Atomic Dynamic Upgrades Using Software Transactional Memory

Luís Pina and João Cachopo
INESC-ID / Instituto Superior Técnico
Lisboa, Portugal
{*luis.pina/joao.cachopo*}*@ist.utl.pt*

***Abstract*—In this paper, we propose the use of a Software Transactional Memory to implement a Dynamic Software Upgrade system that combines two desirable properties. First, it provides a simple semantics to the developer, while allowing upgrades to occur atomically, concurrently with the execution of the program. Second, it converts the program's data lazily, as data is progressively accessed by the execution of the upgraded program.**

We present also experimental results that show that our lazy approach to software upgrades is able to upgrade a system without affecting significantly the maximum response time of the system's operations, whereas an immediate approach shows values of maximum response time up to three orders of magnitude higher.

***Keywords*-dynamic software upgrades, software transactional memory, atomic upgrades**

I. Introduction

A practical dynamic upgrade system must provide means for the developer to convert the program state from the current executing program to an equivalent state compatible with the new program. Existing dynamic upgrade systems expose to the developer the exact moment when such a conversion takes place and support either *immediate upgrades*, converting all the program state when the upgrade is installed, or *lazy upgrades*, converting each portion of the program state as the natural flow of execution touches it for the first time after an upgrade is installed.

Immediate upgrades are simpler to the developer. The program is always running only one version except when converting the program state. Developers just have to address this scenario when writing conversion code.

Despite their simplicity, immediate upgrades pause the program's execution to install an upgrade and convert all the program state. A long enough pause resembles stopping and restarting a program and may thus defeat the whole purpose of a dynamic upgrade system. This is where lazy upgrades come in: The long pause is diluted over the normal program execution after installing an upgrade.

Lazy upgrade semantics, however, are more complex than immediate upgrade semantics. A lazy upgraded system runs a mixture of program versions after an upgrade, which can be confusing for the developer: What happens if the conversion code tries to access a portion of the program's state that was already converted?

In this paper, we show how to provide immediate upgrade semantics to the developer writing the conversion code whilst converting the program state lazily. We describe a novel technique that uses a Software Transactional Memory to execute the conversion code in the logical past, ensuring its correctness according to the *atomic upgrade semantics*, which we also present.

We implemented our approach in Java, resulting on a prototype called DuST'M, described in greater detail elsewhere [10], which was used to perform an experimental evaluation that compares immediate with lazy state conversion. This is, to the best of our knowledge, the first experimental evaluation that compares the two techniques. Our evaluation shows that immediate conversion introduces a pause that grows with the size of the program's state, and that can increase the maximum response time up to three orders of magnitude. On the other hand, our lazy approach keeps a constant maximum response time, regardless of the size of the program's state. However, the lazy state conversion technique introduces a steady state overhead that we measured to be between 18% and 37%.

The rest of this paper is structured as follows: In Section II, we describe the atomic upgrade semantics that we propose. In Section III, we describe how the atomic upgrade semantics can be implemented using a versioned STM. In Section IV, we show the results of an experimental performance evaluation. In Section V, we discuss the related work. Finally, in Section VI, we conclude.

II. Upgrade Semantics

Throughout the rest of this paper, we assume that the upgradable application follows the model that we describe here. We assume that the upgradable application delimits transactions around its operations. We believe that *upgrade safe-points* — points where an application can be stopped to apply an upgrade and safely resume execution on the new program version — are intrinsically related to the atomicity of operations and that, therefore, transactions are the easiest and safest way for specifying safe-points. For this reason, DuST'M ensures that every transaction finishes in the same program version in which it started, and that upgrades appear to occur atomically with respect to all other transactions executing in the system.

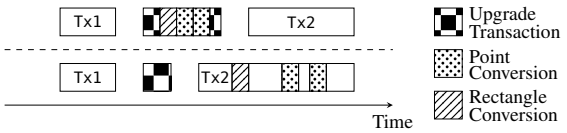

Figure 1. Immediate (top) and Lazy (bottom) Upgrade Semantics.

Moreover, we consider that the application is composed by *upgradable types* and *non-upgradable types*: Upgradable types encompass all the classes written by the program's developers, whereas non-upgradable types correspond to types belonging either to the Java platform or to third-party libraries that developers do not expect to upgrade dynamically. So, we assume that the classes implementing the upgrade system itself are non-upgradable.

To illustrate DuST'M's flexibility, consider the case where class `A` is non-upgradable, both classes `B` and `C` are upgradable, and class `A` is the superclass of `B`, which in turn is the superclass of `C`. We call class `A` the *non-upgradable root* of all of its subclasses (in this case, classes `B` and `C`). DuST'M supports any modification to classes `B` and `C` as long as they retain the same non-upgradable root. For instance, an upgrade can make class `C` the superclass of `B` provided that class `C` continues to be a subclass of `A`.

A dynamic upgrade system that supports program state migration may perform such migration eagerly, when the new upgrade is installed. This behaviour is what we call *immediate upgrades*. To illustrate immediate upgrade semantics, consider the example that Figure 1 shows on its top half. A geometric application represents points and rectangles. Each rectangle is represented as two points: The top left corner and the bottom right corner. Version 1 of the geometry application starts executing. Then, a transaction `Tx1` creates a rectangle (and the respective two points). After that, the application is upgraded, thus installing version 2. With immediate upgrade semantics, transaction `Tx2` finds instances of only version 2 of the program.

A naïve eager immediate upgrade semantics converts all the program state after installing the upgrade, pausing the application while the conversion is taking place. This may be very costly and introduce a long pause in the program's execution. An alternative is to convert the program state lazily, converting each instance only when the new version of the program attempts to access it. The upgrade system does not need to perform such conversion sooner, and is free to delay every conversion to the last possible moment.

The bottom half of Figure 1 shows the same sequence of events previously described, but on an upgrade system that converts the program state lazily. Now, each object is converted when transaction $Tx2$ first manipulates it.

Yet, lazy upgrade semantics is more complex than the immediate upgrade semantics, because, after installing an upgrade, the system may have a mixture of both old and new versions. This may lead to unexpected behaviour. For instance, consider what happens when a point is converted before the rectangle, but the conversion code of the rectangle expects both points to be in the old version. In this case we have a problem, known as the *conversion ordering problem*, because each point is now on a different program version. DuST'M solves this problem by resorting to a multi-version Software Transactional Memory system.

After an upgrade takes place, when a transaction T attempts to use an unconverted object, DuST'M pauses T and launches a special *conversion transaction* T' to convert that object. Transaction T resumes after T' finishes and accesses the converted object. Every conversion transaction T' occurs logically before the transaction T that triggers it.

In fact, even though DuST'M converts the program state lazily, the conversion code executes logically at the exact moment after the upgrade, meaning that DuST'M exposes an immediate upgrade semantics to the developer. A similar semantics was previously introduced by Boyapati et al. [2] in the context of object-oriented database upgrades.

The atomic upgrade semantics naturally solves the conversion ordering problem. Consider that transaction $Tx1$ starts immediately after an upgrade, triggers the conversion of a point, and commits. Soon after, transaction $Tx3$ starts and attempts to manipulate the rectangle, thus triggering its conversion. The conversion transaction executes at the logical past, before transaction $Tx1$ had executed. At this logical time instant, the conversion code is still able to access both points in the old program version.

III. Implementation

DuST'M [10] implements the atomic semantics for the Java programming language using a runtime library and a bytecode post-processor that introduces an extra level of indirection, thus replacing all upgradable types' references by handles. When dereferencing a handle h, DuST'M checks if the object that h keeps must be converted.

The bytecode transformation keeps the original program's semantics. Even though we do not provide any formal proof of this statement, we were able to transform two representative programs that generate a deterministic output: Sunflow, [1] a ray-tracer, and Avrora, [2] a AVR microchip simulator. Both programs generate the same output, after transforming them, on several different inputs. We omit most of DuST'M's implementation details due to space restrictions. They are available elsewhere [10].

DuST'M implements the atomic upgrade semantics by taking advantage of JVSTM's versioning [3]: It keeps the old versions of the migrated objects after converting them. JVSTM transactional locations (VBoxes) perform versioning to enforce isolation between concurrent transactions.

[1] http://sunflow.sourceforge.net/

[2] http://compilers.cs.ucla.edu/avrora/

DuST'M uses such versions to ensure that the conversion code, running in the logical past, can access every object in the expected program version. For instance, DuST'M solves the conversion ordering problem on the previous section by keeping both the old and the converted instances of the point after transaction $Tx1$ finishes. DuST'M would then ensure that transaction $Tx2$ always manipulates the new version of the point and that the conversion code always manipulates the old version of the point.

DuST'M is always able to install upgrades, even if there are transactions running code that is upgraded. In that case, those *old transactions* keep executing in the old program version. New transactions execute on the new program version. Whether old transactions will be allowed to commit when they finish is a different question.

DuST'M requires the developer to identify transactions boundaries in the upgradable application. However, it does not require the developer to use JVSTM for anything else. Yet, we must consider how conversion transactions interplay with non-upgradable types. Consider the following sequence of events: DuST'M installs an upgrade, transaction $Tx1$ writes to non-upgradable instance ω and commits, transaction $Tx2$ starts and triggers a conversion transaction T_R for a rectangle, and transaction T_R reads instance ω.

According to the atomic upgrade semantics, transaction T_R must read the value of instance ω that existed before transaction $Tx1$ executed. If instance ω is transactional, i.e. implemented using JVSTM, transaction T_R accesses the correct value of ω. However, if instance ω is not transactional, the conversion transaction accesses the most recent value of instance ω, thus violating the atomic upgrade semantics. All non-upgradable objects accessed inside conversion transactions must be either transactional or immutable.

Just adding JVSTM transactions to an existing application does not add any extra overhead to its execution: The overhead in JVSTM results from keeping the shared state consistent inside each transaction, and not from starting and stopping the transaction itself.

IV. Performance Evaluation

Immediate upgrades and lazy upgrades offer different performance trade-offs: Immediate upgrades can be implemented with virtually no steady state overhead [11], [12], but they must pause the application to convert the program state. The length of such a pause gets longer by increasing the size of the program state. Lazy upgrades, on the other hand, can install upgrades without imposing any noticeable pause, but they introduce steady state overhead.

A long pause resembles restarting the program on the new version, which defeats the purpose of a dynamic upgrade system. However, high steady state overhead seriously compromises the normal operation of the system. The decision between immediate or lazy upgrades must be carefully balanced and well informed.

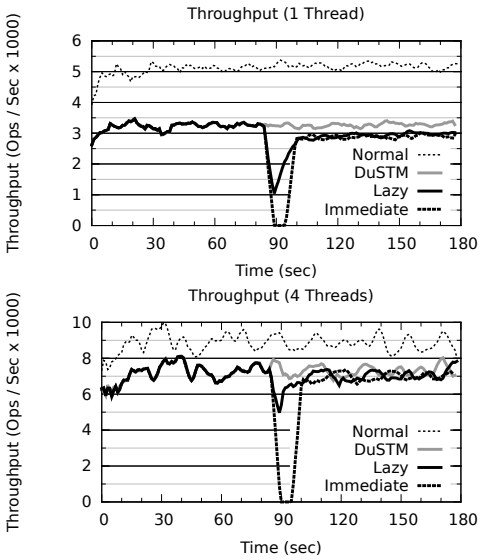

Figure 2. Throughput tracing at each second. The line *Normal* refers to the benchmark without being post-processed by DuST'M, line *DuST'M* refers to the post-processed benchmark without installing any upgrade, lines *lazy* and *immediate* refer to installing one upgrade at 90 seconds and using lazy or immediate upgrade semantics, respectively.

To that end, we evaluated the overhead that DuST'M introduces on STMBench7 [5]. In STMBench7, it is possible to reach every instance starting from a single global static reference. This feature allowed us to compare DuST'M results against immediate upgrade systems: We implemented an immediate upgrade mode in which DuST'M suspends all threads after installing a new program version, converts all STMBench7 instances, and, after converting all instances, resumes STMBench7's execution.

STMBench7 is a synthetic benchmark for evaluating STM implementations. It builds a set of graphs that simulate how real-world applications structure their objects. Then, it traverses the graphs using several operations that resemble how real-world applications navigate through their objects.

STMBench7 is highly configurable. We obtained all the results using a JVSTM backend and configured STMBench7 to run a read-write workload with 1 and 4 threads.

When running STMBench7, we disabled two types of operations: (1) long traversals and (2) structural modifications. Our JVSTM implementation of STMBench7 builds huge read-sets on long traversals, which stress the garbage collection mechanism and thus generates unacceptable amounts of noise on the final results. Likewise, we disabled structural modifications because they delete a large part of the object graphs, resulting in non significant conversion times when running the immediate conversion mode.

STMBench7 tracks how many operations were completed, on average, per second. We modified it to report, at each second, how many operations completed during that second. Then, we ran the benchmark 15 times and registered the

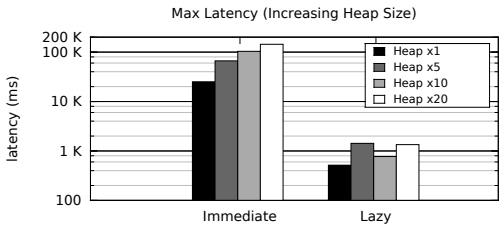

Figure 3. Maximum latency observed for operations whose behaviour does not depend on the total number of instances.

average value of completed operations per each second. We executed the benchmark on an Intel Core i5 750 processor (4 cores) with 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). To measure the throughput overhead in steady state, we ran the benchmark without installing any upgrade using two versions: One unmodified and another post-processed by DuST'M. Figure 2 shows all the measured results. We registered an average overhead of 18% for 4 threads and 37% for 1 thread.

STMBench7 also tracks the maximum latency of each operation. To find how the pause length scales with the total amount of instances, we modified STMBench7 so that it uses more instances by multiplying the total number of a common instance type by a factor of 5, 10, and 20. We ran this test on a machine with 4 AMD Opteron 6168 CPUs (48 cores in 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).

STMBench7 executes several operations that traverse a portion of all the existing objects. Increasing the total number of instances modifies the behaviour of such operations. To assess the pause that dynamic upgrades impose, we registered the maximum latency of operations whose behaviour remains constant despite increasing the total number of instances that the benchmark uses. We executed STMBench7 for 270 seconds, installing an upgrade at 90 seconds, and registered the maximum latency observed on 10 executions. Figure 3 shows those results. As expected for immediate state conversion, the maximum latency increases with the total number of instances. On the other hand, lazy state conversion has almost no effect on the maximum latency.

V. Related Work

Dynamic software upgrade systems have received much attention from the research community over the last years, resulting in a vast literature. But, as we are limited in space, we discuss only the work that more closely relates to ours.

Gupta et al. [6] propose a formal framework for dynamic software upgrades. They define an upgrade as valid if, after a finite amount of time past the upgrade, the upgraded program behaves exactly as if it was running from the start. They define formally the upgrade's validity using a state mapping and prove that this problem is generally undecidable. The upgrade validity problem maps directly to finding upgrade safe-points: If an upgrade is not applied at a safe point, the resulting program can reach an invalid state that does not exist in either the new nor the old program version.

Boyapati et al. [2] introduce a set of *upgrade modularity conditions* that allow persistent object stores to convert the state lazily whilst offering immediate upgrade semantics. They take advantage of object encapsulation to avoid using versions where possible. There are, however, cases in which encapsulation fails[3]. In such cases, the authors state that their system uses versions but they do not discuss how their system manages versions. DuST'M implements, on the programming environment, the same upgrade semantics. This brings new challenges that the authors do not address and DuST'M solves, such as the interplay between upgradable and transactional objects. Moreover, DuST'M uses versions for every object to provide the atomic upgrade semantics.

Neamtiu et. al. [8] consider dynamic upgrading transactional applications. They introduce *transactional version consistency* (TVC), a correctness property that ensures that transactions appear to execute entirely at the same code version. The authors also show how contextual effects can be used to statically enforce TVC by computing them at each code position and using them to decide if a transaction may be safely upgraded on that code position. The authors also discuss an alternative implementation that would apply upgrades optimistically and comit or rollback the transactions according to contextual effects computed at runtime. This technique resembles what Software Transactional Memories already perform for conflict detection. Some of the authors later proposed [7] induced update points, barrier synchronization, and relaxed synchronization to increase the window of opportunity to safely install upgrades.

The Java Virtual Machine itself has HotSwap [4], a limited type of dynamic software upgrades that allows to modify method bodies only. Würthinger et al. [12] presents DCE VM, a modified JVM that supports installing arbitrary modifications. Upgrades may, however, fail and cause the whole program to terminate abruptly. Furthermore, their system does not support any custom program state migration. Subramanian et al. [11] present JVolve, a similar JVM enhanced for supporting dynamic software upgrades. An upgrade may still fail, but JVolve discards it and keeps executing the program on the old version. JVolve supports custom transformer methods to convert the objects between versions. Both DCE VM and JVolve convert the program state immediately and impose virtually no steady state performance overhead.

Orso et al. [9] present DUSC, a dynamic upgrade system that does not modify the JVM. They also introduce an extra level of indirection using a bytecode post-processor.

[3]For instance, mutually referenced objects and circular object structures.

However, when compared to DuST'M, their system presents two major limitations: (1) DUSC discards upgrades if any thread is executing any method that belongs to an upgraded type, and (2) upgrades cannot modify the interfaces of upgradable types nor change the class hierarchy.

VI. Conclusion

This paper shows how to to provide immediate upgrade semantics to the developer writing the conversion code while converting the program state lazily. We describe a novel technique in which we employ a Software Transactional Memory to execute the conversion code in the logical past and ensure its correctness according to the *atomic upgrade semantics*, which we present as an adaptation of the previously proposed *upgrade modularity conditions* [2].

We also present an experimental evaluation that compares immediate with lazy state conversion. We show that the pause that immediate state conversion introduces gets longer by increasing the total size of the program's state, increasing the maximum response time up to three orders of magnitude.

Our lazy approach, on the other hand, keeps a constant maximum response time, regardless of the total size of the program's state. Unlike immediate upgrades, which can be implemented with virtually no steady state overhead [11], [12], lazy upgrades always introduce some overhead. Our experimental evaluation also addresses the steady state overhead and our results show overheads between 18% and 37%.

Our main goal when developing DuST'M is to design a practical dynamic upgrade system that is easy for the developer to use. This paper discusses how to provide simple upgrade semantics without imposing a long upgrade pause.

A part of that goal that this paper does not address is related with how the developer writes the program state conversion logic. We already have a proposal about such code [10]. We plan to validate its expressiveness by using it to specify upgrades applied to representative open source applications, such as the Fénix Edu project [1].

DuST'M's prototype also has some limitations: It keeps versions for every instance that are never garbage collected and it needs to convert every upgradable type at each upgrade, even those types that remain unchanged. We plan to solve these shortcomings by using static analysis on the conversion code to avoid keeping unnecessary versions, implementing a background thread that converts every live object in parallel with the program's normal operation, and using HotSwap [4] to avoid converting unchanged types.

Acknowledgements

This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, and by the Cloud-TM project, which is co-financed by the European Commission through the contract number 257784.

References

[1] Fénixedu. http://fenixedu.sourceforge.net, 2005.

[2] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38:403–417, October 2003.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[4] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[5] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[6] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22:120–131, February 1996.

[7] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM.

[8] I. Neamtiu, M. Hicks, J. S. Foster, and P. 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 '08, pages 37–49, New York, NY, USA, 2008. ACM.

[9] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 649–, Washington, DC, USA, 2002. IEEE Computer Society.

[10] L. Pina and J. Cachopo. DuSTM - dynamic software upgrades using software transactional memory. Technical Report 32/2011, INESC-ID Lisboa, June 2011. Available at http://www.inesc-id.pt/ficheiros/publicacoes/7298.pdf.

[11] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.

[12] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM.