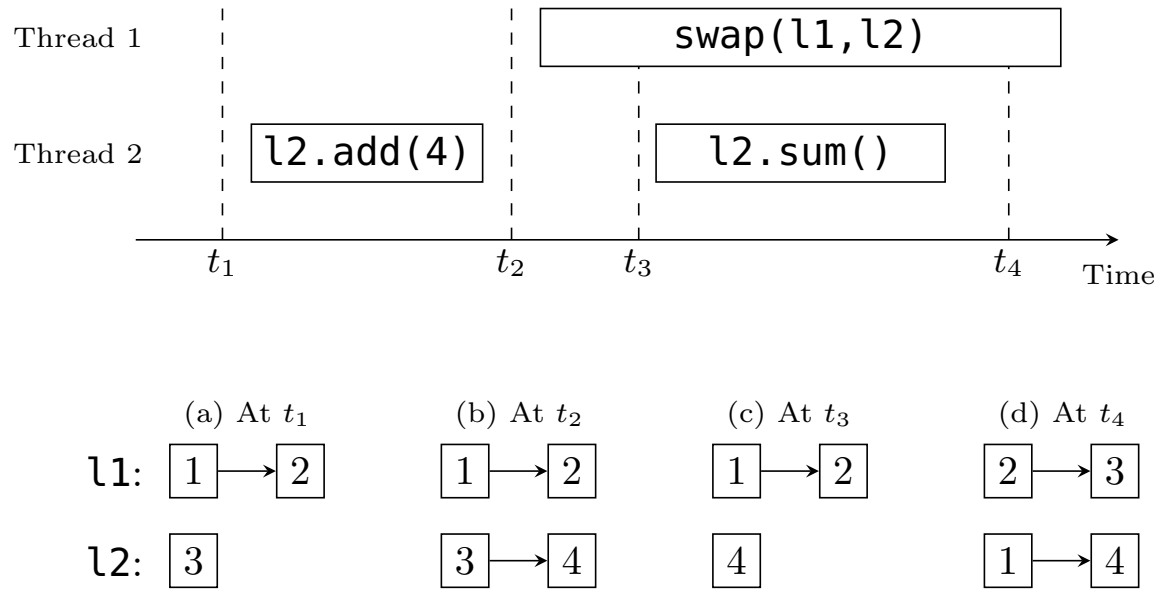


Figure A.6: Thread interleaving that results in an inconsistent execution of the code that Figure A.5 shows. The top part shows how thread operations `swap` and `sum` interleave over time. The bottom part shows how the list looks after each time instant. Time instant t_1 refers to the initial state of the lists. At t_2, thread 2 finishes executing method `l2.add(4)`. At t_3, thread 1 executes line 3. At t_4, thread 2 finishes executing method `swap`. In this execution, all the states that list `l2` takes are: (3), (3,4), and (1,4). Method `sum` sees an inconsistent state of list `l2` that never existed: (4).

```
1  class SyncListWithSwap extends SyncList {
2      static void swap(SyncList l1, SyncList l2) {
3          synchronized(l1) {
4              synchronized(l2) {
5                  int tmp = l2.remove();
6                  l2.add(l1.remove());
7                  l1.add(tmp);
8              }
9          }
10     }
11 }
```

Figure A.7: Implementation of method `swap` using synchronized blocks. Class `SyncList` is defined in Figure A.3.

indivisible step or none of them appears to have ever executed. A transaction that completes successfully *commits* and one that fails *aborts*. Consistency requires that all transactions preserve data integrity. If a transaction starts from a consistent state, it can commit only if it leaves the state consistent after executing. Isolation requires that transactions execute without ever seeing any change made by other concurrent transactions. Finally, durability requires that the changes that transactions make to the system are persisted.

Transactions provide an alternative approach for coordinating concurrent threads. All the programmer has to do is to wrap computation in memory transactions. Atomicity guarantees that the computation either completes successfully and commits its result in its entirety or aborts. Isolation ensures that every transactions produces the same result as it would if no other transactions were executing concurrently. Consistency is guaranteed by atomicity and isolation: Transactions that fail do not change the state and are completely invisible to other transactions, the only transactions that change the state are transactions that commit successfully.

```
1  class TrList extends SeqSortedList {
2      void add(int el) {
3          Transaction tx = TM.start();
4          try {
5            // Method super.add(el) is inlined for better readability of
6            // discussion in the text on this section
7            Node n = new Node();
8            n.element = el;
9
10           Node p = head;
11
12           while (p.next != null && p.element > el)
13               p = p.next;
14
15           p.next = n;
16         } finally {
17             tx.commit();
18         }
19     }
20
21     int remove() {
22         Transaction tx = TM.start();
23         try {
24           return super.remove();
25         } finally {
26             tx.commit();
27         }
28     }
29
30     int sum() {
31         Transaction tx = TM.start();
32         try {
33           return super.sum();
34         } finally {
35             tx.commit();
36         }
37     }
38
39     static void swap(TrList l1, TrList l2) {
40         Transaction tx = TM.start();
41         try {
42           int tmp = l2.remove;
43           l2.add(l1.remove());
44           l1.add(tmp);
45         } finally {
46             tx.commit();
47         }
48     }
49 }
```

Figure A.8: Transactional implementation of a thread-safe sorted simply linked list. The code is the same that Figure A.2 with a transaction wrapping the execution of each method. This code assumes a linear flat nesting model for transactions on method `swap`, as I shall define in Section A.3.5.

Figure A.8 shows how to implement a thread-safe sorted integer list, introduced in Section A.1, using transactions.

Figure A.3 takes the sequential list implementation introduced in Figure A.1 and uses the `synchronized` modifier to wrap all methods with operations that acquire and release the object monitor, thus making the code thread-safe via mutual-exclusion. We can implement a thread-safe list in a similar way but wrap the sequential code with memory transactions instead of monitor acquire/release operations. The code in Figure A.8 shows a possible implementation of a thread-safe list using memory transactions. Note that the developer effort is comparable to using the `synchronized` modifier.

Figure A.2 showed problematic execution that results in an inconsistent list. Consider that we wrap thread 1 with transaction Tx_1 and thread 2 with transaction Tx_2. It is clear that both transactions Tx_1 and Tx_2 cannot run concurrently and commit. One option is to delay one transaction until the other finishes, just like the synchronized list does on Figure A.4. Other option is to allow them to run concurrently but only allow one transaction to actually commit. The other transaction will fail and can be re-executed, this time using the list that the committed transaction generated.

Besides providing thread-safety, transactions also compose naturally. Consider the implementation of method `swap` that Figure A.8 shows. This method provides complex behavior through composing simpler methods `add` and `remove` inside a transaction. Isolation guarantees that no other transaction can ever see any intermediate state. In particular, isolation and atomicity guarantee that the execution that Figure A.2 shows, in which method `sum` sees an intermediate and inconsistent list state, is not possible.

A.3 Basic Concepts of Transactional Memory

This section provides a brief overview on the design space and the basic concepts of transactional memory.

A.3.1 Semantics and Consistency

Transactional Memory ensures that all possible ways of interleaving threads executing concurrent transactions can only result in correct executions. But what does it mean for an execution to be correct? This section answers that questions by describing the correctness property for concurrent transactions that TM provides.

- **Serializability** is the basic correctness condition for database systems. It states that the result of executing concurrent transactions must be the same as *a result* in which the transactions were executed sequentially in some order. Transactions appear to execute in isolation, as if no other transactions was executing concurrently, and they can be reordered[1] or interleaved as long as their execution remains serializable.

  Serializability does not require transactions to preserve execution order. Figure A.9 shows two possible executions of two transactions T_1 and T_2 over list (1,2,3). Transaction T_1 adds element 4 to the list, transaction T_2 computes the sum of the elements on the list. Both executions that Figure A.9 shows are correct according to serializability.

- **Strict Serializability** is a stronger correctness condition: On top of serializability, it requires that transactions preserve execution order. Strict serializability does not allow the interleaving that Figure A.9 shows on the left-hand side. Under strict serializability, the result of transaction T_2 can only be 10, as the right-hand side of Figure A.9 shows.

[1] Transactions executed by the same thread cannot be re-ordered.

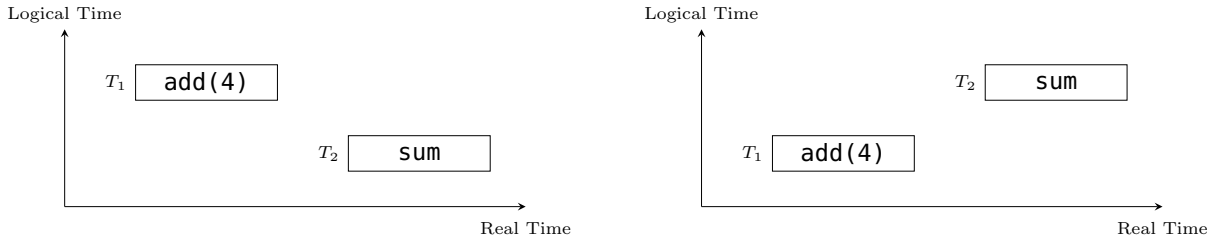


Figure A.9: Execution of two transactions that highlights the difference between serializability and strict-serializability. Both executions are correct according to serializability. On both executions, transaction T_1 executes method `add(4)` over list (1,2,3) and transaction T_2 executes method `sum` over the same list. On the left-hand side, method `sum` returns 6. Even though transaction T_2 starts after transaction T_1 commits, it does not see the changes T_1 made. This is correct according to serializability, but not according to strict serializability. On the right-hand side, method `sum` returns 10. Only this execution is correct according to strict-serializability.

- **Opacity** is even stronger than strict serializability. Strict serializability only specifies what happens to *committed* transactions. It does not say anything about what happens while a transaction is running. Can a transaction break isolation and read dirty values from other transactions, as long as it is guaranteed to abort?

  Note that transactions that execute on TM systems can interact with non-transactional code and perform irrevocable actions (e.g. sending an email). Therefore, these transactions cannot break isolation and execute on inconsistent data, even if they are guaranteed to abort at commit time.

  Guerraoui and Kapałka [GK08] proposed *opacity* as a form of strict serializability in which both running and aborted transactions must also preserve execution order. An opaque TM ensures that all transactions always read a consistent version of the world, otherwise the tentative transaction could not be part of the serial order because some work would have to appear before a conflicting update from another transaction and some work would have to appear after. Opacity is a popular correctness condition that most TMs provide.

A.3.2 Concurrency Control

Transactional Memory requires some synchronization to mediate concurrent access to data. Conflits *occur* when two transactions perform conflicting operations on the same data — either two writes or a read and a write. Conflicts are *detected* when the TM determines that the conflict has occurred. The conflict is *resolved* when the TM takes some action to avoid the conflict. These three events (conflict, detection, and resolution) can occur at different times, but not in a different order. Different concurrency control approaches differ on when each of the events happen.

- **Pessimistic concurrency control** detects and resolves conflicts at the same time they occur. With this type of concurrency control, transactions acquire exclusive control of data before accessing it and release all resources only after they finish. Figure A.10 shows an example of pessimistic concurrency control, where two transaction T_1 and T_2 attempt to add elements 5 and 6, respectively, to list (0, 20) and transaction T_2 ends up waiting for transaction T_1 to commit before being able to acquire the data it needs to make progress.

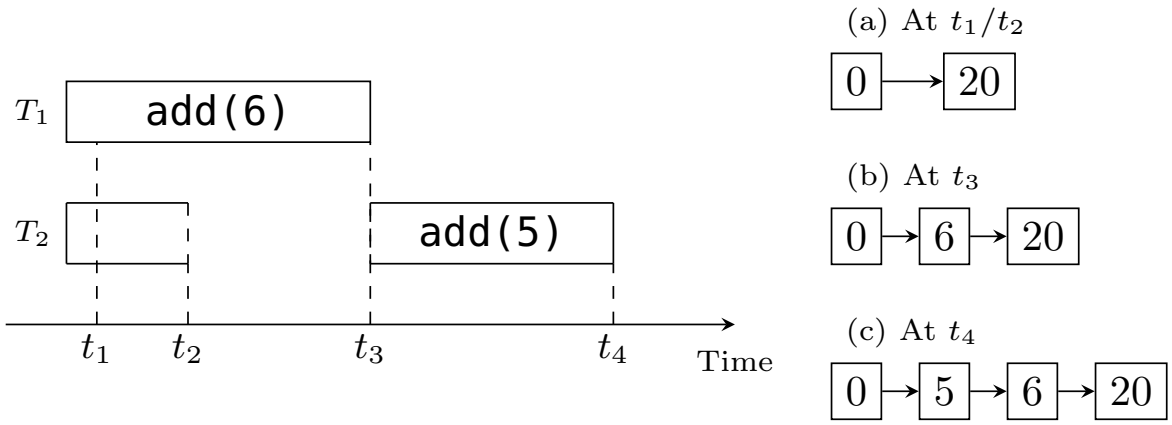


Figure A.10: Execution of two transactions using pessimistic concurrency control. Transactions T_1 and T_2 execute line 10 of Figure A.8 at instants t_1 and t_2, respectively. At that time instant, a conflict for list element 0 *occurs* between transactions T_1 and T_2. With pessimistic concurrency control, the TM *detects* and *resolves* the conflict at that same time instant, allowing T_1 to proceed and making T_2 wait for T_1 to commit.

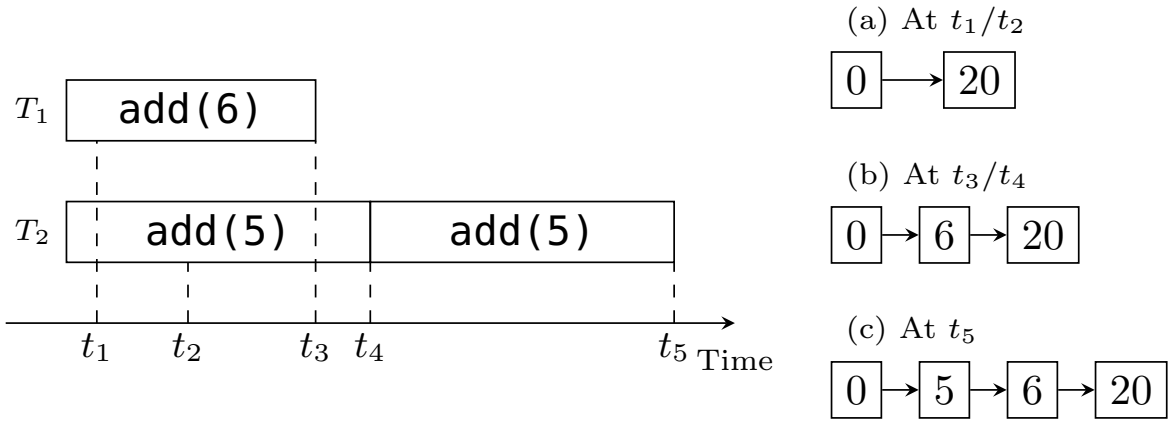


Figure A.11: Execution of two transactions using optimistic concurrency control. Transactions T_1 and T_2 execute line 10 of Figure A.8 at instants t_1 and t_2, respectively. At t_2, a conflict for list element 0 *occurs* between transactions T_1 and T_2. At t_3, transaction T_1 commits. With optimistic concurrency control, the TM *detects* and *resolves* the conflict when transaction T_2 attempts to commit at instant t_4. In this particular example, the TM rolls-back transaction T_23 and executes it again with the changes that transaction T_1 made to the list.

- **Optimistic concurrency control** detects and resolves conflicts after they occur. This type of concurrency control allows concurrent transactions to access the same data and to continue executing even if they conflict, as long as the TM detects and resolves these conflicts before a transactions commits. Figure A.11 shows an example of optimistic concurrency control for the same example described in the previous paragraph. However, instead of waiting, one of the two concurrent transactions is allowed to commit while the other is rolled-back and executed again. Note that the transaction that is rolled-back always sees a consistent state and never breaks isolation.

Both approaches require a careful implementation to ensure progress. Pessimistic concurrency control has to avoid *deadlock* — two transactions T_1 and T_2 attempt to acquire locks A and B in reverse order — and optimistic concurrency control has to avoid *livelock* — transaction T_1 forces transaction T_2 to be aborted only for transactio T_2, on restart, force transaction T_1 to abort. Pessimistic concurrency control works best when conflicts are common and optimistic concurrency control when conflicts are rare.

A.3.3 Version Management

A memory transaction performs several changes as it executes. The TM has to manage those tentative changes, isolating them from other concurrent transactions but making them visible after the transaction commits.

- **Eager version management** or *direct update* modifies the data directly in memory and keeps the overwritten values on an *undo-log* so that the TM can abort and roll-back the transaction. Eager version management requires pessimistic concurrency control because the transaction requires exclusive access to the location to write to it directly.
- **Lazy version management** or *deferred update* delays the updates until the transaction commits. Each transaction keeps its tentative writes on a *redo-log*. Reads have to consult the log so that they see earlier writes made by the same transaction. When a transaction commits, the TM copies the values from the redo-log to their actual memory locations. Aborting a transaction is as simple as discarding the redo-log.

A.3.4 Conflict Detection

The insight behing using TM for optimistic concurrency control is that the instant in which an optimistic TM detects a conflict can be different from the instant in which that conflict occurs.

- **Eager conflict detection** is when the TM detects conflicts for each transaction T when T declares its intent to access shared data or when T references shared data for the first time. At that point, the TM can check if the data that T is about to access is not stale, i.e. not overwritten by any other concurrent transaction.
- **Lazy conflict detection** is when the TM logs all accesses made during each transaction on a *read-set* and validates that the data that a transaction has read is not stale when the transaction tries to commit.

A.3.5 Nesting

A *nested transaction* is a transaction whose execution is contained by another transaction. If we assume that transactions can have at most one pending child transaction (*linear nesting*), that the inner transaction can see changes made by the outer transaction, and that there is just one thread running within each transaction; the behavior of nested transactions can follow one of the following options:

- **Flattened Nesting** propagates the abort of an inner transaction, causing the surrounding transaction to abort too. Committing the inner transaction, however, has no effect until the surrounding transaction commits.

- **Closed Nesting** allows the inner transaction to abort without terminating its surrounding transaction.

- **Open Nesting** commits inner transactions globally, making their changes visible to every other transaction in the system. This happens even if the surrounding transaction is still executing. The TM does not undo the results of committed open transactions if the surrounding transaction aborts.

A TM system can provide support for multiple nested transactions to take place in parallel inside the same surrounding transaction. This alternative to linear nesting is called *parallel nesting*.

A.4 Multiversioned Transactional Memory and the JVSTM

The *Java Versioned Software Transactional Memory* (JVSTM) [CRS06] is a software implementation of a Transactional Memory that provides optimistic concurrency control through lazy version management, lazy conflict detection, and implements a flat nesting model. JVSTM uses special memory locations to keep transactional values. These memory locations are called *Versioned Boxes*, or just *VBoxes*. A conventional memory location keeps a single value which is the last value that was written to it. A VBox is unconventional because it keeps a *history* of values that were written to it. This section explains how JVSTM uses VBoxes to ensure isolation between concurrent transactions and to provide opaque transaction semantics. Section 3.3 shall explain how to use JVSTM and its VBoxes to support DSU on transactional applications.

A.4.1 Transactional Sorted List using JVSTM

JVSTM requires the programmer wrap VBoxes around *transactional memory locations*: The memory locations that must be isolated from modifications made by concurrent transactions.

In the example of the sorted list that we have been following, introduced in Section A.1, the only field that needs to be isolated between transactions is `Node.next`.[2] Figure A.12 shows how to adapt the sequential code, shown in Figure A.2, to use VBoxes to keep field `Node.next`.

A.4.2 Transactions, Versions, and Global Clock

The left-hand side of Figure A.13 shows an example of a JVSTM transaction T that adds element 5 to list (1,10). JVSTM uses a global logical clock to serialize transactions. When a transaction starts, it reads the global clock and uses the current value as its *transaction number*. When a transaction commits, JVSTM increments the global clock and assigns the new value as the transaction *commit number*.

Each VBox keeps an history of values that transactions have written to it. In the example that we are following, transaction T adds element 5 to the list (1,10). The right-hand side of Figure A.13 shows how a list implemented using VBoxes looks before and after transaction T executes. Note that T creates a new node and adds it as a new version to the history of the VBox on field `1.next`. Also, note that all new versions that T creates — on the VBoxes on fields `1.next` and `5.next` — are tagged with the commit number of T — 4 in this case.

[2] This is highlighted by the fact that all other fields are marked as `final`.

```
1  class VBoxList {
2      final Node head;
3
4      void add(int el) {
5          Node n = new Node();
6          n.element = el;
7
8          Node p = head;
9
10         while (p.next != null && p.element > el)
11             p = p.next.get();
12
13         p.next.put(n);
14     }
15     ...
16 }
17
18 class Node {
19     final int element;
20     VBox<Node> next = new VBox<Node>();
21
22     Node(int el) {
23         element = el;
24     }
25 }
```

Figure A.12: Implementation of a sorted list using VBoxes. Field `Node.next`, declared in line 20, is wrapped by a VBox. The ellipsis represents the code for other list methods, which are similar to the code that shows except that all lines that read/write field `Node.next` are modified in the same way as lines 11/13.

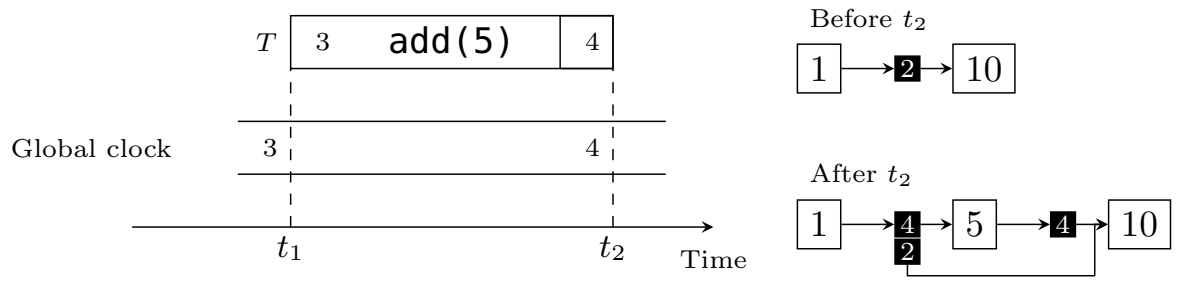


Figure A.13: Example of a JVSTM transaction. The initial list and the global clock are, at instant t_1, (1,10) and 3, respectively. VBoxes are represented as stacks of black squares, in which each square represents a version that the VBox keeps associated with the timestamp written in white letters. At t_1, field `next` on node 1 has Vbox `1.next` with just one version that refers to node 10 and has timestamp 2. The transaction executes method `add(5)` and it commits at instant t_2, adding a new version to VBox `1.next` that keeps a reference to node 5 with timestamp 4.

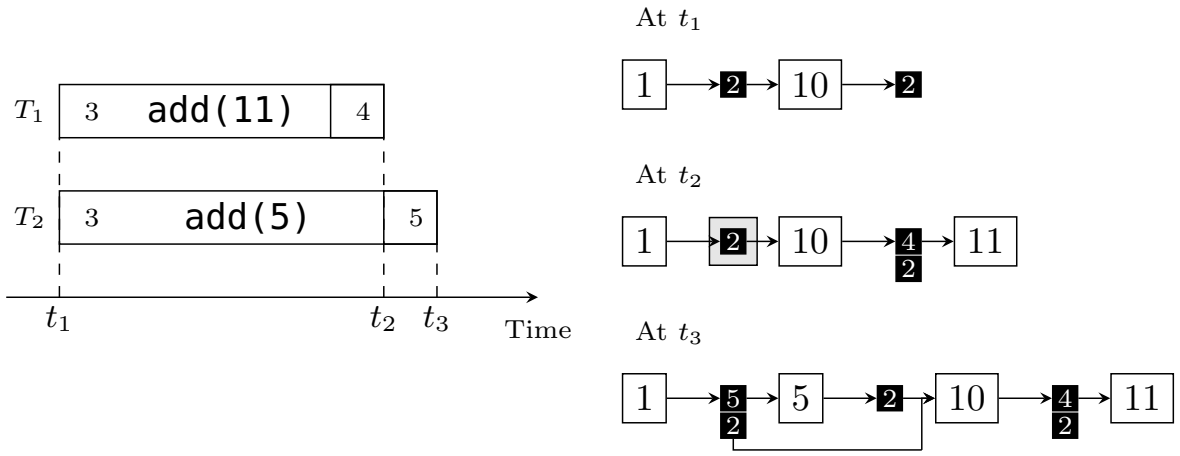


Figure A.14: Example of two JVSTM concurrent transactions that do not conflict. The initial list and the global clock are, at instant t_1, (1,10) and 3, respectively. At instant t_2, transaction T_1 commits and adds a version to VBox `10.next`, with timestamp 4 and value 11. The gray box represents the read-set of transaction T_2 — in this case just VBox `1.next`. At instant t_3, transaction T_2 is able to validate its read-set, because all VBoxes it contains have a smaller timestamp than the start timestamp of T_2, and thus T_2 commits and adds a version to VBox `1.next` with timestamp 5 and value 5.

The commit number provides a clear way to serialize transactions. It is a direct representation of a sequential ordering that is equivalent to the execution of multiple concurrent transactions. For instance, Figure A.14 shows a scenario in which two concurrent transactions T_1 and T_2 add elements 11 and 5, respectively, to list (1,10). Using their commit number, we can see that this execution is equivalent to executing T_1 and then T_2 in sequence.

Besides providing a total order between transactions, the global logical clock also allows JVSTM to detect conflicts. In addition to its transaction number, a JVSTM transaction also keeps a local set of all the VBoxes that it read: The *read-set*. JVSTM can use these three elements — the transaction number, the transaction read-set, and the current value of the global logical clock — to detect conflicts when a transaction tries to commit. For instance, let us consider a simple change to the execution that Figure A.14 so that transactions T_2 commits first and gets commit number 4 instead of 5. Figure A.15 shows such an execution. In this execution, transaction T_1 has Vbox `1.next` on its read-set. The latest version of this VBox is higher than the transaction number of T_1. This means that T_1 read stale data because another committed transaction, T_2 in this case, has modified something that T_1 read.[3]

The program code that uses JVSTM must delimit transactions. When a conflict occurs, the commit method throws a `CommitException` that the program code must handle to resolve that conflict. A possible way to resolve conflicts is to roll-back the transaction, by aborting it, and try it again. The retrying transaction will now be able to see the changes that made it roll-back. Figure A.16 shows how to adapt method `add`, shown in Figure A.12, to resolve conflicts that way. JVSTM provides an annotation `@Atomic` that performs this transformation automatically to annotated methods that already use VBoxes.

[3]This is a benign conflict because T_1 could still be allowed to commit without violating the list semantics. However this is not always the case. By detecting read-write conflicts, JVSTM ensures that the commit number of each always transaction generates a totally ordered sequential execution, at the cost of false-positive conflicts like this. It is possible to avoid benign conflicts with other JVSTM mechanisms [PC11b].

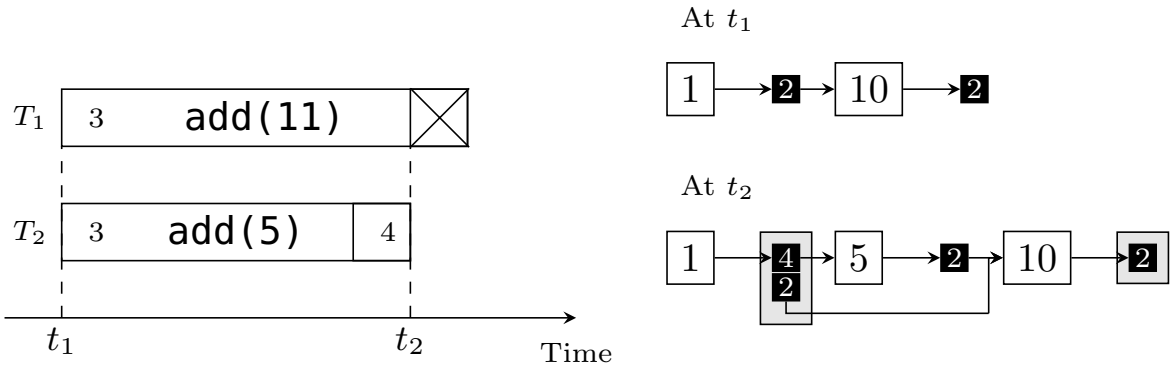


Figure A.15: Example of two JVSTM concurrent transactions that conflict. The initial list and the global clock are, at instant t_1, (1,10) and 3, respectively. At instant t_2, transaction T_2 commits and adds a version to VBox `1.next`, with timestamp 4 and value 5. The gray boxes represent the read-set of transaction T_1 — VBoxes `1.next` and `10.next`. After instant t_2, transaction T_1 is no longer able to validate its read-set, because VBox `1.next` has a version with a timestamp greater than the timestamp of T_1.

During its execution, JVSTM transactions can perform tentative writes to existing VBoxes. Those writes become permanent, and globally visible to other transactions, only when the transaction commits. While a transaction executes, JVSTM keeps its tentative writes makes in a map that is private to that transaction. This map is called the *write-map* and associates each VBox that the transaction wrote with the respective new value. This means that JVSTM has to lookup the write-map when the transaction reads a VBox so that the transaction can read its own writes. When the transaction commits, JVSTM copies all the values on the write-map as new versions to their respective VBoxes, labelled with the transaction commit number.

We can now understand all the steps that JVSTM takes to commit a transaction:

1. Acquire a global lock;
2. Validate that all the VBoxes on the transaction read-set are still valid, i.e. their latest version number is not greater than the transaction number. Abort the transaction if any box fails to validate;
3. Increment the global logical clock and assign the new value as the transaction commit number;
4. Write all the values on the write-map back to their respective VBoxes, as new versions labelled with the transaction commit number;
5. Release the global lock.

The global lock mentioned in steps 1 and 5 ensure that only one transaction can commit at the same time. This vastly simplifies validating the read-set and ensuring that each transaction gets an unique commit number. However, it is a bottleneck that limits the maximum rate of commits that JVSTM can process. There is an alternative commit algorithm [FC11] that performs steps 2–4 providing a lock-free progress guarantee.

```
1  class JVSTMList extends VBoxList {
2    void add(int el) {
3      while (true) {
4        Transaction.begin();
5        boolean txFinished = false;
6        try {
7          super.add(el);
8          Transaction.commit();
9          txFinished = true;
10         return;
11       } catch (CommitException e) {
12         Transaction.abort();
13         txFinished = true;
14       } finally {
15         if (!txFinished)
16           Transaction.abort();
17       }
18     }
19   }
20   ...
21 }
```

Figure A.16: Implementation of a sorted list with atomic operations using JVSTM. The ellipsis on line 7 represents the body of method `add` shown in Figure A.12. The exception handler on line 11 aborts the transaction if a conflict happens at commit time, on line 8. Line 16 aborts the transaction if the original body fails for any other reason. On either case, the loop on lines 3–18 keeps retrying the transaction until it eventually commits successfully, in which case line 10 breaks the loop. All other methods of can be adapted to use JVSTM by changing them to use VBoxes, as Figure A.12 shows, and then wrapping them with the code that this figure shows.

Bibliography

[ABBS05] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th Conference on USENIX Security Symposium*, SSYM, 2005.

[AK09] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys, 2009.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3), 1995.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC, 2013.

[BD93] Toby Bloom and Mark Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8, 1993.

[BFdH+13] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Jun Kato, Sean McDirmid, Michal Moskal, and Nikolai Tillmann. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, 2006.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. Object-oriented programming. chapter CLOS in Context: The Shape of the Design Space. 1993.

[BHA+05] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 Conference on USENIX Annual Technical Conference*, USENIX ATC, 2005.

[BHSS03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing Dynamic Software Updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, USE, 2003.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI, 1998.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*, 2002.

[BLS03a] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2003.

[BLS+03b] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy Modular Upgrades in Persistent Object Stores. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA, 2003.

[BPN08] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP, 2008.

[Bre01] Eric A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 2001.

[BTW07] Tim Boudreau, Jaroslav Tulach, and Geertjan Wielenga. *Rich Client Programming: Plugging into the Netbeans^{TM}Platform*. Prentice Hall, first edition, 2007.

[CCZ+06] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE, 2006.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, 1998.

[CRS06] João Cachopo and António Rito-Silva. Versioned Boxes As the Basis for Memory Transactions. *Science of Computer Programming*, 63(2), 2006.

[CYC+07] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE, 2007.

[DA99] Misha Dimitriev and Malcolm P. Atkinson. Evolutionary Data Conversion in the PJama Persistent Language. In *Proceedings of the Workshop on Object-Oriented Technology*, 1999.

[Dic01] David Dice. Implementing Fast Java Monitors with Relaxed-locks. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM, 2001.

[Dmi01] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[DN09a] Tudor Dumitraş and Priya Narasimhan. Toward Upgrades-as-a-service in Distributed Systems. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware, 2009.

[DN09b] Tudor Dumitraş and Priya Narasimhan. Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware, 2009.

[EVDB05] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande Berbers. Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, SSYM, 2005.

[Fab76] Robert S. Fabry. How to Design a System in Which Modules Can Be Changed on the Fly. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE, 1976.

[FC11] Sérgio Miguel Fernandes and João Cachopo. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2011.

[FMZ+95] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and Database Evolution in the O2 Object Database System. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB, 1995.

[Gem14] GemStone Systems, Inc. GemStone/S 64 bit programming guide, release 3.2. http://downloads.gemtalksystems.com/docs/GemStone64/3.2.x/GS64-ProgGuide-3.2/, 2014.

[GJB96] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2), 1996.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1996.

[GK08] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2008.

[GKT13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, 2007.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[GR09] Hendrik Gani and Caspar Ryan. Improving the transparency of proxy injection in java. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC, 2009.

[GRC11] Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang. Analyzing Software Updates: Should You Build a Dynamic Updating Infrastructure? In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, FASE, 2011.

[Gus03] Jens Gustavsson. A Classification of Unanticipated Runtime Software Changes in Java. In *Proceedings of the International Conference on Software Maintenance*, ICSM, 2003.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[HMH+12] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE, 2012.

[HSD+12] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2012.

[HSH+12] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating Dynamic Software Update Safety Using Systematic Testing. *IEEE Transactions on Software Engineering*, 38(6), 2012.

[HSHF11] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State Transfer for Clear and Efficient Runtime Updates. In *Proceedings of the 3rd Workshop on Hot Topics in Software Upgrades*, HotSWUp, 2011.

[HSHF12] Christopher M. Hayden, Karla Saur, Michael Hicks, and Jeffrey S. Foster. A Study of Dynamic Software Update Quiescence for Multithreaded Programs. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp, 2012.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[KM90] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11), 1990.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[KT08] Dong Kwan Kim and Eli Tilevich. Overcoming jvm hotswap constraints via binary rewriting. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, 2008.

[Kul07] Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, AOSD, 2007.

[KV12] Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the JRebel story. *Software: Practice and Experience*, 2012.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO, 2004.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, 1998.

[LDS92] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In *Proceedings of the International Workshop on Distributed Object Management*, IWDOM, 1992.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.

[MB09] Kristis Makris and Rida A. Bazzi. Immediate Multi-threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX ATC, 2009.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2009.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley, 2005.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP, 2000.

[MR07] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-quiescent Subsystems in Commodity Operating System Kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, 2007.

[MTLT08] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A Tool for Reasoning About Shape and Arithmetic. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV, 2008.

[NH09] Iulian Neamtiu and Michael Hicks. Safe and Timely Updates to Multi-threaded Programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[NHFP08] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual Effects for Version-consistent Dynamic Software Updating and Safe Concurrent Programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2008.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2006.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC, 2002.

[Obj13] Objectivity, Inc. Objectivity for java programmer's guide, release 11.2. http://support.objectivity.com/docs/objectivity/11_2_0/java/guide/html, 2013.

[Oraa] Oracle(TM). Java Platform Debugger Architecture. https://docs.oracle.com/javase/6/docs/technotes/guides/jpda/.

[Orab] Oracle(TM). Java SE 1.4 Enhancements. http://download.java.net/jdk8/docs/technotes/guides/jpda/enhancements1.4.html.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM, 2002.

[OSG14] OSGi™Alliance. *Osgi Core Release 6 Specification*. OSGi™Alliance, 2014.

[PBG13] Mathias Payer, Boris Bluntschli, and Thomas R. Gross. DynSec: On-the-fly Code Rewriting and Repair. In *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades*, HotSWUp, 2013.

[PC11a] Luís Pina and João Cachopo. DuSTM - Dynamic Software Upgrades using Software Transactional Memory. Technical Report 32/2011, 2011.

[PC11b] Luís Pina and João Cachopo. Profiling and Tuning the Performance of an STM-based Concurrent Program. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH Workshops, 2011.

[PC12] Luís Pina and João Cachopo. Atomic Dynamic Upgrades Using Software Transactional Memory. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp, 2012.

[PGS+11] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, 2011.

[PH16] Luís Pina and Michael Hicks. Tedsuto: A General Framework for Testing Dynamic Software Updates. In *Proceedings of the 9th International Conference on Software Testing, Verification and Validation*, ICST, 2016.

[PVH13] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Efficient, General-purpose Dynamic Software Updating for Java. In *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades*, HotSWUp, 2013.

[PVH14] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a Stock JVM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, 2014.

[RA00] Tobias Ritzau and Jesper Andersson. Dynamic Deployment of Java Applications. In *Java for Embedded Systems Workshop*, 2000.

[Riv96] Fred Rivard. Smalltalk: a Reflective Language. In *Proceedings of Reflection 96*, 1996.

[RSM+10] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, ICSE, 2010.

[SHB+07] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 29(4), 2007.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[SS75] Gerald Jay Sussman and Guy Lewis Steele. Scheme: An Interpreter for Extended Lambda Calculus. Technical Report AI Memo No. 349, Massachusetts Institute of Technology, December 1975.

[Ste90] Guy Steele. *Common Lisp: The Language*. Digital Press, 1990.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd edition, 2007.

[tio] TIOBE Software: Tiobe Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed July 2015.

[TS02] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP, 2002.

[VEBD06] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. An Alternative to Quiescence: Tranquility. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM, 2006.

[Ver15] Versant Corp. Versant C++ Programmer's Guide, Release 9.0. http://esd.actian.com/product/Versant_Object_Database/9/docs/Versant_Object_Database_9_Documentation, 2015.

[VF02] L. Veiga and P. Ferreira. Incremental replication for mobility support in OBIWAN. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS, 2002.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic Code Evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ, 2010.

[Zic91] Roberto Zicari. A Framework for Schema Updates In An Object-Oriented Database System. In *Proceedings of the Seventh International Conference on Data Engineering*, 1991.