Speculating on top of an unmodified Java VM *

Ivo Anjo João Cachopo
ESW
INESC-ID Lisboa/Instituto Superior Técnico/Universidade Técnica de Lisboa
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{ivo.anjo,joao.cachopo}@ist.utl.pt

Abstract

As multicore processors become the default for computing devices ranging from smartphones to cloud servers, many researchers have proposed Thread-Level Speculation (TLS) as a solution to the problem of extracting parallelism from sequential algorithms.

On managed code environments, such as the Java Virtual Machine, most TLS research is done on smaller, simpler, research Virtual Machines, which are more amenable to the multiple changes needed for speculative execution, but that do not reflect the full range of optimizations and issues that top tier VMs such as Oracle's HotSpot present.

In this paper, we make the argument for doing TLS on top of an unmodified JVM, and explore why such a system may present performance surpassing current TLS proposals.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Languages, Performance

Keywords Automatic parallelization, Thread-Level Speculation, Software Transactional Memory

1. Introduction

With the ubiquitous availability of multicore processors, parallel programming has long ceased to be the domain of a small group of expert programmers. If we want our applications to go faster, they need to be able to do their work in parallel. This shift is a major challenge for the software industry [19].

Because concurrent programming still presents a far greater challenge than sequential programming, many researchers have proposed automatic parallelization as a means of simplifying the creation of parallel applications, by extracting concurrency from sequential algorithms (see, e.g. [4, 20], for some of the earlier approaches). Furthermore, even though new applications may take advantage of multicore architectures, a vast majority of existing code is still sequential and it is not feasible to rewrite it within a reasonable time frame.

More recent proposals of automatic parallelization systems [6, 14, 17, 22] attempt to go even further by doing Thread-Level Speculation (TLS). The idea behind this approach is that the parallelization system can optimistically try to run certain parts of the code in parallel even without being sure that the outcome will be correct. At runtime, the results from a speculative execution are propagated to the global program state only after successful validation.

Managed code environments, such as the Java platform, provide an excellent target for TLS approaches, because of the comparable ease of analyzing and modifying the bytecode of applications, and the availability of both quality tools for concurrent programming and source code for many Java VM implementations.

To obtain good performance, modern Java VMs use a variety of techniques such as Just-In-Time compilation based on profiling, parallel garbage collection, and fast locking [7]. These JVMs are the result of years of tweaks and optimizations, and as such are not easily approachable for TLS research. As a result, most TLS proposals for Java are implemented on smaller, simpler VMs, which do not benefit from all of the aforementioned advantages.

In this paper, we make the argument for doing speculative execution on top of an unmodified Java VM, instead of trying to modify it. We claim that most, if not all, of the features needed for speculation can be implemented on top of the VM, with normal Java code (or any other language that runs on the JVM). We also explore some of the places where advancements in the JVM could provide better performance.

In the next section, we present some of the reasons why we believe that doing speculation on top of the JVM is a better approach for making use of the new multicore architectures. Then, in Section 3, we discuss some of the challenges in implementing a speculation system on an unmodified VM, and how they might be solved, followed by the conclusions of this short paper.

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

2. (Not) Diving Into The VM

By working on top of the JVM, instead of modifying it, we can take advantage of all the concurrency features provided by the Java platform. The Java Memory Model [12] defines how threads interact through memory, allowing correct multi-threaded applications to be built, even if they avoid the use of locking. The `java.util.concurrent` package provides utility classes to deal with atomic memory operations, and concurrent data structure implementations. The Java fork/join framework [11], recently added to Java 7, implements a lightweight framework for supporting concurrent applications that solve tasks by recursively splitting them into subtasks which can be worked on in parallel.

Production VMs also provide very advanced GC strategies, such as HotSpot's G1 Garbage Collector or Azul's pauseless GC algorithm [8]. Tuning and debugging such algorithms is hard, so TLS implementations normally choose VMs with simpler GC designs; working on top of the JVM allows us to take advantage of them.

On the other hand, doing speculation on top of the JVM tends to impose bigger overheads than other approaches. Many TLS approaches, especially those relying on hardware support, speculate the execution of tens of instructions; to amortize the overheads, speculations on an unmodified VM will need to be longer-running and be able to spawn other speculative executions.

Allowing a speculation to spawn further speculative tasks is important when using method-level speculation [16]. Because method-level speculation normally works by speculating the execution of an entire method, we might further parallelize the application if, inside a method, we speculatively parallelize the execution of other methods it calls.

3. Challenges of implementing TLS on an unmodified Java VM

Working within the confines of the Java platform provides a lot of advantages for TLS, but also imposes restrictions. Features that before were obtained via Virtual Machine modification now have to rely on the normal APIs provided to every Java application, and might have to be obtained via complex transformations.

In this Section, we explore some of the challenges and limitations of implementing the mechanisms needed for TLS on top of unmodified JVMs. In addition, we describe current research that may be used to solve them, and identify future research avenues.

3.1 Transactification

Usually, during speculation, a speculative task needs to change not only its thread-local state (such as local variables on the stack), but also state allocated on the heap. To control the application of these changes to the program state, we need a mechanism that transforms an application so that it can behave transactionally, allowing the TLS system to validate that changes done speculatively are equivalent to the original sequential semantics of the application, and to undo them when they are not.

The JVM has no such mechanisms, but Software Transactional Memory (STM) research has provided fitting solutions. The Deuce [10] Java STM framework does automatic transactification of classes by replacing instructions to read-/write fields and arrays with calls to the framework. It relies on the `sun.misc.Unsafe` internal JVM API that provides very low overhead when writing to or reading from any memory position. The API used by Deuce is not standard on the Java platform, but most JVMs provide it. An alternative would be to use *reflection*, which is a heavier mechanism, but that could be used as fallback if compatibility with all VMs is required.

The DSTM2 [9] STM framework allows the automatic creation of transactional versions of objects based on supplied interfaces. It separates transactification from the STM implementation itself, allowing multiple synchronization strategies to be used.

Another option is provided by JaSPEx [1], which replaces class fields with JVSTM's versioned boxes [5]. Accesses to those fields are in turn modified to call the JVSTM API, which maintains in each box a linked list representing the history of the values of the field, thereby allowing older-running transactions to access values after they have been updated by newer-running ones.

Whereas Software Transactional Memory is a good starting point for being able to validate and undo changes done speculatively, further research on what features are important for speculation and which restrictions can be made to their usual model might provide fruitful: Consider that, for instance, when most TLS systems spawn speculative tasks, the commit order for the tasks is known in advance, and as such the STM implementation might be optimized for such a use case; and also that isolation between multiple speculative tasks might be relaxed, allowing speculations which will commit later access to uncommitted writes from earlier speculations. In addition, concurrent models for nested transactional memory [13] could also be leveraged to provide better support for nested speculations.

While blind transactification of an entire application is a basic approach, on many workloads only a small part of the application objects are accessed by multiple threads. As such, an interesting research challenge is the identification of which parts of an application need to be transactified, or to devise a scheme to perform transactification dynamically, only when needed.

3.2 Non-Transactional Operations

During speculative execution, it is common to find non-transactional operations: operations that will result in state changes outside the control of the STM or other transactional support being used, and that cannot be undone after being applied.

On the JVM, these can be identified by detecting that a method call is invoking a `native` method. This detection can be made statically before the application is run, or at runtime [1].

Static identification consists of building a graph of possible method invocations, starting from the initial entry point of an application, and identifying which methods may trigger execution of `native` methods. Because this approach is very conservative, a better way is runtime control of their execution, which works by identifying all non-transactional operations during bytecode analysis, and by inserting a call to the speculation framework before such operations. The framework can then decide if the operation may execute (or take steps to ensure that it can) returning from that call if the non-transactional operation may proceed, or, alternatively, throwing an exception to prevent the non-transactional operation from proceeding and aborting the speculation.

If the base Java class libraries are not transactified, as described in the previous section, they also have to be considered as non-transactional operations.

Some of these restrictions may be lifted by expanding the transactional system, integrating facilities such as (limited) transactional input/output. Others may need studies and binary code analysis to identify and separate between native method calls that change hidden state and those that do not, such as asking the current time and date from the operating system.

3.3 Stack Manipulation

Depending on the speculative model being applied, the Java stack may need to be saved, changed, and restored. For instance, Pickett and Verbrugge implemented a scheme in their SableSpMT [15] system — Stack buffering — where stack frames from a parent thread are copied to child speculative threads.

Accessing and manipulating the stack is very hard to do in Java. The Apache Commons Javaflow project [3] implements continuations by rewriting application bytecode to duplicate the native Java stack into a parallel data structure managed by the Javaflow library, and by adding additional entry points to Java methods, allowing execution to be resumed from the middle of a method.

In contrast with transactification, stack manipulation seems to be better suited if implemented at the JVM level. For the HotSpot VM, two such approaches have been tried [18, 21]. Both result in much better performance than Javaflow by working directly with the real stack of Java threads, but development seems stalled for the moment.

With more and more dynamic languages that support continuations and coroutines being implemented on top of the JVM, hopefully the Java platform will be extended with such mechanisms. For JVMs that do not support it, the bytecode alternative can be used as a fallback.

3.4 Threads & Scheduling

Nowadays, most Java VMs map each Java thread to a native operating system thread. This means that most of the scheduling decisions are done by the OS scheduler. Older JVMs used to also support "green-threads": multiple user-mode threads that are mapped to a single native thread. Although they might not be useful to most Java applications, having the option to use and manage green threads would allow a speculation system to control which threads are running, allowing their execution to be suspended and resumed.

Mechanisms commonly used by multithreaded applications, such as the thread pools provided by the `java.util.concurrent.ExecutorService` API, which maps tasks to be executed into threads that execute them, or a fork/join framework such as Doug Lea's [11] are problematic for speculation. If a speculative task needs to wait for some other speculation to finish its work, and in turn that speculation is waiting for another, at some point, because the thread pool is limited and all the threads have assigned work, only one thread will be doing useful work. This issue may be alleviated by research on designing a thread pool that is better suited to the different usage patterns of a TLS system.

An additional limitation that affects speculative systems is infinite loops caused by inconsistent reads: a speculative thread may read a value and start looping based on some condition over this value, but the value was invalid (it was read too soon, before another thread wrote the correct one) and so the loop will never end. Unfortunately, it is not possible, in general, to stop such a thread, as the `java.lang.Thread` API for thread control is very limited, most of it being deprecated; on the upside, such threads will have no impact on other, correct ones, and cannot bring down the entire application.

4. Conclusion

In this paper we proposed that Thread-Level Speculation systems be implemented on top of unmodified JVMs, instead of through the use of especially-modified VMs. By doing this, the TLS system will be able to take advantage of the performance features provided by modern production VMs, such as their advanced garbage collection mechanisms and Just-in-Time optimizing compilers.

We described some of the challenges of implementing TLS on top of the JVM, discussed some possible approaches to tackle those challenges, and expanded on features that could be added to the Java platform that would prove useful for speculative execution. We believe that in places where JVM extensions are needed, those extensions — while benefiting speculative execution — are not specific to it, and that they would be useful to many other Java/JVM applications.

In the future, we plan to continue our work on the RuLAM project [2], which aims to deliver speculative parallelization of legacy Java applications.

References

[1] Anjo, I.: JaSPEx: Speculative Parallelization on the Java Platform. Master's thesis, Instituto Superior Técnico (2009)

[2] Anjo, I., Cachopo, J.: RuLAM Project: Speculative Parallelization for Java using Software Transactional Memory [Extended abstract]. In: 8th International Conference on Principles and Practice of Programming in Java (PPPJ 2010). Vienna University of Technology (2010)

[3] Apache Software Foundation: Apache Commons Javaflow, `http://commons.apache.org/sandbox/javaflow/` (2009)

[4] Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., et al.: Polaris: The next generation in parallelizing compilers. In: Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing. pp. 10–1. Springer-Verlag, Berlin/Heidelberg (1994)

[5] Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172–185, Elsevier (2006)

[6] Chen, M., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. In: Proceedings of the 30th annual international symposium on Computer architecture. pp. 434–446. ACM New York, NY, USA (2003)

[7] Click, C.: A JVM Does That???, `http://www.azulsystems.com/resources/presentations/a_jvm_does_that` (2010)

[8] Click, C., Tene, G., Wolf, M.: The pauseless gc algorithm. In: Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments. pp. 46–56. ACM (2005)

[9] Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices 41(10), 253–262, ACM (2006)

[10] Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. Programmability Issues for Multi-Core Computers (MULTIPROG'10) (2010)

[11] Lea, D.: A Java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande. pp. 36–43. ACM New York, NY, USA (2000)

[12] Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 378–391. ACM, New York, NY, USA (2005)

[13] Moss, J.E.B., Hosking, A.L.: Nested transactional memory: model and architecture sketches. Sci. Comput. Program. 63(2), 186–201, Elsevier North-Holland, Inc. (2006)

[14] Oancea, C., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. pp. 223–232. ACM (2009)

[15] Pickett, C., Verbrugge, C.: Software thread level speculation for the Java language and virtual machine environment. Languages and Compilers for Parallel Computing pp. 304–318, Springer (2006)

[16] Pickett, C., Verbrugge, C., Kielstra, A.: Understanding method level speculation. Tech. rep., Sable Research Group, School of Computer Science, McGill University (2009)

[17] Spear, M., Kelsey, K., Bai, T., Dalessandro, L., Scott, M., Ding, C., Wu, P.: Fastpath speculative parallelization. Languages and Compilers for Parallel Computing pp. 338–352, Springer (2010)

[18] Stadler, L., Würthinger, T., Wimmer, C.: Efficient coroutines for the java platform. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 20–28. ACM (2010)

[19] Sutter, H., Larus, J.: Software and the concurrency revolution. Queue 3(7), 54–62, ACM (2005)

[20] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: an infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Not. 29(12), 31–37, ACM, New York, NY, USA (1994)

[21] Yamauchi, H.: Continuations in servers, `http://wikis.sun.com/display/mlvm/StackContinuations` (2010)

[22] Yoo, R., Lee, H.: Helper Transactions: Enabling Thread-Level Speculation via A Transactional Memory System. PESPMA 2008 p. 63 (2008)