Accelerating Sequential Consistency for Java with Speculative Compilation

Lun Liu
University of California, Los Angeles
USA
lunliu93@cs.ucla.edu

Todd Millstein
University of California, Los Angeles
USA
todd@cs.ucla.edu

Madanlal Musuvathi
Microsoft Research, Redmond
USA
madanm@microsoft.com

Abstract

A memory consistency model (or simply a memory model) specifies the granularity and the order in which memory accesses by one thread become visible to other threads in the program. We previously proposed the *volatile-by-default* (VBD) memory model as a natural form of *sequential consistency* (SC) for Java. VBD is significantly stronger than the Java memory model (JMM) and incurs relatively modest overheads in a modified HotSpot JVM running on Intel x86 hardware. However, the x86 memory model is already quite close to SC. It is expected that the cost of VBD will be much higher on the other widely used hardware platform today, namely ARM, whose memory model is very weak.

In this paper, we quantify this expectation by building and evaluating a *baseline* volatile-by-default JVM for ARM called VBDA-HotSpot, using the same technique previously used for x86. Through this baseline we report, to the best of our knowledge, the first comprehensive study of the cost of providing language-level SC for a production compiler on ARM. VBDA-HotSpot indeed incurs a considerable performance penalty on ARM, with average overheads on the DaCapo benchmarks on two ARM servers of 57% and 73% respectively.

Motivated by these experimental results, we then present a novel *speculative* technique to optimize language-level SC. While several prior works have shown how to optimize SC in the context of an offline, whole-program compiler, to our knowledge this is the first optimization approach that is compatible with modern implementation technology, including dynamic class loading and just-in-time (JIT) compilation. The basic idea is to modify the JIT compiler to treat each object as *thread-local* initially, so accesses to its fields can be compiled without fences. If an object is ever accessed by a second thread, any speculatively compiled code for the object is removed, and future JITed code for the object will include the necessary fences in order to ensure SC. We demonstrate that this technique is effective, reducing the overhead of enforcing VBD by one-third on average, and additional experiments validate the *thread-locality hypothesis* that underlies the approach.

CCS Concepts • **Software and its engineering → Concurrent programming languages**; **Just-in-time compilers**; **Runtime environments**.

Keywords memory consistency models, volatile by default, sequential consistency, Java virtual machine, speculative compilation

ACM Reference Format:
Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2019. Accelerating Sequential Consistency for Java with Speculative Compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314611

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314611

1 Introduction

A memory consistency model (or simply a memory model) specifies the granularity and the order in which memory accesses by one thread become visible to other threads in the program. We recently proposed a strong memory model for Java called *volatile-by-default* [20]. In this semantics, all variables are treated as if they were declared `volatile`, thereby providing *sequential consistency* (SC) at the level of Java bytecode. Specifically, all programs are guaranteed to obey the per-thread program order, and accesses to individual memory locations (including `doubles` and `longs`) are always atomic. In contrast, the Java memory model (JMM) [22] only provides these guarantees for programs that are data-race-free, and it has a weak and complex semantics for programs that contain data races.

The cost of the strong volatile-by-default semantics is a loss of performance versus the JMM — the compiler must restrict its optimizations to avoid reordering accesses to shared memory, and it must also insert fence instructions in the generated code to prevent the hardware from performing such reorderings. This cost is relatively modest for a modified

version of Oracle's HotSpot JVM running on Intel x86 hardware [20], but the x86 memory model is already quite close to SC. In particular, `volatile` reads do not require any hardware fences in x86. As reads tend to far outnumber writes in programs, it is reasonable to expect the overhead of the volatile-by-default semantics to be much higher on weaker memory models such as ARM and PowerPC, which require fences for both `volatile` reads and writes.

In this paper, we quantify this expectation by building and evaluating a *baseline* volatile-by-default JVM for ARM using the same implementation technique as previously used for x86 [20]. This new JVM, which we call VBDA-HotSpot, is a modification of the ARM port of the HotSpot JVM from OpenJDK8u[1]. Through this baseline we report, to the best of our knowledge, the first comprehensive study of the cost of providing language-level SC for a production compiler on ARM. Motivated by these experimental results, we then present a novel *speculative* technique to optimize language-level SC and demonstrate its effectiveness in reducing the overhead of VBDA-HotSpot. While several prior works have shown how to optimize SC in the context of an offline, whole-program compiler [17, 35, 38], to our knowledge this is the first optimization approach that is compatible with modern implementation technology, including dynamic class loading and just-in-time (JIT) compilation. The implementations of VBDA-HotSpot and the optimized version, called S-VBD, are available on our GitHub repository: https://github.com/Lun-Liu/schotspot-aarch64.

Baseline Overhead. Experiments on our baseline implementation, VBDA-HotSpot, show that the volatile-by-default semantics incurs a considerable performance penalty on ARM, as expected. However, we observe that the performance overhead crucially depends on the specific fences used to implement the `volatile` semantics. With the default fences that HotSpot employs to implement `volatile` loads and stores on ARM, VBDA-HotSpot incurs average and maximum overheads of 195% and 429% (!) on the DaCapo benchmarks [7] for a modern 8-core ARM server. But employing an alternative choice of ARM fences reduces the average and maximum overheads on that machine respectively to 73% and 129%. We also find similar results on a 96-core ARM server, with VBDA-HotSpot incurring an average and maximum overhead of 57% and 157% with the alternative fences.

Speculative Compilation. The overheads above motivate the need to develop techniques to optimize language-level SC. Prior work in this area has leveraged heavyweight, whole-program analyses to eliminate fences, such as delay-set analysis [35] and object escape analysis [17, 38]. While that work shows that such optimizations can provide low overheads for language-level SC, they are not compatible with modern virtual-machine technology, where classes are loaded dynamically and code is optimized and compiled during execution. Indeed, the HotSpot JVM already includes an escape analysis, and VBDA-HotSpot does not insert fences for field accesses of non-escaping objects, but the constraints of the JVM require this analysis to be fast and intraprocedural, thereby limiting its effectiveness.

We propose a new optimization technique that, to the best of our knowledge, is the first optimization for language-level SC that is compatible with modern language features such as JIT compilation and dynamic class loading. Like other JIT-based optimizations, it is *speculative*. The basic idea is to modify the JIT compiler to treat each object as *safe* initially, meaning that accesses to its fields can be compiled without fences. If an object ever becomes *unsafe* during execution, any speculatively compiled code for the object is removed, and future JITed code for the object will include the necessary fences in order to ensure SC.

It is challenging to turn this high-level idea into an effective optimization. First, dynamic concurrency analyses for Java are very expensive, so a direct application of these analyses to track whether an object is *safe* would defeat the purpose. For example, dynamic data-race detection would provide the most precise notion of *safe*, but such analyses for Java have average overheads of 8× or more [40].

Instead our approach treats an object as *safe* if it is *thread-local*, i.e. accessed only by the thread that created it. While this approach unnecessarily inserts fences on objects that are shared but data-race-free, we hypothesize that many objects are thread-local, thereby allowing us to remove a large percentage of fences. Our experimental evaluation shows that this hypothesis is well founded. Further, we leverage this approximate notion of *safe* to optimize its enforcement, and in particular we develop an intraprocedural analysis to remove many per-access thread-locality checks that are provably redundant.

Second, our technique must properly account for memory accesses that come from interpreted code as well as those from JITed native code. We have chosen to always insert fence instructions for memory accesses in interpreted code. This approach trades some performance for implementation simplicity, but since "hot" code will eventually get compiled we expect the performance loss to be minimal. The memory accesses from interpreted code must still be checked for violations of thread locality in order to ensure correctness of the JITed code.

Third, our approach must handle the case when some instances of a class are thread-local and others are not. To avoid having multiple compiled versions of a method, which would cause code bloat and violate the existing constraints of the HotSpot JVM, we choose a design where there is exactly one compiled version per method. That is, as soon as any instance of a class violates thread locality, all instances of

[1] http://hg.openjdk.java.net/aarch64-port/jdk8u

the class will use the version of compiled code containing fences.

Finally, once a violation of thread locality is detected for some object, we require a low-complexity way to switch from the fence-less to fence-ful versions of that object's code. We demonstrate that the HotSpot JVM's *deoptimization* capabilities can be adapted for this purpose. Specifically, we use HotSpot's *dependency tracking* mechanism to record the compiled methods that may access objects of a given class `C`. The first time that some instance of `C` is found to violate thread locality, VBDA-HotSpot invokes HotSpot's deoptimization facility to safely pause all threads and remove the compiled versions of all methods that depend on `C` before resuming execution. If JIT compilation is later triggered on any of these methods, the fence-ful version will be used. This approach dramatically simplifies our implementation and allows us to gain confidence in its correctness.

In addition to speculative compilation, we have devised an orthogonal optimization that reduces the number of fences required to enforce the volatile-by-default semantics for ARM. We observe that some of the barriers inserted for `volatile`s in Java are only there to prevent reorderings with regular, non-`volatile` accesses. Hence in the volatile-by-default setting, where all accesses are treated as `volatile`, it is safe to eliminate some of these barriers, which in turn eliminates some fence instructions in the generated code.

Our modified version of VBDA-HotSpot that employs the speculative approach and the fence optimization is called S-VBD. On the DaCapo benchmarks, S-VBD reduces the overhead of enforcing the volatile-by-default semantics by roughly 1/3, bringing the average overhead from 73% to 51% and from 57% to 37% on our two servers respectively. Further, the maximum overhead over the baseline HotSpot JVM reduces even more, from 129% to 78% and from 157% to 73% on the two machines. Finally, by isolating the portion of S-VBD's overhead due to the speculative checks, we are able to validate the thread-locality hypothesis that underlies our speculative technique.

In summary, this paper provides the first empirical measurement of the cost of language-level SC for ARM in a production Java compiler. We also describe the first optimization approach for language-level SC that works within the constraints of the Java language and modern JVM technology, and we demonstrate that it provides significant performance improvements. Longer-term, the goal of this line of research is to make the performance of strong language-level memory models like volatile-by-default competitive even on weak hardware.

2 A Volatile-by-Default JVM for ARM

This section presents and evaluates a baseline implementation of the volatile-by-default semantics for ARM, which we call VBDA-HotSpot. We employ the same implementation technique previously used to create VBD-HotSpot, a volatile-by-default version of the HotSpot JVM for x86 [20]. The basic idea is to treat all heap accesses in both the interpreter and compiled code as if they were declared `volatile`. To our knowledge this is the first comprehensive study of the cost of language-level SC, for any language, on ARM.

2.1 Implementation

VBD-HotSpot [20] is a modification to the HotSpot JVM, version 8u, to provide the volatile-by-default semantics on Intel x86 hardware. That JVM does not have an ARM backend. However, Oracle's OpenJDK includes another project that is a port of the HotSpot JVM to support the Linux/Aarch64 platform (the 64-bit mode of the ARMv8 Architecture). Therefore, to create VBDA-HotSpot we ported the platform-independent portions of VBD-HotSpot to version 8u of that JVM[2] and then augmented it with the ARM-specific implementation of the volatile-by-default semantics. This involves modifying the JVM's interpreter and JIT compiler, which we discuss in turn.

Interpreter. The HotSpot JVM uses a *template-based interpreter* for performance reasons. In this style a `TemplateTable` maps each bytecode instruction to a *template*, which is a set of assembly instructions. Hence the interpreter is platform-specific, requiring us to develop a new volatile-by-default version for ARM.

We manually inspected the template instructions in the ARM interpreter for the bytecodes that read from or write to memory, such as `getfield` and `putfield`. The template code for each bytecode checks the `volatile` attribute of the given field and adds the necessary fences if the attribute is set. In VBDA-HotSpot we have modified this template code to unconditionally add the necessary fences, thereby treating all memory reads and writes as `volatile`. Interestingly, the template code for `getfield` already unconditionally adds the necessary fences without checking the `volatile` attribute of the field, so it did not require any modification.

As in VBD-HotSpot [20] we also treat accesses to array elements as `volatile` by inserting the appropriate fences in the template code for the corresponding bytecodes, such as `aaload` and `aastore`. Array accesses are the primary reason that VBDA-HotSpot requires modifying the JVM and cannot be implemented as a source-to-source or bytecode-to-bytecode transformation, as there is no way in the Java language or bytecode to declare array elements as `volatile`.

To implement the semantics of `volatile` on ARM, the interpreter must insert a load-load and load-store barrier after a `volatile` load, providing *acquire* semantics for the load; a store-store barrier and a load-store barrier before a `volatile` write, providing *release* semantics for the write; and a store-load barrier after a `volatile` write. The interpreter uses ARM's `dmb` (data memory barrier) instruction for

[2] http://hg.openjdk.java.net/aarch64-port/jdk8u

this purpose. In particular, it needs a `dmb ishld` instruction to enforce acquire semantics after a load, `dmb ish` to enforce release semantics before a store, and `dmb ish` to enforce store-load dependencies after a store.

However, the baseline HotSpot JVM has a bug of inserting an overly weak barrier before `volatile` writes in the interpreter. Specifically it inserts a `dmb ishst` instruction, which performs a store-store barrier but not also a load-store barrier; obtaining both barriers instead requires a `dmb ish` instruction.[3] We have fixed this bug and use the fixed version of the baseline HotSpot JVM in all of our experiments. Interestingly, the use of `dmb ishst` before writes suffices for VBDA-HotSpot, because the preceding memory operation must end in a `dmb ish` (for stores) or a `dmb ishld` (for loads), both of which act as load-store barriers.

Server Compiler. VBD-HotSpot provides a volatile-by-default version of HotSpot's *server* compiler, which performs aggressive optimizations. In VBD-HotSpot all memory-access nodes in the HotSpot compiler's intermediate representation (IR), called the *ideal graph*, are treated as `volatile` by unconditionally including appropriate memory-barrier nodes before and after them. This has the effect of restricting downstream compiler optimizations and causing the necessary hardware fences to be inserted during code generation. VBD-HotSpot adds memory-barrier nodes to all memory accesses, including accesses to array elements. Because the modifications to the compiler were made in HotSpot's IR, they are platform-independent. Therefore, we simply ported those modifications to the version of HotSpot that supports ARM.

By default, HotSpot performs an optimization to identify `volatile` loads and stores in the ideal graph and implement them with aarch64's `ldar` and `stlr` instructions, in order to respectively obtain acquire semantics for a load and release semantics for a store with a single instruction. These *one-way* fence instructions were introduced in ARMv8 in part in order to support `volatile` accesses more efficiently [4]. If the backend cannot identify that a memory-barrier node is part of a `volatile` read or write, it employs the regular *two-way* fence instruction (`dmb`) for that node as described above for the interpreter. The HotSpot JVM also includes a flag `-XX:+UseBarriersForVolatile` to turn off the optimization and force the compiler to always use `dmb` instructions to implement memory barriers. VBDA-HotSpot's implementation is independent of the backend and so we can employ and evaluate both approaches.

Intrinsics. The HotSpot interpreter and compiler rely on many *intrinsics*, which are custom implementations for common library routines such as arithmetic and string operations. We manually examined all of the intrinsics for ARM and inserted the necessary fences around memory accesses to ensure the volatile-by-default semantics.

Correctness. VBD-HotSpot includes a suite of litmus tests, such as a version of Peterson's lock, that sometimes exhibit non-SC behavior under the unmodified HotSpot JVM. We added several more litmus tests to this suite that can expose ARM's weaker behaviors [31]. In total we have 14 litmus tests, and for each test we have a version using objects and a version using arrays. These tests are also available on our GitHub repository: https://github.com/Lun-Liu/schotspot-aarch64/tree/master/litmustests. We use this suite to gain confidence in our implementation by ensuring that the non-SC behaviors never occur under VBDA-HotSpot. These litmus tests are also used to gain confidence in the S-VBD implementation described in Section 3.

2.2 Performance Evaluation

We compared the performance of VBDA-HotSpot to that of the baseline JVM on several benchmark suites. We ran experiments on two multicore 64-bit ARM servers: machine A has 8 Cortex A57 cores, 16G memory, and is running openSUSE Tumbleweed; machine B has 2 Cavium ThunderX CN8890 CPU (96 cores in total), 128G memory, running Ubuntu 16.04.

2.2.1 DaCapo Benchmarks

The DaCapo benchmark suites are a widely used set of Java applications to evaluate Java performance [7]. We use the latest maintenance release (9.12-MR1) of the DaCapo benchmarks from January 2018. Among all tests, we remove `batik` which fails on the baseline aarch64 port of OpenJDK 8u (even without any of our modifications), `tradesoap` which fails periodically, apparently due to an incompatibility with the `-XX:-TieredCompilation` flag that VBDA-HotSpot requires (as discussed below)[4], and `tomcat` due to a problem unrelated to DaCapo[5]. We also replace `lusearch` with the new `lusearch-fix` benchmark that includes a bug fix, as recommended by the authors of the DaCapo benchmarks in their latest release.

We used the default workload and thread numbers for each benchmark. We employed an existing methodology for Java performance evaluation [13]. In each JVM invocation we ran a benchmark for 15 warm-up iterations and then calculated the average running time of the next five iterations. We ran five invocations of each benchmark using this process and calculated the average of these per-invocation averages. Finally we calculated the relative execution time of each benchmark using the average of the averages and then calculated the geometric mean of the relative execution times over all benchmarks.

[3] This bug has been confirmed and fixed by the developers: http://hg.openjdk.java.net/jdk/jdk/rev/e2fc434b410a

[4] https://bugs.openjdk.java.net/browse/JDK-8067708

[5] https://bugs.openjdk.java.net/browse/JDK-8155588

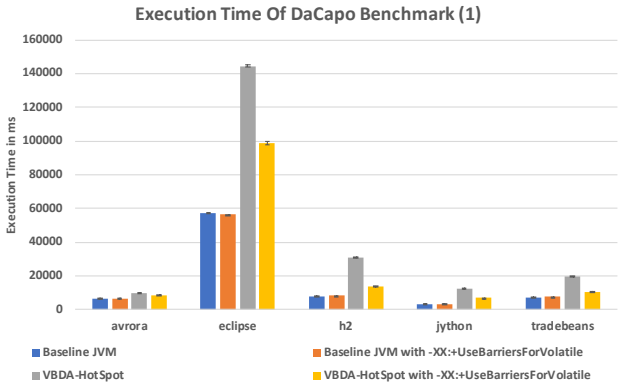

Figure 1. Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

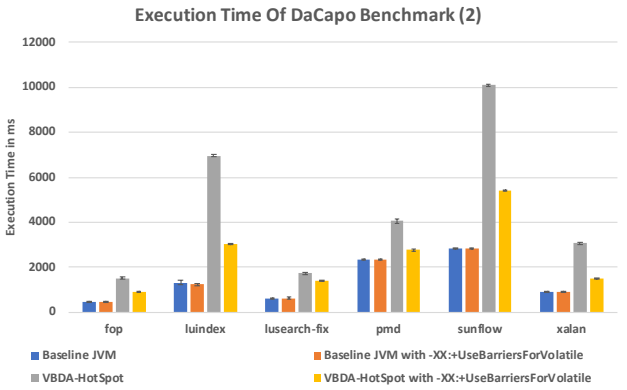

Figure 2. Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

Figures 1 and 2 show the execution time in ms for the baseline JVM and VBDA-HotSpot on machine A. The error bars show 95% confidence intervals. We use the flag `-XX:-TieredCompilation` in all versions in order to turn off *tiered compilation*, which employs multiple compilers, since we have only modified the server compiler to respect the volatile-by-default semantics. We have verified that for the baseline HotSpot JVM, there is very little performance difference with and without tiered compilation on the DaCapo benchmarks.

The figures show that for the baseline JVM, the performance with or without the `-XX:+UseBarriersForVolatile` flag is almost the same (1% difference). However, VBDA-HotSpot is much faster with the flag than without it, even though the new one-way fences are intended to improve the performance of `volatile` accesses. On further investigation, we identified two causes for this counter-intuitive behavior. First, we ran some microbenchmarks and were not able to identify any performance improvement of the acquire-release operations over the use of memory barriers.

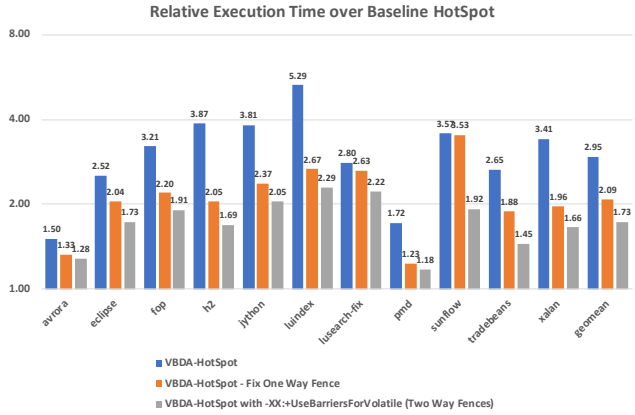

Figure 3. Relative execution time of VBDA-HotSpot, VBDA-HotSpot with bug fix for one way fences, VBDA-HotSpot with two way fences on machine A, y-axis in logarithmic scale.

So it appears that the ARM hardware that we use is still not exploiting the release-acquire semantics of the one-way fence instructions in their implementation.

Second, HotSpot's support for these instructions does not seem to be mature. For instance, these new instructions do not (yet) support offset-based addressing, so the compiler often requires an additional register to use these instructions. As has been reported by others, this adversely interacts with the current register-allocation heuristics of HotSpot.[6] Fixing those heuristics as suggested in the bug report makes a dramatic difference, as shown in Figure 3, reducing the average overhead of VBDA-HotSpot versus the baseline HotSpot JVM from 195% to 109%. However, the version with two-way fences is still significantly faster, with an average overhead of 73%. Therefore, in the rest of the paper we report numbers with the `-XX:+UseBarriersForVolatile` flag for VBDA-HotSpot.

The first series in Figures 4 and 5 respectively shows the relative execution time of VBDA-HotSpot over the baseline HotSpot JVM on machine A and machine B (for machine A these are the same numbers as shown in the last series in Figure 3). The geometric mean of the relative execution time shows an average overhead of 73% for DaCapo benchmarks, with a maximum overhead of 129% for `luindex` on machine A, and an average overhead of 57% with a maximum overhead of 157% for machine B.

To better understand the overhead of VBDA-HotSpot, we also implemented an "x86-like" version of VBDA-HotSpot that inserts the store-load barriers after each store but removes all other barriers in the interpreter, in the intrinsics implementations, and in the ideal graph for the compiler. The second series in Figures 4 and 5 respectively shows the

[6] https://bugs.openjdk.java.net/browse/JDK-8183543

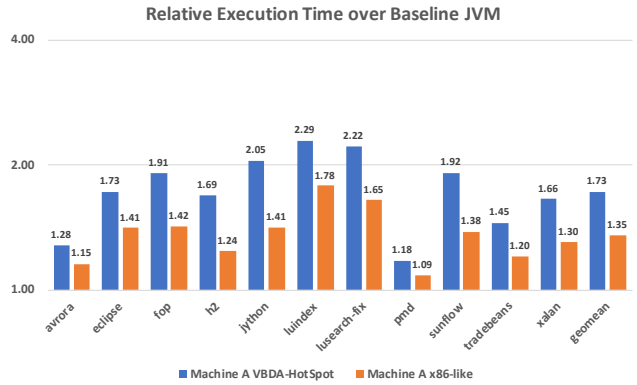

Figure 4. Relative execution time of VBDA-HotSpot and x86-like VBDA-HotSpot over baseline JVM for DaCapo on machine A, y-axis in logarithmic scale.

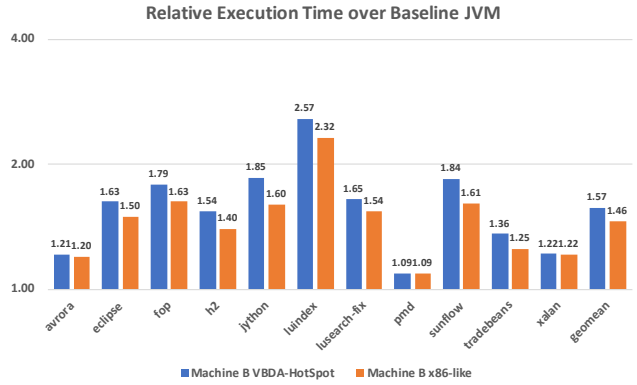

Figure 5. Relative execution time of VBDA-HotSpot and x86-like VBDA-HotSpot over baseline JVM for DaCapo on machine B, y-axis in logarithmic scale.

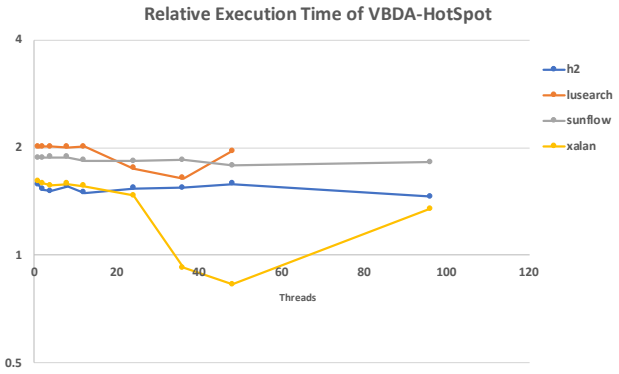

Figure 6. The relative cost of VBDA-HotSpot with different numbers of threads/cores on machine B. `lusearch` does not support running with 96 threads, y-axis in logarithmic scale.

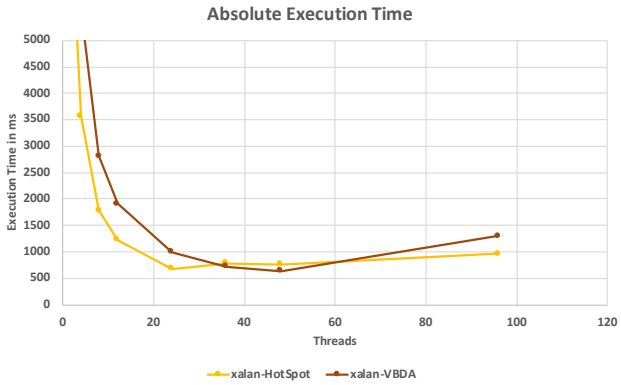

Figure 7. The execution time of VBDA-HotSpot and baseline JVM running `xalan` with different numbers of threads/cores on machine B.

relative execution time of this x86-like VBDA-HotSpot over the baseline JVM on machine A and machine B.

The x86-like VBDA-HotSpot results in average and maximum overheads of 35% and 78% for DaCapo on machine A, and average and maximum overheads of 46% and 132% for machine B. In other words, the additional fences on reads required by VBDA-HotSpot only double the overhead versus the x86-like version, despite the fact that reads dominate writes in typical programs. The x86-like implementation must insert a full fence, `dmb ish`, after a `volatile` write to implement the store-load barrier, so it seems that additional fences do not incrementally add much overhead.

2.3 Scalability Experiments

We also performed experiments on both machine A and machine B to understand how the overhead of VBDA-HotSpot changes with the number of threads/cores available, as an indication of how the cost of volatile-by-default may change as processors introduce more cores. Since committing a memory operation involves inter-core communication (e.g., coherence messages) and fences require a core to stall until all prior operations have committed, one may expect the overhead of the volatile-by-default approach to increase with the number of cores. We previously showed that in fact the relative overhead of volatile-by-default decreases or stays the same as the number of cores increases on x86 [20], apparently because of the additional cost of regular loads and stores, but we were interested to investigate whether the same would hold true on the weaker ARM platform.

Machine A has 4 sockets, each with 2 cores. Machine B has 2 sockets, each with 48 cores. For these experiments we used the `-t` option in DaCapo to set the number of external threads for each test and Linux's `taskset` command to pin execution to certain cores. We choose the benchmarks in DaCapo which exhibit external concurrency, namely `h2`, `lusearch-fix`, `sunflow`, `xalan`.

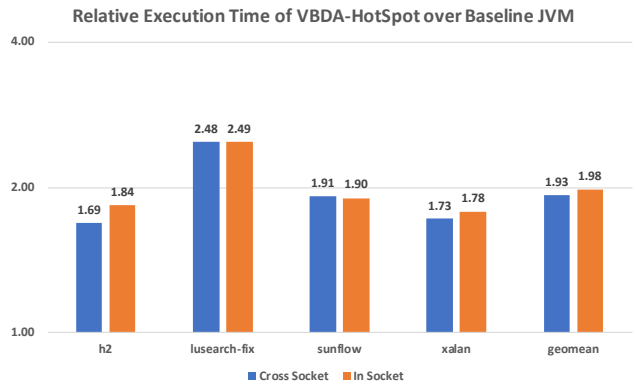

Figure 8. The cost of VBDA-HotSpot cross-socket and within socket on machine A, y-axis in logarithmic scale.

Experiments on both machines show a similar trend as in the prior x86 work: as the number of driver threads/cores increases, the relative overhead of VBDA-HotSpot stays the same or decreases modestly. The results of our experiment on machine B are shown in Figure 6; the results for machine A are similar. The figure shows how the relative execution time of VBDA-HotSpot changes with different numbers of external threads. Interestingly, VBDA-HotSpot is faster than the baseline HotSpot JVM for the `xalan` benchmark at 36 and 48 cores. Plotting the absolute execution times, as shown in Figure 7, we see that the baseline JVM stops scaling after 24 threads while VBDA-HotSpot stops at 48 threads.

Cross-socket memory accesses tend to be more expensive than within-socket accesses. To understand how the additional fences of VBDA-HotSpot affect this trend, we performed an experiment to measure the relative performance difference for VBDA-HotSpot when running on cores within the same socket versus on cores across sockets. We ran the DaCapo benchmarks with 2 driver threads in two different configurations: one using cores 0 and 1, which are on the same socket, and one using cores 0 and 2, which are on different sockets. Figure 8 shows the relative execution times for each configuration on VBDA-HotSpot versus that same configuration executed on the baseline JVM. From the results we can see that the relative overhead of VBDA-HotSpot in the multiple-sockets configuration is the same or slightly lower than that in the single-socket configuration. These results are again consistent with the earlier experiments on x86 [20].

2.4 Spark Benchmarks

Finally, we tested VBDA-HotSpot's performance on big data and machine learning benchmarks for Apache Spark [41]. As in prior work [20] we use the `spark-tests` and `mllib-tests` benchmarks from the `spark-perf` repository[7] provided by Databricks.

We ran Spark in standalone mode on a single machine, which reduces the latency of network communication versus running Spark on a cluster. Therefore, this experiment shows the worst-case cost. The executor memory is set to 4GB and the driver memory is set to 1GB. We also use a scale factor of 0.005 for workloads to adapt for single-machine execution. The `spark-perf` framework runs each benchmark multiple times and calculates the median execution time. We ran the `spark-perf` framework for 5 invocations and calculated the average of the median execution times of each test.

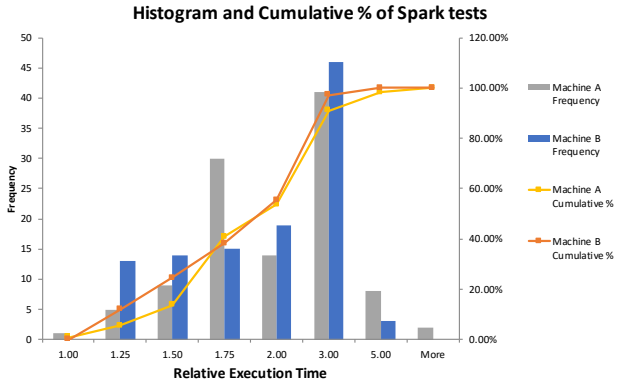

Figure 9. Histogram and cumulative % of relative execution time of VBDA-HotSpot for Spark benchmarks.

Figure 9 summarizes the results in a histogram. For example, the second gray bar from the left indicates that on Machine A there are 5 benchmarks that incur a relative execution time between 1.00 (exclusive) and 1.25 (inclusive). We omitted four benchmarks that failed on the original JVM (*als*, *kmeans*, *gmm*, and *pic*). The geometric mean of VBDA-HotSpot's relative execution time on Machine A and Machine B is 2.03 and 1.85 respectively, representing a 103% and 85% overhead over the baseline HotSpot JVM. These results are consistent with those found for VBD-HotSpot in the prior work[20].[8] We suspect that the high overhead for Spark tests versus the DaCapo benchmarks is due to the many array reads and writes that are necessary to implement Spark's key data structure, the resilient distributed dataset (RDD) [41]. Indeed, on a version of VBDA-HotSpot that (unsoundly) omits fences for array-element accesses, the average overhead of the Spark benchmarks on Machine A is reduced from 103% to 46%.

[7] The original repository is at https://github.com/databricks/spark-perf. We used an updated version that is compatible with Apache Spark 2.0 at https://github.com/a-roberts/spark-perf.

[8] See the appendix in the updated version of the paper at http://web.cs.ucla.edu/~todd/research/oopsla17.pdf.

3 Speculative Compilation

As shown in the previous section, the baseline implementation of the volatile-by-default semantics for Java incurs considerable overhead on a weak platform like ARM. Therefore, we have designed and implemented an alternative approach to enforcing the volatile-by-default semantics in the JVM, which we call *speculative compilation*. The basic idea is to modify the JIT compiler to treat each object as *safe* initially, meaning that accesses to its fields can be compiled without fences. If an object ever becomes *unsafe* during execution, any speculatively compiled code for the object is removed, and future JITed code for the object will include the necessary fences in order to ensure SC. We call the version of HotSpot that uses this approach to ensuring the volatile-by-default semantics S-VBD.

3.1 Design Overview

Several design decisions must be made to turn the above high-level idea into a concrete approach that in fact provides performance improvements.

First, the notion of *safe* must be instantiated. It must capture a large percentage of objects at runtime to reduce the overhead of volatile-by-default semantics, but the cost of checking safety should not mask the achieved savings. The most precise approach would be to convert an object from *safe* to *unsafe* only when a data race is detected on that object. However, dynamic data-race detection is quite expensive, so employing it would erase any performance advantage of this approach over the implementation of volatile-by-default described in the previous section.

Instead, we treat an object as *safe* if it is *thread-local*: all accesses to the object's fields occur on the thread that created the object. This definition is motivated by the expectation that many objects will be single-threaded throughout their lifetime. These include non-escaping objects that are not allocated in the stack due to the imprecision in HotSpot's escape analysis, and objects that are reachable from global data structures but are nevertheless logically thread-local. VBDA-HotSpot unnecessarily incurs the cost of fences for these objects.

To track this notion of safety, it suffices to record the ID of the thread that creates each object. Whenever an object's field is accessed, we compare the recorded ID to the ID of the current thread, in order to decide whether the object can still be treated as *safe* or not. Once an object becomes *unsafe* it remains so for the rest of its lifetime, so no more checking is required.

Even though checking thread locality is much less expensive than checking for data races, the runtime overhead would be prohibitive if we do this check on every field access. However, a key property of the way we define thread locality is that many of these checks can be statically eliminated. Specifically, the check whether an object is created by the current thread is *invariant* throughout a method since all accesses in a method are executed by the same thread. Therefore, we can safely replace multiple per-access checks to an object with a single check at the beginning of the method. Note that this is sound even if an object becomes non-thread-local in the middle of a method — the second thread that accesses the object will force a decompilation of this method. We have implemented an intraprocedural analysis in S-VBD that performs this optimization for the receiver object `this` of each method.

Second, speculative compilation requires that we have both a *slow* and *fast* version of each method, respectively with and without fences inserted. The most precise approach would be to keep track of the appropriate version on a per-object basis. However, to vastly simplify our implementation we instead switch on a per-class basis. That is, as soon as any instance of class `C` becomes *unsafe*, we switch to the *slow* version of `C`'s compiled methods, and this version is used for all instances of the class. This approach ensures that only a single version of `C`'s compiled code is active at any given point in time, which accords with a constraint in the original HotSpot JVM.

Third, we must decide how to switch from *safe* to *unsafe* mode in a correct and low-complexity way. We observe that the HotSpot JVM already has support for *deoptimization* of compiled methods, which is used when an assumption about a method (e.g., that no method overrides it) is violated (e.g., when a new class is dynamically loaded). We show how to leverage this capability for our purpose. Specifically we use HotSpot's *dependency tracking* mechanism to record the speculatively compiled *fast* methods that may access fields of objects of a given class `C`. The first time that some instance of `C` is found to be *unsafe*, S-VBD invokes HotSpot's deoptimization facility to safely pause all threads and remove the compiled versions of all methods that depend on `C` before resuming execution. If JIT compilation is later triggered on any of these methods, the *slow* versions will be used.

Finally, we have described our design for accesses to the fields of an object. Conceptually this speculative approach could also be used for accesses to static fields and array elements. However, to reduce implementation complexity we currently treat these accesses exactly as in VBDA-HotSpot. Specifically, we unconditionally insert the appropriate memory barriers for these accesses to ensure the `volatile` semantics. We also unconditionally insert fences for intrinsics as in VBDA-HotSpot.

3.2 Implementation

Implementing this design is non-trivial: both the JIT compiler and the interpreter must be updated to perform safety checks, and *fast* code must never be executed after a relevant safety check fails, even when that failure happens on another thread. This subsection describes our implementation in detail.

```
compile(m) {
  if(m.class.mode==fast) {
    compile fast_version(m);
    critical_section_begin;
    if(m.class.mode != fast)
      abort_compilation();
    else
      register_compiled_method();
    critical_section_end;
  }
  else {
    compile slow_version(m);
    register_compiled_method();
  }
}
```

Figure 10. Just-in-time compilation of a method.

To simplify the presentation, we first describe our implementation under the assumption that all field accesses are of the form `this.f`. If that is the case, then it suffices to replace all per-field-access checks with a single check of the `this` object at the beginning of each method. As mentioned above, we have implemented an intraprocedural analysis at class-load time that performs this optimization. We then describe the more general case where per-field-access checks are required in the next subsection.

To determine whether an object is *safe*, we add another word in each object header which contains the ID of the thread that created the object. Therefore the safety check simply compares this value to the ID of the current thread. We also must remember whether a class is using the *fast* or *slow* versions of its methods; we add a flag to HotSpot's VM-level representation of each class for this purpose.

Figure 10 shows what happens when a method gets "hot" enough and is chosen to be compiled. We check whether the method's class is in *fast* or *slow* mode and compile the corresponding version of the method. After compilation of the *fast* version we check the class's mode again. If the class's mode has changed, it means that some object of the class has been found to be *unsafe* on another thread in the meanwhile, so we abort the compilation. Otherwise we register the compiled method for subsequent execution. (If there are inlined methods we also need to re-check their classes' modes before registering the compiled method.) The process of checking the mode again and registering the compiled method is atomic so there is no potential for time-of-check time-of-use errors.

Figure 11 provides pseudocode for the two versions of each compiled method. The *slow* version is simply the method with all fences added, as in the baseline volatile-by-default approach. The *fast* version first performs the safety check. If the method's receiver object is still *safe*, then its method body

```
slow_version(m) {
  vbd(m.body);
}

fast_version(m) {
  if (curr_thread == this.creator_thread)
    m.body;
  else
    switch_to_slow(this.class);
}

switch_to_slow(C) {
  critical_section_begin;
  if(C.mode == slow)
    return;
  C.mode = slow;
  deoptimize_to_slow(C);
  critical_section_end;
}
```

Figure 11. The *slow* and *fast* versions of a method.

```
interpreter_version(m) {
  if (this.class.mode == fast &&
      curr_thread != this.creator_thread) {
    switch_to_slow(this.class);
  }
  slow_version(m);
}
```

Figure 12. The interpreted version of a method.

is executed, without requiring any added fences. Otherwise, all compiled methods of `this`'s class must be invalidated to be recompiled in their *slow* versions.

The `switch_to_slow` pseudocode in the figure illustrates the latter process. We first change the mode of the given class `C` to *slow*. The `deoptimize_to_slow` function (definition not shown) then leverages the HotSpot JVM's existing mechanism for *deoptimization* to invalidate all compiled methods that *depend upon* `C`, which includes the methods of `C` and its superclasses, as well as any methods in which one of these methods is inlined. This function also changes the mode of all of `C`'s superclasses to *slow*. The `deoptimize_to_slow` function is implemented as a "VM operation" in the HotSpot JVM, which causes all other threads to be stopped before its execution so that it can safely invalidate compiled methods. Also, we make the `switch_to_slow` function shown in Figure 11 atomic to prevent multiple threads from deoptimizing the same methods and to prevent the `compile` function in Figure 10 from concurrently registering any *fast* methods for class `C`.

Finally, we describe modifications to the HotSpot interpreter. As in VBDA-HotSpot, the interpreter always includes the additional fences necessary to ensure the volatile-by-default semantics. However, we additionally must perform the check at the beginning of each method that the receiver object is *safe*, and if not then all compiled methods that depend on the object's class must be deoptimized. Pseudocode is shown in Figure 12.

3.3 Implementing Per-Access Checks

The above description assumed that all field accesses are of the form `this.f`, but Java allows field accesses to arbitrary objects (e.g., for fields that are declared `public`). For objects other than `this` S-VBD performs safety checks on a per-field-access basis. This subsection describes how such checks are implemented.

As mentioned earlier, at class-load time an intraprocedural analysis identifies field accesses whose receiver object is definitely `this`, so we can avoid checks on these accesses. The analysis also rewrites all other `getfield` bytecodes in the method to a new `check_getfield` bytecode that we have defined in S-VBD, and similarly for all other `putfields` in the method. Later, whenever a `check_getfield` bytecode is encountered during interpretation or compilation, we simply treat it as if it were an inlined call to a *getter* method on the receiver object. That is, we follow exactly the scheme shown in the previous subsection, except that the various checks are inlined into the method containing the `check_getfield` bytecode. Similarly, a `check_putfield` bytecode is treated as an inlined call to a *setter* method.

Making this approach work requires one addition to the scheme shown earlier. If a field of class `D` is accessed by method `m` of class `C`, then we must make sure to deoptimize `C.m` whenever class `D` is deoptimized. Otherwise, the compiled version of `C.m` will still be using the *fast* version of the field access even after `D` has been switched to *slow* mode. To do this we record a dependency of the method `C.m` on class `D` whenever we encounter such a field access, extending the dependency-tracking mechanism that HotSpot uses for deoptimization as described earlier.

3.4 Optimizing Fence Insertion

In addition to speculative compilation, we implemented an orthogonal optimization that reduces the number of fences required to enforce the volatile-by-default semantics for ARM. The first two rows of Table 1 show the memory barriers required before and/or after a `volatile` memory access in Java, as described in the JMM Cookbook [16], and the corresponding ARM instructions used to achieve those barriers in HotSpot. For example, a `volatile` load requires a LoadLoad and LoadStore barrier after it, which is implemented by a `dmb ish ld` instruction in ARM.

The baseline implementation of VBDA-HotSpot, which uses the approach of Liu et al. [20], simply inherits this implementation strategy for `volatiles` from HotSpot. However, we observe that some of the barriers are only there to prevent reorderings between `volatile` and *non*-`volatile` accesses. Hence in the volatile-by-default setting, where all accesses are treated as `volatile`, it is safe to eliminate some of these barriers, which in turn eliminates some unnecessary fence instructions in the generated code.

The last two rows in Table 1 show our optimized approach. The implementation of VBD loads is the same as that for `volatile` loads in Java. However, a VBD store does not require a preceding LoadStore fence, due to the LoadStore fence after each VBD load. Further, in place of the StoreStore fence that precedes a `volatile` store, it is equivalent in VBD to move this fence *after* each store, since there are no non-`volatile` stores. The result is that we have eliminated the need for any memory barriers before a VBD store. Further, while we have added a StoreStore barrier after a VBD store, the corresponding implementation of the required barriers in ARM remains the same, namely the use of a full fence `dmb ish`.

We implemented this optimized strategy, which we call `VBD-Opt`, in S-VBD. For the interpreter, we changed the barriers inserted as described above. For the server compiler, for simplicity of implementation we keep the original VBD design at the IR level, so compiler optimizations must respect all of the original memory barriers. However, during the code generation phase, we eliminate the `dmb ish` instruction before each store.

3.5 Performance Evaluation

3.5.1 DaCapo Benchmarks

We measured the peak performance of S-VBD for the DaCapo benchmarks using the same methodology as in the previous section. The fourth series in Figures 13 and 14 shows the overhead of our approach over the baseline HotSpot JVM on machine A and machine B. The geomean overhead of the S-VBD approach is respectively 51% and 37% for the two machines, which is a significant improvement over the geomean overheads of the original VBDA-HotSpot (the first series in the figures) at 73% and 57%. Also, the maximum overhead across all benchmarks respectively reduces from 129% to 78% and from 157% to 73%.

Figures 13 and 14 also isolate the effect of each of our optimizations: the second series shows the relative performance when using just speculative compilation, and the third series shows the relative performance when using just the VBD-Opt fence optimization. On its own each optimization provides a considerable performance improvement, but speculative compilation clearly is the more effective optimization. As the fourth series shows, together they are even

Table 1. The implementation for `volatile` accesses on ARM in HotSpot (first two rows). An optimized implementation for memory accesses on ARM in S-VBD (last two rows).

	Barriers Needed Before	Barriers Needed After	Aarch64 Instruction Sequences
`volatile` load	None	LoadLoad and LoadStore	`ldr` `dmb ish ld ; wait for load`

`volatile` store	LoadStore and StoreStore	StoreLoad	`dmb ish ; full fence`
			`str`
			`dmb ish ; full fence`

VBD Load	None	LoadLoad and LoadStore	`ldr` `dmb ish ld ; wait for load`
VBD Store	None	StoreLoad and StoreStore	`str` `dmb ish ; full fence`

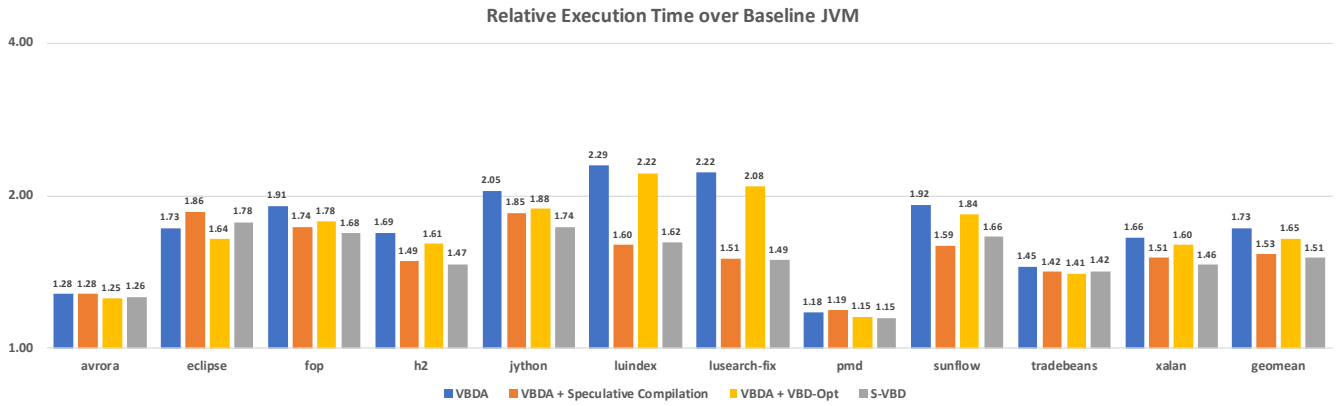

Figure 13. Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine A, y-axis in logarithmic scale.

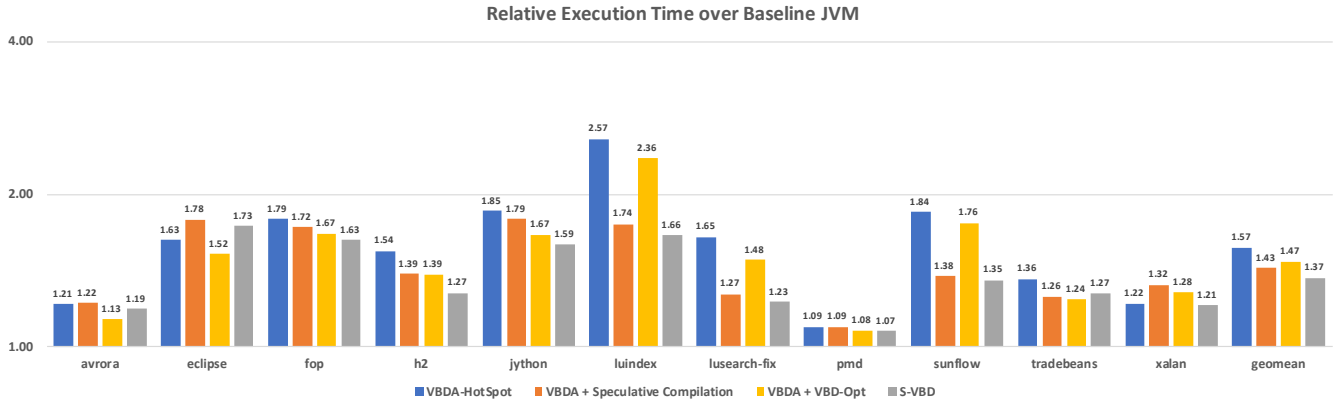

Figure 14. Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine B, y-axis in logarithmic scale.

more beneficial in terms of reducing the overhead of the volatile-by-default semantics.

Finally, speculative compilation's use of deoptimization is likely to impair startup performance. We measured the startup performance of both VBDA-HotSpot and S-VBD using the methodology described by Georges et al. [13]. We run n invocations of each benchmark, each time measuring the execution time of one iteration, until either the confidence interval for the sampled times is less than 2% of the

average execution time or until n is 30. We discard the first JVM invocation of each benchmark because it might change some system state such as dynamically loaded libraries or the data cache. Finally, we report the average execution time and confidence interval for each benchmark and calculate the relative execution time of each benchmark using these averages.

The relative startup performance of VBDA-HotSpot and S-VBD compared to the baseline HotSpot JVM is shown in Figure 15. The confidence interval of each benchmark is less than 5% of the average execution time after 30 invocations. As expected, the use of deoptimization causes S-VBD to have a significantly higher impact on startup performance than VBDA-HotSpot.

3.5.2 CheckOnly Overhead

To further understand the overheads of S-VBD, we implemented a *check-only* version, which performs all of the safety and mode checks as described above but never deoptimizes any methods. Note that this *check-only* version also keeps all barriers for array accesses and intrinsics. Figure 16 shows the relative execution time of this version versus the baseline HotSpot JVM on machine A and machine B. The experiment shows that the cost of the checks required by the speculative approach is considerable, on its own incurring well over half of the overhead incurred by S-VBD. These results also validate the *thread-local hypothesis* that underlies our speculative compilation technique. Specifically, the large overhead of the checks implies that the overhead due to fences on field accesses is relatively modest, meaning that the thread-locality hypothesis is effective at removing many fences.

3.5.3 Spark Benchmarks

We also measured the peak performance of S-VBD for the `spark-perf` benchmarks using the same methodology as in the previous section. Figure 17 summarizes the results of `spark-tests` and `mllib-tests` in a histogram. The geometric mean of S-VBD's relative execution time on Machine A and Machine B is 2.01 and 1.86 respectively, representing a 101% and 86% overhead over the baseline HotSpot JVM. Comparing these results to the ones for VBDA-HotSpot from Figure 9 we see that our speculative compilation strategy provides little speedup for these benchmarks. We suspect this is due to the fact mentioned earlier that these benchmarks have many array accesses. Since S-VBD does not speculate on array accesses it incurs the same cost as VBD-HotSpot for these accesses.

4 Related Work

Language-Level Sequential Consistency In earlier work we proposed the volatile-by-default semantics for achieving SC for Java and evaluated it on Intel x86 hardware [20]. The current work is, to the best of our knowledge, the first comprehensive study of the cost of providing SC for any language on ARM, which is a much weaker memory model than x86. We also propose a novel, speculative approach to implementing volatile-by-default as well as an optimized choice of fences for the volatile-by-default semantics. We show that these optimizations are effective in reducing the overhead of SC on weak hardware.

Vollmer et al. [39] implement the SC semantics for the Haskell programming language and demonstrate negligible overheads on x86. They also demonstrated low overheads for some benchmarks on ARM but did not do an extensive study due to the limited portability of Haskell libraries. The key takeaway is that a pure, functional programming language like Haskell naturally limits conflicting memory accesses among threads and so can support SC with low overhead. As such, these results do not extend to imperative languages like Java.

We evaluated the volatile-by-default semantics in a production JVM with modern features such as dynamic class loading and just-in-time compilation. In contrast, prior work has evaluated the cost of SC for Java in the context of an offline whole-program compiler, which admits more opportunities for optimization but is incompatible with modern JVMs. Shasha and Snir [35] propose a whole-program delay-set analysis for determining the barriers required to guarantee SC for a given program. Sura et al. [38] implement this technique for Java and Kamil et al. [17] do the same for a parallel variant of Java called Titanium. These works demonstrate low performance overhead for SC on both x86 and POWER. Alglave et al. [2] implemented SC for C programs similarly.

Other work has achieved language-level SC guarantees for Java [1, 9] and for C [25, 37] through a combination of compiler modifications and specialized hardware. These works show that SC can be comparable in efficiency to weak memory models with appropriate hardware support. The technique of Singh et al. [37] is similar to our speculative approach in identifying safe and unsafe memory accesses. However, they rely on specialized hardware as well as operating system support to perform the speculation, while we speculate purely at the JVM level. Finally, several works demonstrate testing techniques to identify errors in Java and C code that can cause non-SC behavior (e.g., [11, 14]).

Language-Level Region Serializability Other work strives to efficiently provide stronger guarantees than SC for programming languages through a form of *region serializability*. In this style, the code is implicitly partitioned into disjoint regions, each of which is guaranteed to execute atomically. Therefore SC is a special case of region serializability where each memory access is in its own region. Several works have explored a form of region serializability for Java [6, 33, 34, 42]. These approaches are implemented in the Jikes research virtual machine [3] and evaluated only on x86. Work on region

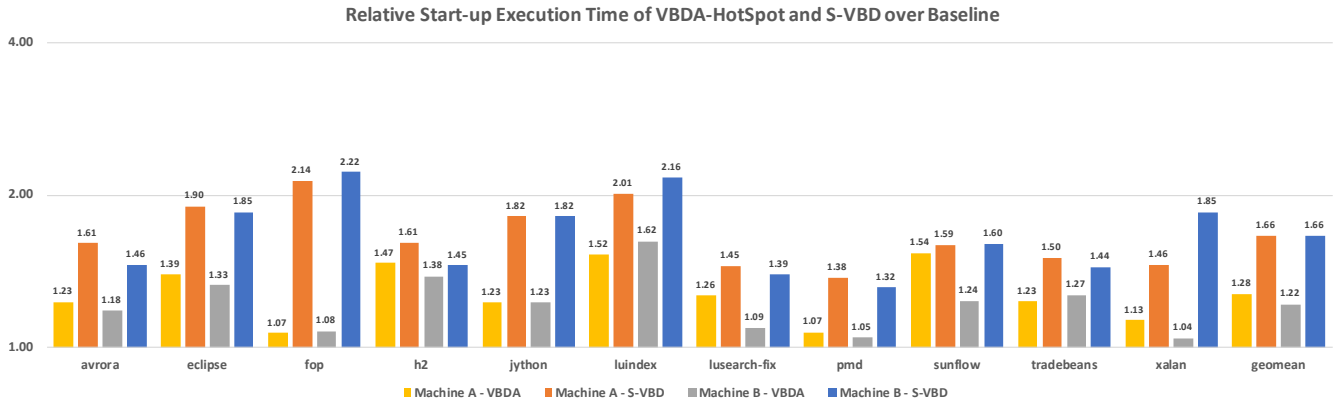

Figure 15. Relative startup execution time of VBDA-HotSpot and S-VBD over the baseline JVM, y-axis in logarithmic scale.

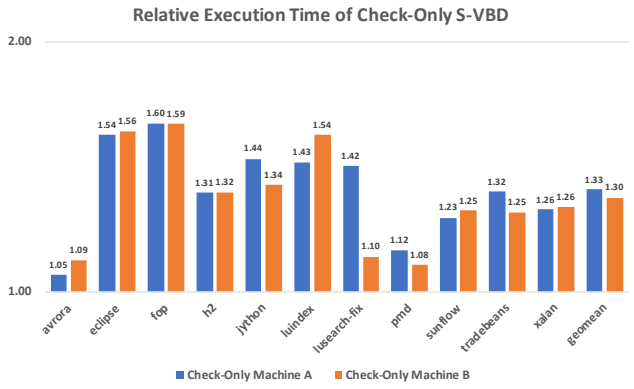

Figure 16. Relative execution time of check-only S-VBD over baseline JVM on machine A and machine B, y-axis in logarithmic scale.

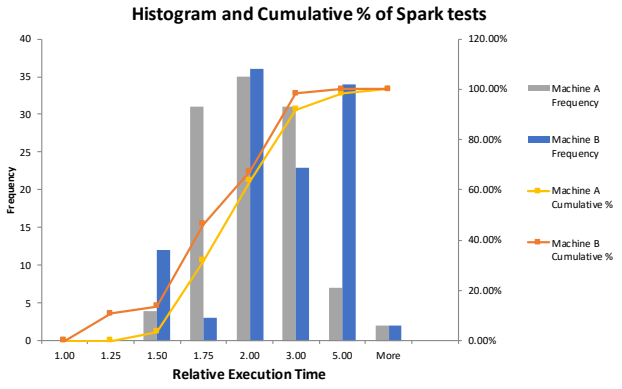

Figure 17. Histogram and cumulative % of relative execution time of VBDA-HotSpot for Spark benchmarks.

serializability for C has achieved good performance either through special-purpose hardware [21, 24, 36] or by requiring $2N$ cores to execute an application with N threads [27].

Memory Model Safety The notion of "safety" in the JMM disallows out-of-thin-air values [22], but it has proven difficult to ensure while also admitting desired optimizations [5]. Several recent works have defined new memory models that attempt to resolve this tension [8, 15, 18, 19, 26, 28]. Many of these works formalize the new memory model along with compilation strategies to common hardware platforms, allowing them to prove properties such as the absence of thin-air reads. To our knowledge only the work by Ou and Demsky [26] provides an empirical evaluation; they demonstrate low overheads for C/C++ programs running on ARM hardware. Our work adopts and empirically evaluates a significantly stronger notion of safety for Java than these works [23], as it additionally preserves the program order of instructions and the atomicity of primitive types.

Weak Memory Model Performance for Java Demange et al. [10] define an x86-like memory model for Java. They present a performance evaluation that uses the Fiji real-time virtual machine [29] to translate Java code to C, which is then compiled with a modified version of the LLVM C compiler [25] and executed on x86 hardware. Ritson and Owens [30] modified the HotSpot compiler's code-generation phase for both ARM and POWER to measure the cost of different instruction sequences to implement the JMM.

5 Conclusion and Future Work

Any proposed language-level memory model must be efficiently implementable on ARM in order to be viable, due to ARM's popularity and its weak memory model. In this paper we have performed the first comprehensive study of the cost of providing language-level sequential consistency for a production compiler on ARM. Our experiments show that an existing technique that provides the volatile-by-default semantics for Java on Intel x86 hardware at modest cost in fact incurs considerable overhead on ARM. Motivated by these experimental results, we then present a novel *speculative*

technique to optimize language-level SC and demonstrate its effectiveness in reducing the overhead of enforcing volatile-by-default. To our knowledge this is the first optimization approach for language-level SC that is compatible with modern implementation technology, including dynamic class loading and just-in-time (JIT) compilation.

Longer-term, the goal of this line of research is to make the performance of strong language-level memory models like volatile-by-default competitive even on weak hardware. Based on our results, promising avenues of future work include techniques to reduce the cost of S-VBD's speculation checks and techniques to optimize fence insertion for array accesses.

Acknowledgments

We thank our shepherd John Wickerson and the other reviewers for their constructive feedback. We also thank an anonymous reviewer on an earlier version of this paper for pointing out the bug in the HotSpot interpreter's treatment of `volatile` writes. We are grateful to the Works on ARM team, especially Edward Vielmetti, for setting up and providing access to an ARM server (machine B) and to Xiwei Ma for help implementing our litmus tests. This work is supported in part by the National Science Foundation awards CCF-1527923 and CNS-1704336.

References

[1] Wonsun Ahn, Shanxiang Qi, Jae-Woo Lee, Marios Nicolaides, Xing Fang, Josep Torrellas, David Wong, and Samuel Midkiff. 2009. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *42nd International Symposium on Microarchitecture*.

[2] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *Computer Aided Verification - 26th International Conference*. 508–524.

[3] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–418.

[4] ARMv8 2018. ARM Cortex-A Series Programmer's Guide for ARMv8-A Version: 1.0, Section 13.2.1. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CHDCJBGA.html Accessed April 2018.

[5] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 283–307.

[6] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-only Region Conflict Exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, 241–259.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190.

[8] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, Article 7, 6 pages.

[9] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proc. of the 34th Annual International Symposium on Computer Architecture*. 278–289.

[10] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. 2013. Plan B: A Buffered Memory Model for Java. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 329–342.

[11] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial Memory for Detecting Destructive Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 244–254.

[12] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '10)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[13] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, 57–76.

[14] Mohammad Majharul Islam and Abdullah Muzahid. 2016. Detecting, Exposing, and Classifying Sequential Consistency Violations. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 241–252. https://doi.org/10.1109/ISSRE.2016.48

[15] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 759–767. https://doi.org/10.1145/2933575.2934536

[16] JSR133 2018. JSR-133 Cookbook for Compiler Writers. Accessed November 2018. http://g.oswego.edu/dl/jmm/cookbook.html

[17] A. Kamil, J. Su, and K. Yelick. 2005. Making sequential consistency practical in Titanium. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society.

[18] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 175–189.

[19] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[20] Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A Volatile-by-default JVM for Server Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 49 (Oct. 2017), 25 pages.

[21] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions. In *Proc. of the 37th Annual International Symposium on Computer Architecture*.

[22] J. Manson, W. Pugh, and S. Adve. 2005. The Java memory model. In *Proceedings of POPL*. ACM, 378–391.

[23] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. 2015. The Silently Shifting Semicolon. In *1st Summit on Advances in Programming Languages*

(SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs)), Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. 177–189.

[24] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI '10*. ACM, 351–362.

[25] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[26] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-thin-air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

[27] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. ...And Region Serializability for All. In *5th USENIX Workshop on Hot Topics in Parallelism, HotPar'13*, Emery D. Berger and Kim M. Hazelwood (Eds.). USENIX Association.

[28] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

[29] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. 2010. High-level Programming of Embedded Hard Real-time Devices. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. 69–82.

[30] Carl G. Ritson and Scott Owens. 2016. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Article 24, 11 pages.

[31] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[33] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 561–575.

[34] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2015. Toward Efficient Strong Memory Model Support for the Java Platform via Hybrid Synchronization. In *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015*, Ryan Stansifer and Andreas Krall (Eds.). ACM, 65–75.

[35] D. Shasha and M. Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 282–312.

[36] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. 2011. Efficient processor support for DRFx, a memory model with exceptions. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 53–66.

[37] Abhayendra Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. 2012. End-to-end Sequential Consistency. In *Proc. of the 39th Annual International Symposium on Computer Architecture*. 524–535.

[38] Z. Sura, X. Fang, C.L. Wong, S.P. Midkiff, J. Lee, and D. Padua. 2005. Compiler techniques for high performance sequentially consistent Java programs. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2–13.

[39] Michael Vollmer, Ryan G. Scott, Madanlal Musuvathi, and Ryan R. Newton. 2017. SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, 283–298.

[40] James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: A Verified, High-performance Precise Dynamic Race Detector. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 354–367. https://doi.org/10.1145/3178487.3178514

[41] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[42] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. 2017. Avoiding Consistency Exceptions Under Strong Memory Models. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, 115–127.