NOTE: After the publication of this paper in OOPSLA 2017, we found a bug in our test configuration for the `spark-perf` benchmarks that caused incorrect flags to be used for some JVM invocations. As a result the published performance numbers for these benchmarks under-count the performance cost of our technique. (The published numbers for other benchmarks are accurate.) We have fixed the bug and rerun all `spark-perf` tests, and the corrected results are in the appendix that we have included at the end of this document.

A Volatile-by-Default JVM for Server Applications

LUN LIU, University of California, Los Angeles, USA
TODD MILLSTEIN, University of California, Los Angeles, USA
MADANLAL MUSUVATHI, Microsoft Research, Redmond, USA

A *memory consistency model* (or simply *memory model*) defines the possible values that a shared-memory read may return in a multithreaded programming language. Choosing a memory model involves an inherent performance-programmability tradeoff. The Java language has adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The downside, however, is that programmers are exposed to a complex and unintuitive semantics and must carefully declare certain variables as `volatile` in order to enforce program orderings that are necessary for proper behavior.

This paper proposes a simpler and stronger memory model for Java through a conceptually small change: *every* variable has `volatile` semantics by default, but the language allows a programmer to tag certain variables, methods, or classes as `relaxed` and provides the current Java semantics for these portions of code. This *volatile-by-default* semantics provides *sequential consistency* (SC) for all programs by default. At the same time, expert programmers retain the freedom to build performance-critical libraries that violate the SC semantics.

At the outset, it is unclear if the `volatile`-by-default semantics is practical for Java, given the cost of memory fences on today's hardware platforms. The core contribution of this paper is to demonstrate, through comprehensive empirical evaluation, that the `volatile`-by-default semantics is arguably acceptable for a predominant use case for Java today — server-side applications running on Intel x86 architectures. We present VBD-HotSpot, a modification to Oracle's widely used HotSpot JVM that implements the `volatile`-by-default semantics for x86. To our knowledge VBD-HotSpot is the first implementation of SC for Java in the context of a modern JVM. VBD-HotSpot incurs an average overhead versus the baseline HotSpot JVM of 28% for the Da Capo benchmarks, which is significant though perhaps less than commonly assumed. Further, VBD-HotSpot incurs average overheads of 12% and 19% respectively on standard benchmark suites for big-data analytics and machine learning in the widely used Spark framework.

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

Additional Key Words and Phrases: memory consistency models, volatile by default, sequential consistency, Java virtual machine

ACM Reference Format:
Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A Volatile-by-Default JVM for Server Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 49 (October 2017), 25 pages. https://doi.org/10.1145/3133873

Authors' addresses: 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.

2475-1421/2017/10-ART49
https://doi.org/10.1145/3133873

1 INTRODUCTION

A *memory consistency model* (or simply *memory model*) defines the possible values that a shared-memory read may return and thus plays a central role in the semantics of a multithreaded programming language. Choosing a memory model for a language involves an inherent performance-programmability tradeoff. *Sequential consistency* (SC) [Lamport 1979] provides an intuitive programming model by ensuring that a program's instructions (appear to) execute in a global total order consistent with the per-thread program order. But achieving SC requires the compiler to disable some common optimizations, and on current hardware platforms it further requires the compiler to restrict hardware optimizations by emitting expensive fence instructions on shared-memory accesses.

To avoid this cost, the Java language [Manson et al. 2005] has adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The downside, however, is that programmers must carefully declare certain variables as `volatile` in order to enforce the per-thread program orderings that are required for proper behavior. If a programmer does not annotate the necessary variables as `volatile`, programs are exposed to the relaxed semantics of the Java memory model (JMM), which is complex, unintuitive, and can violate critical program invariants. For example, under the JMM an object can be accessed before its constructor has completed, accesses to `long`s and `double`s are not guaranteed to be atomic, and some programming idioms, such as double-checked locking [Bacon et al. 2017; Schmidt and Harrison 1997], are not guaranteed to behave correctly. As discussed in Section 3, this possibility is not just theoretical: errors due to the failure to annotate certain variables as `volatile`, which we term *missing-annotation bugs*, can be found across a range of widely used Java applications.

This *performance-by-default* approach to memory models is acceptable for and consistent with the design philosophy of languages like C and C++ [Boehm and Adve 2008]. However, we argue that such an approach conflicts with the design philosophy of "safe" languages like Java. Indeed, when it comes to type and memory safety, Java instead employs a *safe-by-default* and *performance-by-choice* approach: type and memory safety are provided by default, but there is a library of type-unsafe operations meant for use by expert programmers in performance-critical code [Mastrangelo et al. 2015].

In the same vein, this paper proposes a *safe-by-default* and *performance-by-choice* approach to enforcing per-thread program order in Java. This approach involves a conceptually simple change to the memory model: every variable has `volatile` semantics by default, but the language allows a programmer to tag certain classes, methods, or variables as `relaxed` and provides the current JMM semantics for these portions of code. This *volatile-by-default* semantics provides a natural form of sequential consistency for Java by default, at the bytecode level: bytecode instructions (appear to) execute atomically and in program order. This also implies that all Java primitive values, including (64-bit) `double`s and `long`s, are atomic irrespective of the bit-width of the underlying architecture. At the same time, expert programmers retain the freedom to build performance-critical libraries that violate this semantics, and they are responsible for protecting clients from any weak behaviors that can result.

A `volatile`*-by-default JVM.* At the outset, it is unclear if the `volatile`-by-default semantics is practical for Java, given the cost of memory fences on today's hardware platforms. We should deem the `volatile`-by-default approach as unacceptable if the only way to make programs reasonably efficient is to declare large portions as `relaxed`. In fact, current conventional wisdom about the cost of these fences and the associated cost of providing SC to the programmer strongly points against the `volatile`-by-default approach. For instance, Kaiser et al. [2017] say that SC is "woefully

unrealistic" due to "the significant performance costs ... on modern multi-core architectures" and Demange et al. [2013] say that "SC would likely cripple performance of Java programs on all modern microprocessors." The core contribution of this paper is to demonstrate, through comprehensive empirical evaluation, that the `volatile`-by-default semantics is in fact acceptable for a predominant use case for Java today — server-side Java applications running on Intel x86 architectures.

The straightforward way to implement the `volatile`-by-default semantics is through a source-to-source translation that adds the appropriate `volatile` annotations. The advantage of this approach is that it is *independent* of the JVM, allowing us to evaluate the cost of `volatile`-by-default semantics on various JVMs and hardware architectures. Unfortunately, neither Java nor the Java bytecode language provides a mechanism to declare array elements as `volatile`. Thus, such an approach fails to provide the desired semantics. We considered performing a larger-scale source-to-source rewrite on array accesses, but it would be difficult to separate the cost of this rewrite from the measured overheads. Finally, once we settled on changing an existing JVM implementation, we considered doing so in a research virtual machine [Alpern et al. 2005]. But it was not clear how the empirical observations from such a JVM would translate to a production JVM implementation.

Therefore we implement the `volatile`-by-default semantics for x86 by directly modifying Oracle's HotSpot JVM which is part of the OpenJDK version 8u [OpenJDK 2017]. We call our modified version VBD-HotSpot. To the best of our knowledge, this is the first implementation of SC for a production Java virtual machine (JVM) that includes the state-of-the-art implementation technology, such as dynamic class loading and just-in-time (JIT) compilation. This in turn enables us to provide the first credible experimental comparison between SC and the JMM.

We implemented VBD-HotSpot by reusing mechanisms already in place for handling `volatile` variables in the interpreter and compiler. This provides us two advantages. First and foremost, the correctness of VBD-HotSpot mostly follows from the correctness of the HotSpot JVM's implementation of `volatile` semantics. Second, we automatically obtain the benefits of optimizations that HotSpot already employs to reduce the overhead of `volatile` accesses. VBD-HotSpot is open-source and available for download at https://github.com/SC-HotSpot/VBD-HotSpot.

Results. An advantage of modifying the JVM is that the `volatile`-by-default guarantees extend beyond Java to other languages that target the Java bytecode language and commonly run on the HotSpot JVM, such as Scala. We make use of this benefit by benchmarking `volatile`-by-default on the traditional DaCapo benchmarks [Blackburn et al. 2006] as well as a set of big-data analytics and machine learning benchmarks that run on the widely used Spark framework [Zaharia et al. 2016].

For the DaCapo benchmarks on a modern server, with no `relaxed` annotations, the overhead of VBD-HotSpot versus the unmodified HotSpot JVM is 28% on average, with a maximum of 81%. Given that VBD-HotSpot potentially inserts a fence on *every* heap store, we believe that this overhead is less than commonly assumed. Our experiments show that the benchmarks that incur the highest overheads are either single-threaded or mostly-single-threaded. Excluding these benchmarks reduces the average overhead to 21%.

For the Spark benchmarks, the results are even more striking. Our evaluation shows that the overhead is 12% on average for big-data analytics benchmarks and 19% on average for the machine-learning benchmarks, again without any `relaxed` annotations. From profiling these benchmarks we see that they tend to spend a lot of time doing I/O and also waiting at barriers (e.g., between a *map* task and a *reduce* task), which masks the extra latency due to fences.

Another common assumption is that the cost of SC increases with the number of processor cores and sockets. For instance, a fence on x86 stalls the processor until it has received read invalidations from all processors that have a shared copy of addresses in its pending store buffer. One expects these stalls to be longer when there are more threads in the program, and when it is running across

a larger number cores and sockets. However, our evaluation shows that this assumption is not true, at least in our experimental setting. The overhead of SC *improves* with increased concurrency and with the number of sockets; apparently the increased cost of fences is compensated by the increased cost of regular memory accesses.

Finally, we added `relaxed` annotations to some benchmarks in order to reduce the overhead of VBD-HotSpot. Our experiments on the benchmarks with the largest overheads for VBD-HotSpot show that roughly half of their overheads arise from 20 or fewer methods. Moreover, many of these methods are in standard libraries, such as `java.io` and string-manipulation libraries. These are good candidates for `relaxed` annotations, which experts can introduce after careful inspection and will benefit many applications.

What about ARM? A natural question is whether the results above translate to the other important use case of Java today — mobile applications running on low-power ARM devices. ARM's memory model is weaker than x86 and as such our experiments likely underestimate the cost of `volatile`-by-default semantics on ARM. Specifically, our experiments on x86 capture the cost of disabling non-SC compiler optimizations and the cost of inserting fences on regular stores. On the other hand, our experiments do not capture the cost of converting regular reads into `volatile` reads, as the latter comes free on x86. As reads are more prevalent than writes, one should expect `volatile`-by-default to be more expensive on ARM.

Providing a credible experimental evaluation for ARM is beyond the scope of this paper, as it requires a significant additional engineering effort. The main HotSpot JVM, which we modified to produce VBD-HotSpot (and which required 9 person-months), does not have an ARM backend. The OpenJDK contains a separate project with an ARM port of HotSpot. However, as future work we instead plan to modify the Android runtime's ahead-of-time bytecode compiler[1], which is the dominant Java platform on ARM devices today.

Finally, it is worth noting that ARM recently introduced "one-way" barrier instructions [ARMv8 2017] primarily to reduce the cost of `volatile` accesses in Java (and similar `atomic` accesses in C/C++). Speculation-based techniques for implementing the `LDAR` instruction, the one-way barrier required for `volatile` reads, are well known [Gharachorloo et al. 1991] and have been used in Intel architectures for decades. Thus, if ARM employs these optimizations to make `volatile` reads more efficient, the difference in the cost of VBD-HotSpot on x86 and VBD-HotSpot on ARM will reduce.

Summary. This paper makes the following contributions:

- We propose a `volatile`-by-default semantics for Java.
- We have implemented this semantics for Intel x86 as a modification of the HotSpot Java Virtual Machine. To our knowledge this is the first implementation of SC for Java in a production virtual machine.
- Our experimental evaluation provides a performance comparison between `volatile`-by-default and the Java memory model in the context of the HotSpot JVM. We show that the `volatile`-by-default semantics can be a practical choice today for common server-side applications on x86, such as big-data analytics.

The rest of the paper is structured as follows. Section 2 overviews the current Java memory model, and Section 3 describes the class of missing-annotation bugs that can occur under this memory model due to insufficient `volatile` annotations. Section 4 motivates and defines our proposed `volatile`-by-default semantics, and Section 5 describes our implementation of the `volatile`-by-default semantics for x86 in VBD-HotSpot. Section 6 presents the results of an experimental

[1] https://source.android.com

comparison of VBD-HotSpot with the unmodified HotSpot JVM. Section 7 compares with related work, and Section 8 concludes.

2 THE JAVA MEMORY MODEL

The Java memory model was defined more than a decade ago [Manson et al. 2005] and attempts to strike a practical balance among programmer understandability, implementation flexibility, and program safety.

programmer understandability The JMM designers considered sequential consistency to be "a simple interface" and "the model that is easiest to understand" [Manson et al. 2005]. However, largely due to SC's incompatibility with standard compiler and hardware optimizations, the JMM adopts a weak memory model based on the DRF0 style [Adve and Hill 1990], whereby SC is only guaranteed for *data-race-free* programs. A *memory race* occurs when two threads concurrently access the same memory location and at least one of those accesses is a write. A program is considered to be data-race-free if all instance variables that participate in a memory race are declared `volatile`.[2] The JMM guarantees SC semantics for `volatile` variables, which requires implementations to disable many compiler optimizations and to emit fence instructions that prevent the hardware from violating SC through out-of-order execution.

implementation flexibility The SC memory model does not allow instructions to appear to be reordered. However, several important optimizations, for example out-of-order execution in hardware and common subexpression elimination in compilers, have the effect of instruction reordering. By guaranteeing SC only for data-race-free programs, the JMM can admit most traditional optimizations.

program safety The JMM strives to ensure safety for all programs, even ones with data races. The JMM's notion of program safety is centered around the idea of preventing "out-of-thin-air reads" [Manson et al. 2005]. In the presence of data races, some compiler optimizations can in principle lead to a situation whereby multiple program transformations justify one another in a cycle. Such transformations can introduce values in the program that would never otherwise occur, which creates a potentially serious security concern. The JMM prevents out-of-thin-air reads by defining a complex *causality* requirement on the legal executions of incorrectly synchronized programs, which imposes some restrictions on the optimizations that a compiler may perform [Manson et al. 2005].[3]

Because the JMM guarantees SC for data-race-free programs, programmers "need only worry about code transformations having an impact on their programs' results if those programs contain data races" [Manson et al. 2005]. However, data races are both easy to introduce and difficult to detect; it is as simple as forgetting to grab a lock, grabbing the wrong lock, or omitting a necessary `volatile` annotation. Therefore in practice many programs are exposed to the effects of compiler and/or hardware optimizations, which can cause a variety of surprising behaviors and violate critical program invariants:

non-atomic primitives Writes to `doubles` and `longs` are not atomic under the JMM, but rather are treated as two separate 32-bit writes. Therefore, in the presence of a data race readers can see values that are a combination of two different writes. Understanding this to be

[2]Local variables are thread-local and hence can never participate in a memory race.

[3]The JMM's causality is known to disallow some optimizations that it was intended to allow, notably common subexpression elimination [Cenciarelli et al. 2007; Sevcík and Aspinall 2008]. Nonetheless, current Java virtual machines continue to perform this optimization. While there is no evidence that today's JVMs in fact admit out-of-thin-air reads, this issue must be resolved to prevent the possibility in the future.

problematic, the Java Language Specification states that "implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible."[4]

partially-constructed objects Consider the following example, where one thread tries to safely *publish* an object to another thread (assuming `d` and `ready` are respectively initialized to `null` and `false`):

Thread 1	Thread 2
`d = new Data();`	`if (ready)`
`ready = true;`	`  d.use();`

Under the JMM, it is possible for the second thread to read the value `true` for `ready` but incur a null pointer exception on the call `d.use()`. More perniciously, `d` may be non-null at that point but its constructor may not yet have completed, so the object is in an arbitrary state of partial construction when `use` is invoked.[5]

broken synchronization idioms The publication idiom above is one example of a custom synchronization idiom that is not guaranteed to work as expected in the JMM in the presence of data races. Other examples include double-checked locking [Schmidt and Harrison 1997] and Dekker's mutual exclusion algorithm.

3 MISSING-ANNOTATION BUGS

The misbehaviors above are instances of what we call *missing-annotation bugs*. In these examples, the synchronization protocol intended by the programmer is correct and need not be changed. Rather, the error is simply that the programmer has forgotten to annotate certain variables as `volatile`. This omission allows compiler and hardware optimizations to violate intended program invariants. Adding `volatile` annotations forces the Java implementation to disable those optimizations and thereby restore the desired invariants. For example, a `double` or `long` field that is declared `volatile` will have atomic reads and writes. Similarly, declaring `ready` as `volatile` in our publication idiom above ensures that the second thread will only ever see a fully constructed object.

Missing-annotation bugs are easy to make and hence it is not surprising that they are common in Java applications. A quick search on the Apache Software Foundation's issue-tracking system found more than 100 issues where the fix required annotating a field as `volatile`. We report the first twenty here: AMQ-6251, AMQ-6495, BOOKKEEPER-848, CASSANDRA-11984, CASSANDRA-2490, FELIX-4586, HBASE-15261, HDFS-1207, HDFS-4106 , HDFS-566, HTTPCLIENT-594 , KAFKA-2112, KAFKA-4232, LOG4J2-247, LOG4J2-254, OAK-3638, OAK-4294, SLING-1144, SLING-3793, SPARK-3101, SPARK-6059.[6] These errors occur in popular systems such as the Cassandra database, the HDFS distributed file system, and the Spark system for big-data processing[7] and can thereby impact the applications that employ these systems.

Missing-annotation bugs contrast with data races that violate program invariants due to the programmer's failure to implement the necessary synchronization protocol. Examples of the latter errors include atomicity violations that arise from failing to hold a lock or holding the wrong lock, and ordering violations that arise when threads fail to signal one another properly. Yet as Boehm [2011] points out, missing-annotation bugs are far from "benign" but rather can cause surprising and harmful behaviors. For instance, exposing a partially constructed object, as shown above, can have serious security implications [TSM03-J 2017]. The `volatile`-by-default semantics discussed below eliminates all missing-annotation bugs.

[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

[5] Java does guarantee that the `final` fields of `d` will be properly initialized when `use` is invoked.

[6] Each bug contains the project name and the bug ID. Its details can be found at https://issues.apache.org/jira/browse/<ProjectName>-<BugID>.

[7] Spark is implemented in Scala, which compiles to Java bytecode and inherits Java's memory model.

4 VOLATILE-BY-DEFAULT SEMANTICS FOR JAVA

Under the JMM, the onus is on programmers to employ the `volatile` annotation everywhere that is necessary to protect the program from compiler and hardware optimizations that can reorder instructions. By doing so, the JMM can allow most compiler and hardware optimizations. We argue that this *performance-by-default* approach is not consistent with Java's design philosophy as a "safe" programming language. Instead we advocate a *safe-by-default* and *performance-by-choice* approach to Java's concurrency semantics.

To that end, we propose the `volatile`-by-default semantics for Java, which makes one conceptually simple change: all instance variables are treated as if they were declared `volatile`. In this way, missing-annotation bugs cannot occur and all Java programs are guaranteed SC semantics by default. With this change, the `volatile` annotation becomes semantically a no-op. Instead, we introduce a `relaxed` annotation that allows a programmer to tag variables, methods, or classes that should employ the current JMM semantics. Expert programmers can use this annotation in performance-critical code to explicitly trade off program guarantees for increased performance.

Precisely defining the SC semantics requires one to specify the granularity of thread interleaving, which has been identified as a weakness of the SC memory model [Adve and Boehm 2010]. The `volatile`-by-default semantics does this in a natural way by providing SC at the bytecode level: bytecode instructions (appear to) execute atomically and in program order. This also implies that all Java primitive values, including (64-bit) `doubles` and `longs`, are atomic irrespective of the bit-width of the underlying architecture. The `volatile`-by-default semantics provides a clean way for programmers to understand the possible behaviors of their concurrent programs, provided they understand how Java statements (such as increments) are translated to bytecode.

Of course, "safety" is in the eye of the beholder, and there are many possible definitions. We argue that the `volatile`-by-default semantics is a natural *baseline* guarantee that a "safe" language should provide for all programs. The `volatile`-by-default memory model clearly satisfies the JMM's desired programmability and safety goals. In terms of programmability, `volatile`-by-default is strictly stronger than the JMM, so all program guarantees provided by the JMM are also provided by `volatile`-by-default. In terms of safety, the `volatile`-by-default semantics prevents all cyclic dependencies and hence rules out the particular class of such cycles that can cause out-of-thin-air reads. Moreover, `volatile`-by-default eliminates all missing-annotation bugs.

Further, the `volatile`-by-default semantics provides a more general notion of safety by protecting several fundamental program abstractions [Marino et al. 2015]. First, all primitives are accessed atomically. Second, sequential reasoning is valid for all programs. This ensures, for example, that an object cannot be accessed until it is fully constructed (unless the program explicitly leaks `this` during construction), and more generally that program invariants that rely on program order are guaranteed regardless of whether the program has data races.

Finally, we note that though the `volatile` keyword is semantically a no-op in the `volatile`-by-default semantics, it is still useful as a means for programmers to document their intention to use a particular variable for synchronization. Indeed, `volatile` annotations can make the code easier to understand and can be used by tools to identify potential concurrency errors. However, under the `volatile`-by-default semantics, and in sharp contrast to the JMM, an accidental omission or misapplication of `volatile` annotations will *never* change program behavior.

5 A VOLATILE-BY-DEFAULT JVM FOR X86

Our original idea was to implement the `volatile`-by-default semantics for Java through a simple transformation on either Java source code or Java bytecode to add `volatile` annotations. However, ensuring SC requires that all heap accesses be treated as `volatile`, which in Java includes both

instance variables and array elements. Unfortunately neither Java nor its bytecode language provides a way to declare array elements to be `volatile` — declaring an array to be `volatile` ensures SC semantics for accesses to the array itself but not to its elements.

Therefore, we opted to instead implement the `volatile`-by-default semantics through a direct modification to the Java virtual machine, which executes Java bytecode instructions. We chose to modify Oracle's HotSpot JVM, which is widely used and part of the OpenJDK — the official reference implementation of Java SE. In particular we modified the version of HotSpot that is part of the OpenJDK 8u, which is the latest stable version [OpenJDK 2017]. Our modified version, called VBD-HotSpot, adds a flag `-XX:+VBD` that allows users to obtain `volatile`-by-default semantics.

5.1 HotSpot Overview

The HotSpot JVM is an implementation of the Java Virtual Machine Specification [Java Virtual Machine Specification 2017]. To execute Java bytecode instructions, HotSpot employs just-in-time (JIT) compilation. In this style, bytecodes are first executed in interpreter mode, with minimal optimizations. During execution, HotSpot identifies parts of the code that are frequently executed ("hot spots") and compiles them to optimized native code for better performance.

The HotSpot JVM has one interpreter, but its behavior depends on the underlying hardware platform being used. HotSpot includes two just-in-time compilers. The *client* compiler, also called C1, is fast and performs relatively few optimizations. The *server* compiler, also called C2 or *opto*, optimizes code more aggressively and is specially tuned for the performance of typical server applications.

We have implemented the `volatile`-by-default semantics for Java server applications on x86, which are a dominant use case in practice. Accordingly we have modified the HotSpot interpreter when executing on x86 hardware as well as the HotSpot server compiler.

5.2 Volatile-by-Default 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 (and hence platform-specific). The `TemplateTable` is used at JVM startup time to create an interpreter in memory, whereby each bytecode is simply an index into the `TemplateTable`.

Figure 1 illustrates how the template-based interpreter works. The bytecode pointer (BCP) is currently pointing at the bytecode `putfield` (181). The interpreter uses this bytecode as an index in the `TemplateTable` (right side of the figure) to find the address of the corresponding template. The interpreter then jumps to this address to begin executing the template (left side of the figure). After writing to the field, the last four lines of the template show how the interpreter reads the next BCP index, increments the BCP, and jumps to the next code section. In this example, we add 3 to BCP (`%r13`) to point to the next bytecode, because the length of the `putfield` instruction is 3 bytes: a 1-byte opcode and a 2-byte index representing the field.

Figure 1 also shows how the HotSpot JVM handles accesses to `volatile` variables. The SC semantics for `volatile` accesses is achieved by inserting the appropriate platform-specific fences before/after such accesses. In the case of x86, which has the relatively strong *total store order* (TSO) semantics [Owens et al. 2009], a `volatile` read requires no fences and a `volatile` write requires only a subsequent *StoreLoad* barrier, which ensures that the write commits before any later reads [JSR133 2017]. In the figure, %edx is already loaded with the field attribute for `volatile`. Instruction (1) tests if the field is declared `volatile`. If so, the `lock` instruction (2) will be executed, which acts as a StoreLoad barrier on x86; otherwise the `lock` instruction is skipped.

To implement our `volatile`-by-default semantics for x86, we therefore modified the template for `putfield` to unconditionally execute the `lock` instruction. This is done by removing instruction

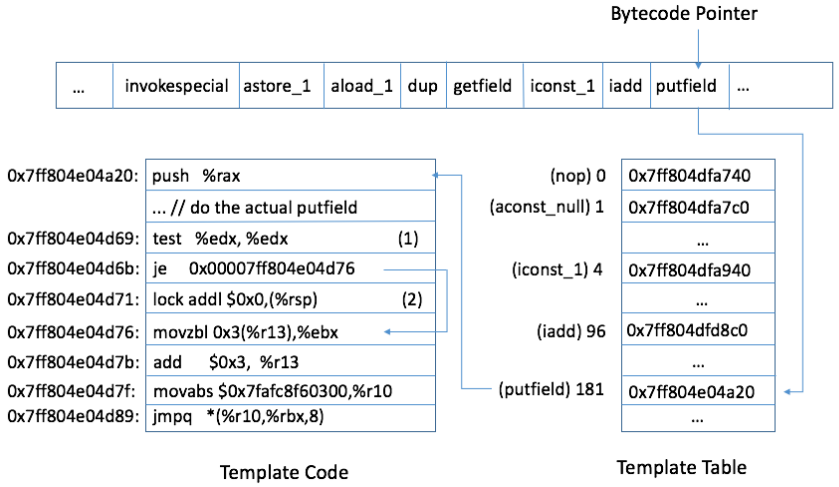

Fig. 1. Interpretation example of bytecode `putfield`

(1) and the following jump instruction `je`. We also added the `lock` instruction to the templates for the various bytecode instructions that perform array writes (e.g., `aastore` for storing objects into arrays, `bastore` for storing booleans into arrays, etc.).

Inserting memory-barrier instructions ensures that the hardware respects SC, but it does not prevent the interpreter itself from performing optimizations that can violate SC. The interpreter performs optimizations through a form of bytecode rewriting, including rewriting bytecodes to new ones that are not part of the standard Java bytecode language. For example, on encountering a `putfield` bytecode and resolving the field to which it refers, the interpreter rewrites the bytecode into a "fast" version (`fast_aputfield` if the field is an Object, `fast_bputfield` if the field is a boolean, etc.) The next time the interpreter executes the enclosing method, it will execute the faster version of the bytecode, avoiding the need to resolve the field again.

We manually inspected all of the interpreter's bytecode-rewriting optimizations and found that they never reorder the memory accesses of the original bytecode program. In other words, the interpreter does not perform optimizations that violate SC. However, to ensure SC semantics we had to modify the templates for all of the `fast_*putfield` bytecodes in order to unconditionally execute the `lock` instruction, as shown earlier for `putfield`.

Finally, the interpreter treats a small number of common and/or special methods, for example math routines from `java.lang.Math`, as *intrinsic*: the interpreter has custom assembly code for them. However, we examined the x86 implementations of these intrinsics and found that none of them contain writes to shared memory, so they already preserve SC.

5.3 Volatile-by-Default Compiler

When the JVM identifies a "hot spot" in the code, it compiles that portion to native code. As mentioned earlier, we have modified HotSpot's high-performance server compiler, which consists of several phases. First a hot spot's bytecode instructions are translated into a high-level graph-based intermediate representation (IR) called Ideal. The compiler performs several local optimizations on Ideal-graph nodes as it creates them. It then performs more aggressive optimizations on the

graph in a subsequent optimization phase. Next the optimized Ideal graph is translated to a lower-level platform-specific IR, and finally machine code is generated in the code generation phase. Optimizations are performed during each of these last two phases as well.

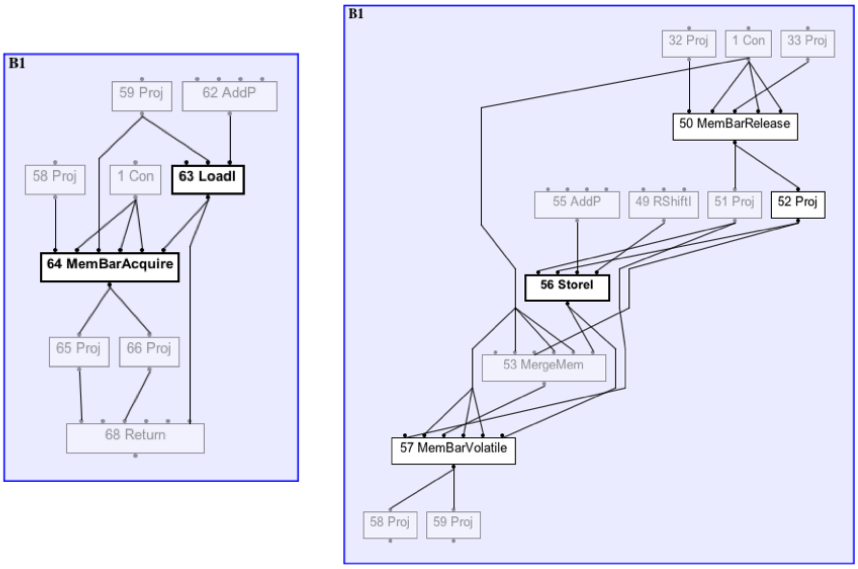

Fig. 2. Ideal graph sections for `volatile` reads (left) and writes (right).

At the Ideal graph level, the semantics of `volatile` is implemented by three kinds of memory-barrier nodes, each of which represents a specific combination of the four basic memory barriers: *LoadLoad*, *LoadStore*, *StoreLoad*, and *StoreStore*. Figure 2 shows snippets of the Ideal graph for `volatile` loads and stores. Each `volatile` load is followed by a `MemBarAcquire` node, which enforces "acquire" semantics: subsequent instructions (both load and store) cannot be reordered before the barrier node. Each `volatile` store is preceded by a `MemBarRelease` node, which enforces "release" semantics: prior instructions cannot be reordered after the barrier node. Each `volatile` store is also followed by a `MemBarVolatile` node, which prevents subsequent `volatile` memory accesses from being reordered before the barrier node.[8]

The memory-barrier nodes in the Ideal graph are translated to their counterparts in the lower-level IR. When generating machine code, they are finally translated into the appropriate assembly instructions. On x86 both the `MemBarAcquire` and `MemBarRelease` nodes become no-ops, since TSO already enforces those instruction orders. However, it is critical to keep these memory-barrier nodes in the code until the point of code generation, in order to prevent the compiler from performing optimizations that violate their semantics.

Given this structure, we chose to implement the `volatile`-by-default semantics by modifying the phase that creates the Ideal graph. Specifically, we modified that phase to emit the appropriate memory-barrier nodes around *all* loads and stores, rather than only `volatile` ones. As in the interpreter, this was done both for accesses to instance variables and to array elements.

An additional complication is that the server compiler treats many methods as intrinsic, providing a custom Ideal graph for each one. We carefully examined the implementation and documentation

[8] On the POWER processor [Mador-Haim et al. 2012], which is not multi-copy atomic, a `MemBarVolatile` also precedes each `volatile` load, but this is not necessary for x86.

of these intrinsics to ensure `volatile`-by-default semantics. First, some intrinsics, for example math routines from `java.lang.Math`, only access local variables and hence need not be modified. Second, we added appropriate memory-barrier nodes in the implementations of many intrinsics that perform memory loads and/or stores. For example, `getObject` from `sun.misc.Unsafe` loads an instance variable or array element directly by offset. We modified its Ideal-graph implementation to include a subsequent `MemBarAcquire` node, as is already done for the `getObjectVolatile` intrinsic from the same class. Finally, for some intrinsics, specifically certain string operations, we simply set the flag `-XX:-OptimizeStringConcat`, which causes the methods to be compiled normally instead of using the intrinsic implementations.

Modifying the compiler at this early stage ensures that we need not worry about the potential for any downstream compiler optimizations to violate SC, since those optimizations already respect the semantics of memory-barrier nodes. This holds true even for the local optimizations that are performed during Ideal-graph construction. For example, the local optimizations on a `Store` node, such as redundant store elimination, already take into account the presence of any preceding memory-barrier nodes, which is necessary to avoid violating the semantics of `volatile` stores.

5.4 Optimizations

Another important benefit of implementing the `volatile`-by-default semantics in the Ideal graph is that it allows us to take advantage of the optimizations that the server compiler already performs on memory-barrier nodes at different phases in the compilation process. For example, the compiler performs an optimization to remove redundant memory-barrier instructions. In this way, the optimizations that the server compiler already performs to optimize `volatile` accesses are automatically used to lower the cost of SC semantics.

We also added an optimization to the compiler that removes memory barriers for accesses to objects that do not escape the current thread. The HotSpot JVM already performs an escape analysis, which we simply reuse. In fact, earlier versions of the HotSpot JVM performed this optimization for a subset of non-escaping objects called *scalar-replaceable objects*, but it seems to have been accidentally removed in version 8u: the code for the optimization is still there but it was modified such that it never actually removes any memory barriers. We updated this code to properly remove `MemBarAcquire` and `MemBarVolatile` nodes for all non-escaping objects.[9]

Finally, the HotSpot JVM inserts a `MemBarRelease` node at the end of a constructor if the object being constructed has at least one `final` field, in order to ensure that clients only see the initialized values of such fields after construction. In VBD-HotSpot, this `MemBarRelease` node is unnecessary, because each individual field write in the constructor is already surrounded by appropriate memory-barrier nodes. Therefore, VBD-HotSpot does not insert memory barriers after constructors.

5.5 Correctness

Our main implementation technique, in both the VBD-HotSpot interpreter and compiler, is to simply reuse the existing mechanisms for handling accesses to `volatile` variables. Therefore, the correctness of VBD-HotSpot largely hinges on the correctness of those existing mechanisms, which have been in wide use as well as refined and debugged over more than a decade. We also validated VBD-HotSpot's correctness in several ways. First, we added a *VBDVerify* phase in the server compiler after the creation of the Ideal graph, which traverses the Ideal graph to check that all loads and stores are surrounded by appropriate memory-barrier nodes. Second, we created a suite of several litmus tests that sometimes exhibit non-SC behavior under the unmodified HotSpot

[9]Removing `MemBarRelease` nodes is trickier to implement, so we have not done it though it would be safe to do.

JVM, such as a version of Dekker's mutual exclusion algorithm. We have run these litmus tests many times on the VBD-HotSpot compiler and they have never exhibited a non-SC behavior, which helps lend confidence in our implementation.

5.6 Volatile-by-Default for Java and Scala

Finally, we note that VBD-HotSpot ensures `volatile`-by-default semantics for Java bytecode, but that does not immediately provide a guarantee in terms of the original Java source program. However, we have manually examined the widely used `javac` compiler that is part of the OpenJDK, which compiles Java source to bytecode, and ensured that it performs no optimizations that have the effect of reordering memory accesses. Hence compiling a Java program with `javac` and executing the resulting bytecode with VBD-HotSpot provides `volatile`-by-default semantics for the original program. We also examined the `scalac` compiler that compiles Scala source to Java bytecode[10] and found no optimizations that reorder memory accesses, so the same guarantees hold for Scala programs running on VBD-HotSpot.

6 EXPERIMENTAL RESULTS

In this section we describe our experiments that provide insight into the performance cost of SC for JVM-based server applications, which are a dominant use case today. We compared the performance of VBD-HotSpot to that of the original HotSpot JVM on several benchmark suites. The experiments are run on a modern x86 server machine, specifically a 12-core machine with 2 Intel Xeon E5-2620 v3 CPUs (2.40 GHz) with hyperthreading, which provides 24 processing units in total.

6.1 DaCapo Benchmarks

The DaCapo benchmarks suite is a set of open-source Java applications that is widely used to evaluate Java performance and represents a range of application domains [Blackburn et al. 2006]. We used the DaCapo 9.12 distribution. We excluded five of the benchmarks: *batik* and *eclipse* are not compatible with Java 8; *tradebeans* and *tradesoap* fail periodically, apparently due to an incompatibility with the `-XX:-TieredCompilation` flag[11], which VBD-HotSpot employs (see below); and *lusearch* has a known concurrency error that causes it to crash periodically.[12]

We ran the DaCapo benchmarks on our server machine and used the default workload and thread number for each benchmark. We used an existing methodology for Java performance evaluation [Georges et al. 2007]. For each JVM invocation, we ran each benchmark for 20 iterations, with the first 15 being the warm-up iterations, and we calculated the average running time of the last five iterations. We ran a total of 10 JVM invocations for each test and calculated the average of the 10 averages.

Figures 3 and 4 respectively show the absolute and relative execution times of VBD-HotSpot versus the baseline HotSpot JVM. By default the HotSpot JVM uses *tiered compilation*, which employs both the client and server compilers. Since we only modified the server compiler, VBD-HotSpot employs the `-XX:-TieredCompilation` flag to turn off tiered compilation and employ only the server compiler. Therefore we also present the results for running the original HotSpot JVM with this flag.

The geometric mean of all relative execution times represents a slowdown of 28% versus the original JVM, and the maximum slowdown across all benchmarks is 81%. The results indicate that

[10] http://www.scala-lang.org/download

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

[12] Interestingly, we observed the crash when executed on the original JVM but never on VBD-HotSpot, though we cannot be sure that the bug will never manifest under SC.

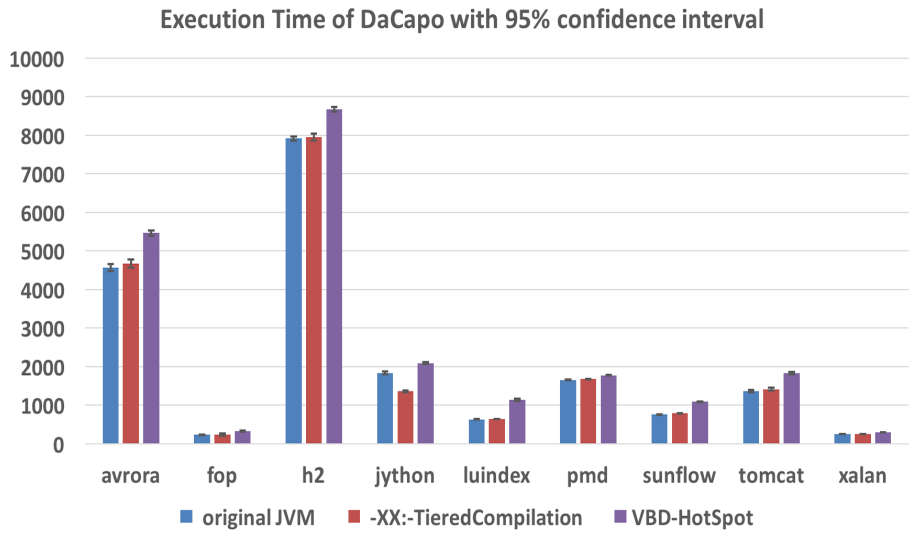

Fig. 3. Execution time in milliseconds of VBD-HotSpot on the DaCapo benchmarks. "original JVM" means running the baseline HotSpot JVM without additional flags; "-XX:-TieredCompilation" means running the baseline HotSpot JVM with -XX:-TieredCompilation; "VBD-HotSpot" shows results of running VBD-HotSpot.

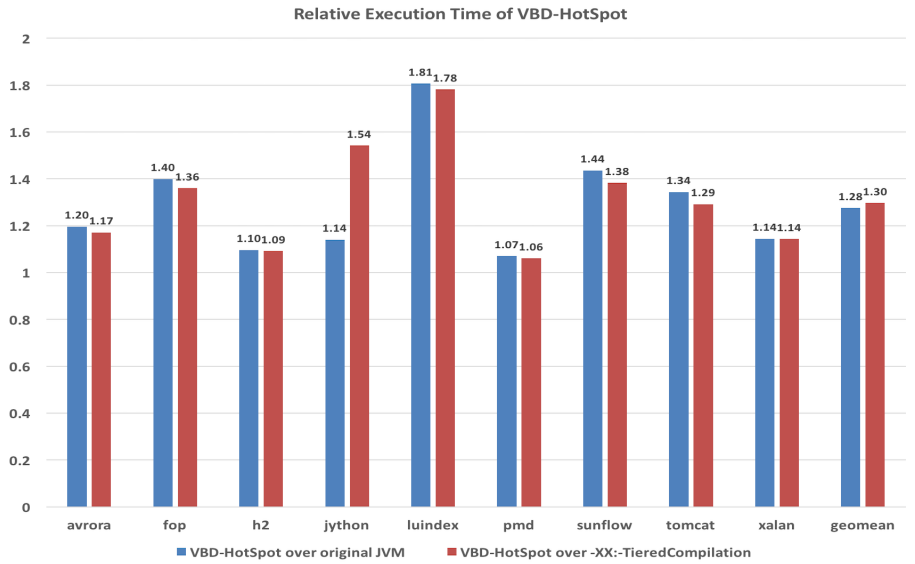

Fig. 4. Relative Execution time of VBD-HotSpot on the DaCapo benchmarks

SC incurs a significant cost on today's JVM and hardware technology, though perhaps less than is commonly assumed. The `-XX:-TieredCompilation` baseline is slightly slower than the default configuration for all but one benchmark (*jython*), which has a significant speedup. Because of *jython*'s speedup, the geometric mean of the overhead of VBD-HotSpot increases by 2%. However,

the maximum overhead for any benchmark decreases by 3%. In the rest of our experiments we present results relative to the default configuration of HotSpot, with tiered compilation enabled.

Interestingly, the three benchmarks that are mostly single-threaded incur some of the highest overheads. Specifically, *fop* is single-threaded, most of the tests for the *jython* benchmark are single-threaded, and *luindex* is single-threaded except for a limited use of helper threads that exhibit limited concurrency; all other benchmarks are multithreaded.[13] Ignoring the three benchmarks that are largely single-threaded, the geometric mean of VBD-HotSpot's relative execution time versus the original JVM is only 1.21 (i.e., a 21% slowdown) with a maximum overhead of 44%.

We conjecture that this difference in the cost of VBD-HotSpot for single-threaded and multi-threaded programs is due to the fact that multithreaded programs already must use synchronization in order to ensure desired program invariants and prevent data races. Hence the overhead of such synchronization might mask the cost of additional fences and also allow some of VBD-HotSpot's inserted fences to be safely removed by HotSpot's optimizations. On the other hand, for single-threaded programs every fence we add incurs additional overhead.

Of course, if the programmer is aware that her program is single threaded (or has limited concurrency such that it is obviously data-race-free), then she can safely run on the unmodified JVM and still obtain `volatile`-by-default semantics. Programmers can choose to do that in VBD-HotSpot simply by not setting the `-XX:+VBD` flag.

6.2 Spark Benchmarks

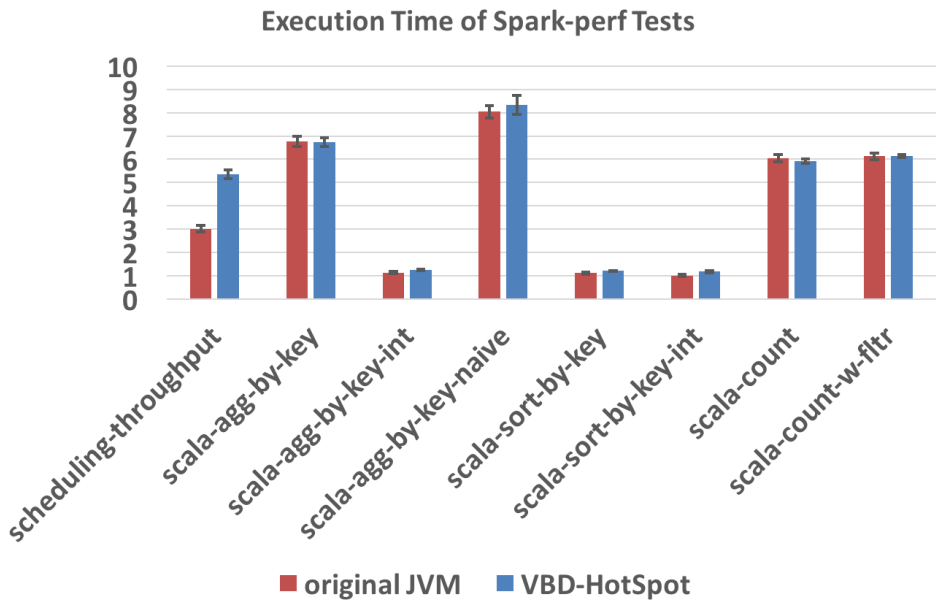

Fig. 5. Median execution time in seconds for `spark-tests`

Big-data analytics and machine learning are two common and increasingly popular server-side application domains. To understand the performance cost of the `volatile`-by-default semantics for these domains, we evaluated VBD-HotSpot on two benchmark suites for Apache Spark [Zaharia

[13] http://dacapobench.org/threads.html

Fig. 6. Average of 10 median execution times in seconds with 95% confidence interval for `mllib-tests` (part 1 of 4)

et al. 2016], a widely used framework for data processing. Specifically, we employ two sets of Spark benchmarks provided by Databricks as part of the `spark-perf` repository[14]: `spark-tests` includes several big-data analytics applications, and `mllib-tests` employs Spark's MLlib library [Meng et al. 2015] to perform a variety of machine-learning tasks. These experiments also illustrate how VBD-HotSpot can extend the `volatile`-by-default semantics to languages other than Java that compile to the Java bytecode language, since Spark is implemented in Scala.

We ran Spark in standalone mode on a single machine: the driver and executors all run locally as separate processes that communicate through specific ports. Since running Spark locally reduces the latency of such communication versus running Spark on a cluster, this experimental setup allows us to understand the worst-case cost of the `volatile`-by-default semantics. In our experiments, the executor memory is 4GB and the driver memory is 1GB. The `spark-perf` framework runs each benchmark multiple times and calculates the median execution time. Similar to the DaCapo tests, we ran `spark-perf` framework for 10 invocations and calculated the average of the median execution time.

Figure 5 shows the median execution time for the eight `spark-tests` benchmarks when run on the original HotSpot JVM as well as VBD-HotSpot. For four out of the eight benchmarks, the overhead of VBD-HotSpot is small or negligible, ranging from a slowdown of 3.5% to a speedup of 2.3%. The three shortest-running benchmarks (*scala-agg-by-key-int*, *scala-sort-by-key*, and *scala-sort-by-key-int*) have overheads ranging from 8% to 15%. As mentioned earlier, profiling reveals that the `spark-tests` benchmarks spend significant time doing I/O and waiting at barriers, which explains the relatively low overheads. The *scheduling-throughput* benchmark is an outlier, with an

[14]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.

Fig. 7. Average of 10 median execution times in seconds with 95% confidence interval for `mllib-tests` (part 2 of 4)

overhead for VBD-HotSpot of 77.5%. This benchmark performs a large amount of data serialization and deserialization, so VBD-HotSpot must insert fences for each of these data writes.

Figures 6 through 9 show the results for the `mllib` benchmarks.[15] We excluded four benchmarks that failed on the original JVM (*als*, *kmeans*, *gmm*, and *pic*). The geometric mean of VBD-HotSpot's relative execution time is 1.19, or a 19% slowdown. Figure 10 summarizes the results in a histogram. More than 37% of the benchmarks incur an average overhead of 10% or less, and more than 65% incur an average overhead of 20% or less. Only 11 out of 102 benchmarks have an overhead greater than 50%.

6.3 Scalability Experiments

We performed two experiments to understand how the cost of the `volatile`-by-default semantics changes with the number of threads/cores available. Our server machine has six physical cores per socket and two sockets, for a total of 12 physical cores. Further, the server has hyperthreading, which provides two logical cores per physical one, for a total of 24 logical cores. We were interested to understand how the cost of VBD-HotSpot would change with increased concurrency in general, as well as the cost difference on cores within one socket versus cores across multiple sockets.

For these experiments we used the `-t` option in DaCapo to set the number of driver threads for each test and Linux's `taskset` command to pin execution to certain cores. The `-t` option in DaCapo does not apply to the three largely single-threaded benchmarks that were mentioned in Section 6.1. It also does not apply to *avrora* and *pmd* — though these benchmarks use multithreading internally,

[15]Note that several benchmarks use the same application code but with different arguments. Also note that the y-axis of Figure 6 has a different scale than that of the other figures, due to the longer execution times of its benchmarks.

Fig. 8. Average of 10 median execution times in seconds with 95% confidence interval for `mllib-tests` (part 3 of 4)

they always use a single driver thread. Therefore our experiments only employed the remaining four DaCapo benchmarks.

First we tested the overhead of the four benchmarks with 1, 3, 6, 9, 12, and 24 driver threads. For the experiment with N driver threads we pin the execution to run on cores 0 through $N-1$, where cores 0-5 are different physical cores on one socket, cores 6-11 are different physical cores on the other socket, and cores 12-23 are the logical cores enabled by hyperthreading.

The results of our experiment are shown in Figure 11. The y-axis shows the relative execution time of running on VBD-HotSpot versus the baseline HotSpot JVM on each benchmark, and the x-axis provides this result for differing numbers of driver threads. Figure 12 provides the results in a different way, showing the absolute execution times in milliseconds with different numbers of driver threads. As the number of driver threads/cores increases from 1 to 12, there is a trend of improved performance for VBD-HotSpot relative to the original HotSpot JVM. The relative execution time then is flat or decreases modestly at 24 driver threads. These results belie a common assumption that SC performance suffers with increased concurrency. They also accord with an experiment by Boehm [2012] showing that a lock-based version of a particular parallel algorithm scales better than a version with no synchronization.

Second, we performed an experiment to specifically understand the performance difference for VBD-HotSpot when running on cores within the same socket versus on cores across sockets. We ran the DaCapo benchmarks with 6 driver threads in two different configurations: one using cores 0-5, which are all on the same socket, and one using cores 0-2 and 6-8, so that we have two sockets and three cores per socket.

Figure 13 shows the relative execution times for each configuration on VBD-HotSpot versus that same configuration executed on the original HotSpot JVM. The overhead of VBD-HotSpot in

Fig. 9. Average of 10 median execution times in seconds with 95% confidence interval for `mllib-tests` (part 4 of 4)

the multiple-sockets configuration is uniformly lower than that in the single-socket configuration, sometimes significantly so. This is an interesting result because in our experience SC is assumed to be more expensive cross-socket, due to the increased overhead of fences. However, it appears that the cross-socket configuration slows the rest of the execution down to a greater degree, thereby offsetting the additional cost of fences.

6.4 Relaxed Execution

We also performed experiments to gauge the potential for judicious usage of `relaxed` annotations to improve the performance of VBD-HotSpot. We profiled four of the five Da Capo benchmarks that incur the most overhead for VBD-HotSpot[16], as well as the *scheduling-throughput* benchmark from the `spark-tests` suite, to determine the methods in which each benchmark spends the most execution time. Figure 14 shows how the overheads of these benchmarks are reduced when the top k methods in terms of execution time are annotated as `relaxed`, for k ranging from 0 to 20. Declaring a method to be `relaxed` causes the method to be compiled exactly as in the original HotSpot JVM, so memory-barrier nodes are only inserted for accesses to variables that are explicitly declared `volatile`. Note that the interpreter still executes these methods with `volatile`-by-default semantics, and any methods called by these methods are both interpreted and compiled with `volatile`-by-default semantics.

The figure shows that annotating the top 20 or fewer methods as `relaxed` provides a large reduction in the overhead of VBD-HotSpot. Two of the benchmarks have particularly dramatic reductions in overhead: *luindex* reduces from 1.82 to 1.17 and *scheduling-throughput* reduces from 1.67 to 1.16. Many of the top methods are in the Java standard library and so could be declared

[16]We were not able to perform this experiment for *tomcat* as our profiler crashes when running this benchmark.

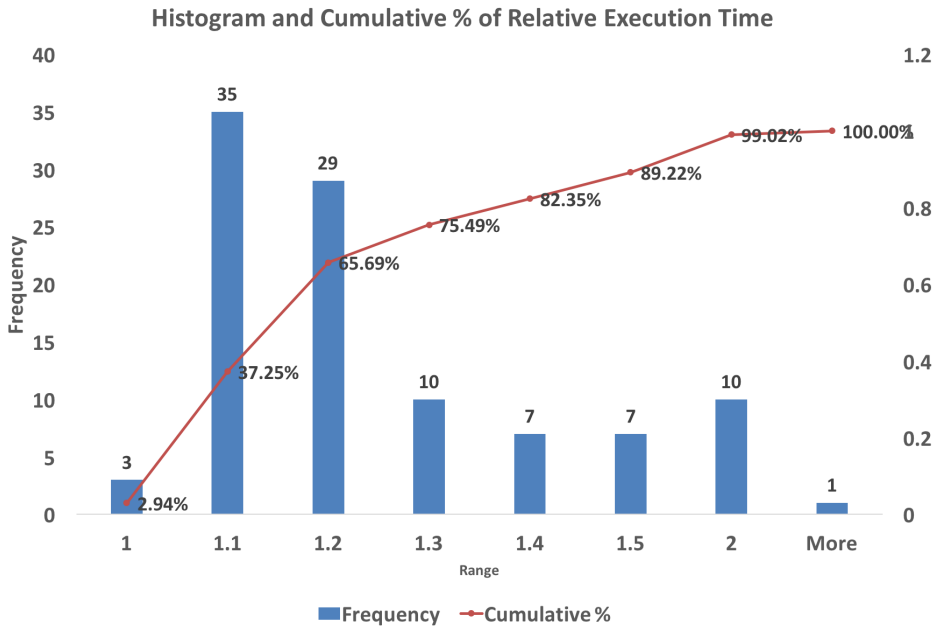

Fig. 10. Histogram and cumulative % of relative execution time for `mllib-tests`

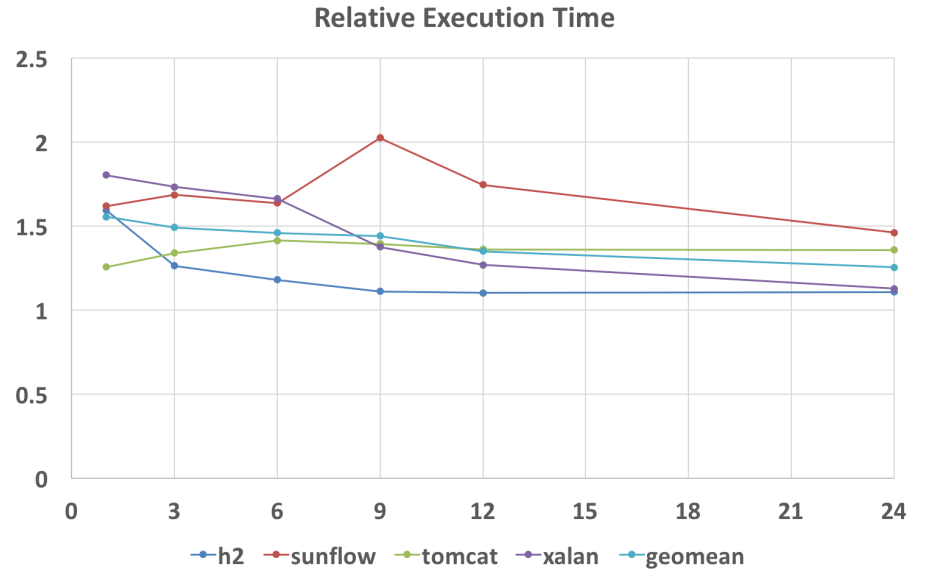

Fig. 11. The relative cost of VBD-HotSpot with different numbers of threads/cores

`relaxed` once and then used by many applications. For example, 16 of the top 20 methods for *scheduling-throughput* are in the `java.io` library and perform reading and writing of object streams.

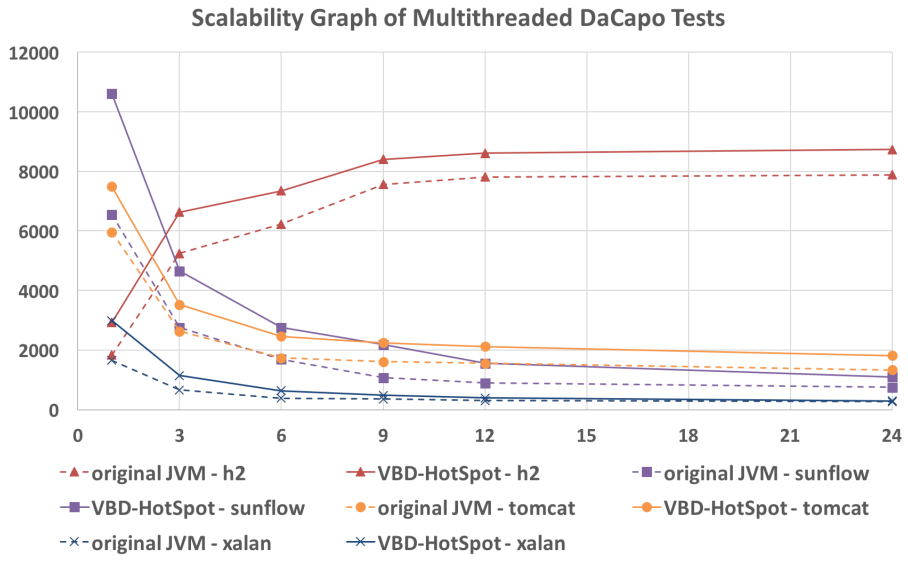

Fig. 12. Scalability graph with different numbers of threads/cores

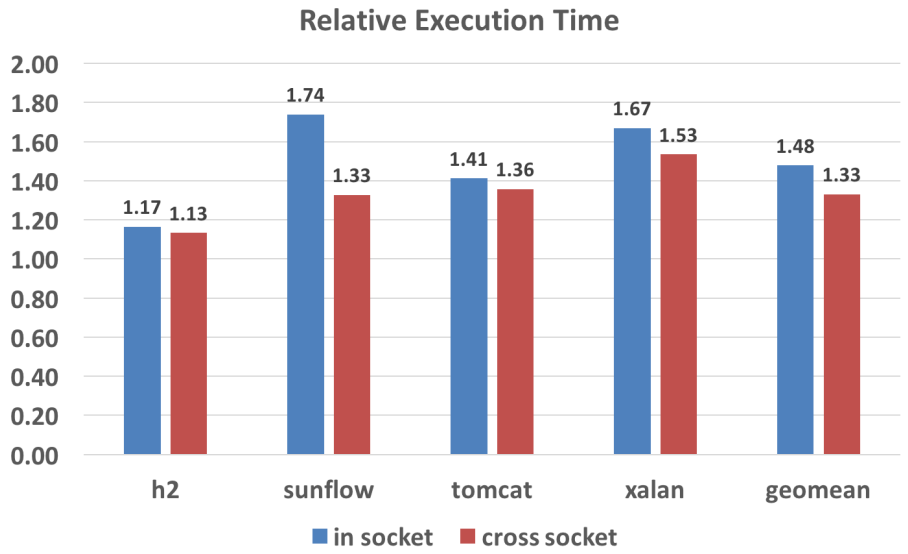

Fig. 13. The cost of VBD-HotSpot within socket and cross-socket

With 20 `relaxed` annotations each on the five benchmarks in Figure 14, the geometric mean of VBD-HotSpot's overhead reduces to 18% for the entire Da Capo suite (with a max overhead of 34% for *tomcat*) and 6.6% for the entire `spark-tests` suite (with a max overhead of 16%). These results show that for big-data applications, which are important use cases for the JVM on servers today, the `volatile`-by-default semantics can be a practical choice on modern server hardware, with a judicious choice of `relaxed` annotations.

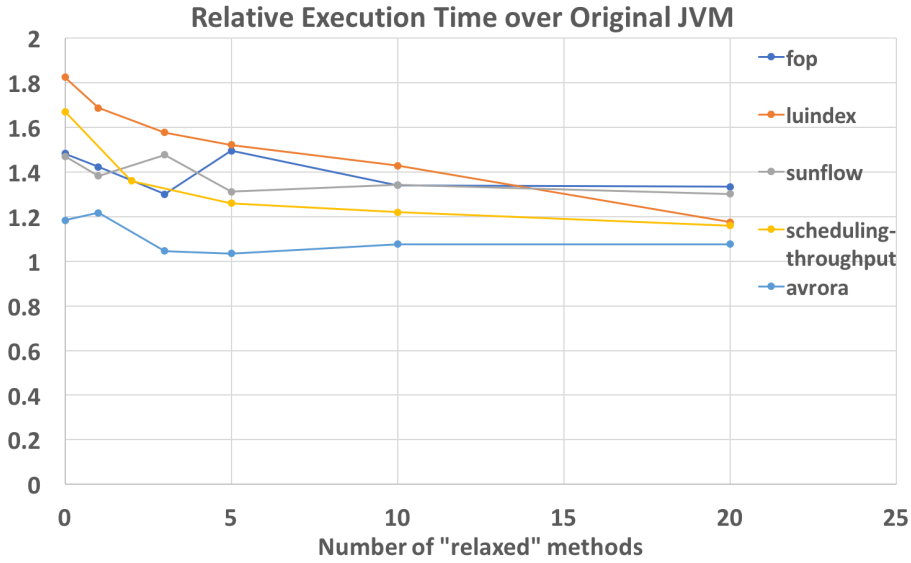

Fig. 14. The cost of VBD-HotSpot with `relaxed` methods

6.5 Consumer PCs

Finally, we also ran our benchmarks on several consumer PC machines, in addition to our server machine. PC1 is a 6-core machine with an Intel Core i7-3930K CPU (3.20GHz), which was released in the fourth quarter of 2011. PC2 is a 4-core machine with an Intel Core i7-4790 CPU (3.20GHz), which was released in the second quarter of 2014. PC3 is a 4-core machine with an Intel Core i7-6700 CPU (3.40GHz), which was released in the third quarter of 2015. Hyperthreading is enabled on all three machines. Therefore we respectively have 12, 8, and 8 processing units for PC1, PC2, and PC3.

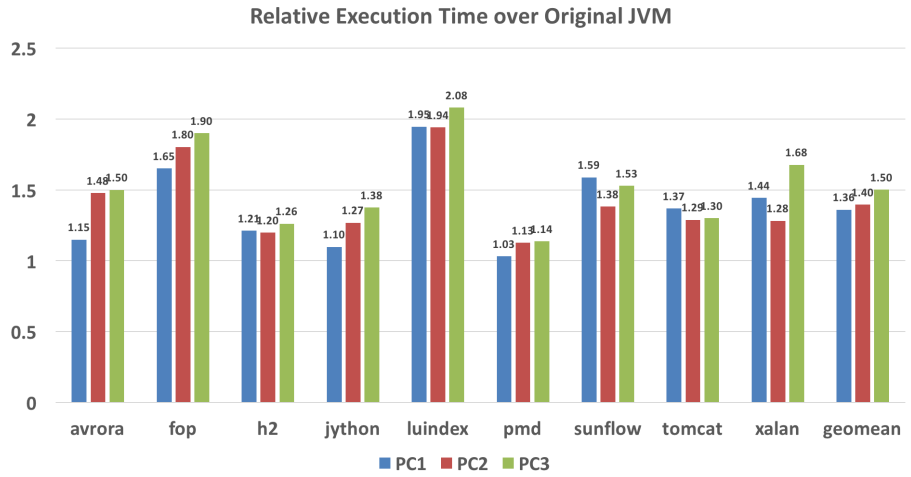

Fig. 15. Relative execution time of the DaCapo benchmarks

We ran the DaCapo benchmarks on these machines using the same setup as in Section 6.1. Figure 15 shows the relative execution time for VBD-HotSpot of the benchmarks on the three machines, normalized to the execution time when run on the baseline HotSpot JVM. The geometric mean of the overhead due to the `volatile`-by-default semantics is respectively 36%, 40%, and 50% on machines PC1, PC2, and PC3, which is somewhat higher than the overhead of VBD-HotSpot on our server machine (Section 6.1).

Though not uniformly so, the results indicate an upward trend on the cost of the `volatile`-by-default semantics over time, since PC1 is the oldest and PC3 the newest machine. It's hard to identify the exact cause of this trend, or whether it is an actual trend, since the machines differ from one another in several ways (number of processors, execution speed, microarchitecture, etc.). However, the absolute performance of the benchmarks improves over time. Therefore, one possible explanation is that the performance of fences is improving relatively less than the performance of other instructions.

7 RELATED WORK

Language-Level Sequential Consistency. Sura et al. [2005] implemented a high-performance SC compiler for Java, and Kamil et al. [2005] did the same for a parallel variant of Java called Titanium. However, both compilers use heavyweight analyses such as interprocedural escape analysis and delay-set analysis [Shasha and Snir 1988] in the context of an offline, whole-program compiler. Alglave et al. [2014] implemented SC for C programs similarly. In contrast, VBD-HotSpot is implemented in the production HotSpot JVM and maintains the modern JIT compilation style. Recent work [Vollmer et al. 2017] has shown that SC semantics for the Haskell programming language has negligible overhead on x86. This work relies heavily on Haskell's largely functional style and its type system's clean separation of functional and imperative code.

Other work has achieved language-level SC guarantees for Java through a combination of compiler modifications and specialized hardware [Ahn et al. 2009; Ceze et al. 2007]. Such a combination has also been used to provide SC semantics for C [Marino et al. 2011; Singh et al. 2012]. The results of these works show that SC can be comparable in efficiency to weak memory models with appropriate hardware support. Finally, several works demonstrate testing techniques to identify errors in Java and C code that can cause non-SC behavior (e.g., [Flanagan and Freund 2010; Islam and Muzahid 2016]).

Language-Level Region Serializability. Another line of 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. Sengupta et al. [2015a] enforce a form of region serializability for Java through a two-phase locking protocol implemented in the Jikes research virtual machine [Alpern et al. 2005]. The approach achieves good performance but has high implementation complexity. For example, code regions must be transformed such that they can be safely restarted in the event of a deadlock, and a whole-program static data-race detector is used to optimize the technique. Follow-on work by the authors incorporates coarse-grained locking along with runtime profiling to choose the granularity of locking for different memory accesses [Sengupta et al. 2015b]. Work on region serializability for C has achieved good performance either through special-purpose hardware [Lucia et al. 2010; Marino et al. 2010; Singh et al. 2011] or by requiring $2N$ cores to execute an application with N threads [Ouyang et al. 2013].

Memory Model Safety. The notion of "safety" in the JMM disallows out-of-thin-air values [Manson et al. 2005]. Recent work [Boehm and Demsky 2014] proposes a lightweight fencing mechanism to

guarantee this notion of safety (for Java and C/C++) by preserving load-to-store orders. However, that work does not empirically evaluate the cost of the proposal. Our work adopts and empirically evaluates a stronger notion of safety for Java, which additionally preserves the program order of instructions [Marino et al. 2015] and the atomicity of primitive types.

Weak Memory Model Performance. Finally, other work measures Java performance for weaker memory models than SC. Demange et al. [2013] define a TSO-like memory model for Java. They present a performance evaluation that uses the Fiji real-time virtual machine [Pizlo et al. 2010] to translate Java code to C, which is then compiled with a modified version of the LLVM C compiler [Marino et al. 2011] and executed on x86 hardware. Ritson and Owens [2016] modified the HotSpot compiler's code-generation phase for both ARM and POWER to experiment with different platform-specific instruction sequences to implement the JMM.

8 CONCLUSION

This paper defines the `volatile`-by-default semantics as a natural way to make the memory consistency model of Java and other JVM-based languages safe-by-default and performant-by-choice. The `volatile`-by-default semantics protects most programmers from the vagaries of relaxed-memory-model behavior by providing sequential consistency by default, while still allowing expert programmers to avoid fence overheads on performance-critical libraries. We presented VBD-HotSpot, a modification of Oracle's HotSpot JVM that enforces the `volatile`-by-default semantics on Intel x86 hardware. To our knowledge this is the first implementation of SC for Java in the context of a production JVM and hence the first realistic performance evaluation of the cost of SC for Java. Our experiments indicate that while VBD-HotSpot incurs a significant performance cost relative to the baseline HotSpot JVM for some programs, it can be a practical choice today for certain application domains, notably big-data analytics and machine learning.

In our implementation of VBD-HotSpot we largely inherited the existing HotSpot optimizations for `volatile` accesses. In future work we are interested to explore ways to optimize VBD-HotSpot to make the `volatile`-by-default semantics more widely applicable in practice. We also plan to build and evaluate a `volatile`-by-default implementation of Java for ARM hardware by modifying the Android runtime's bytecode compiler.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their useful feedback. This work is supported in part by the National Science Foundation award CCF-1527923.

REFERENCES

Sarita V. Adve and Hans-J. Boehm. 2010. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM* 53, 8 (Aug. 2010), 90–101. https://doi.org/10.1145/1787234.1787255

S. V. Adve and M. D. Hill. 1990. Weak ordering—a new definition. In *Proc. of the 17th Annual International Symposium on Computer Architecture*. ACM, 2–14.

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*.

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.

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.

ARMv8 2017. ARM Cortex-A Series Programmer's Guide for ARMv8-A Version: 1.0, Section 13.2.1. (2017). https://developer.arm.com/docs/den0024/latest/13-memory-ordering/132-barriers/1321-one-way-barriers Accessed July 2017.

D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J. W. Maessen, J. D. Mitchell, K. Nilsen, B. Pugh, and E. S. Sirer. Accessed April 2017. The "Double-Checked Locking is Broken" Declaration. (Accessed April 2017). http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

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.

Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA.

Hans-J. Boehm. 2012. Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, 9–14.

H. J. Boehm and S. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proc. of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 68–78.

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.

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming (Lecture Notes in Computer Science)*, Rocco De Nicola (Ed.), Vol. 4421. Springer, 331–346.

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.

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.

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.

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.

K. Gharachorloo, A. Gupta, and J. Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the International Conference on Parallel Processing*. 355–364.

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

Java Virtual Machine Specification 2017. Accessed July 2017. (2017). https://docs.oracle.com/javase/specs/jvms/se8/html

JSR133 2017. JSR-133 Cookbook for Compiler Writers. Accessed July 2017. (2017). http://g.oswego.edu/dl/jmm/cookbook.html

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. 17:1–17:29.

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.

L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers* 100, 28 (1979), 690–691.

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*.

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 495–512.

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

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.

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.

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*.

Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, 695–710.

Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. *CoRR* abs/1505.06807 (2015). http://arxiv.org/abs/1505.06807

OpenJDK 2017. Accessed July 2017. (2017). http://openjdk.java.net

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.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009 (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 391–407.

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.

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.

Douglas C. Schmidt and Tim Harrison. 1997. Double-checked Locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Pattern Languages of Program Design 3*, Robert C. Martin, Dirk Riehle, and Frank Buschmann (Eds.). Addison-Wesley Longman Publishing Co., Inc., 363–375.

Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015a. 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.

Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2015b. 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.

Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. 27–51.

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.

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.

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.

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.

TSM03-J 2017. TSM03-J. Do not publish partially initialized objects. Accessed July 2017. (2017). https://www.securecoding.cert.org/confluence/display/java/TSM03-J.+Do+not+publish+partially+initialized+objects

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.

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.

Appendix

After the publication of this paper in OOPSLA 2017, we found a bug in our test configuration for the `spark-perf` benchmarks that caused incorrect flags to be used for some JVM invocations.[1] As a result the published performance numbers for these benchmarks under-count the performance cost of our technique. (The published numbers for other benchmarks are accurate.) We have fixed the bug and rerun all `spark-perf` tests, and the corrected results are in this appendix.

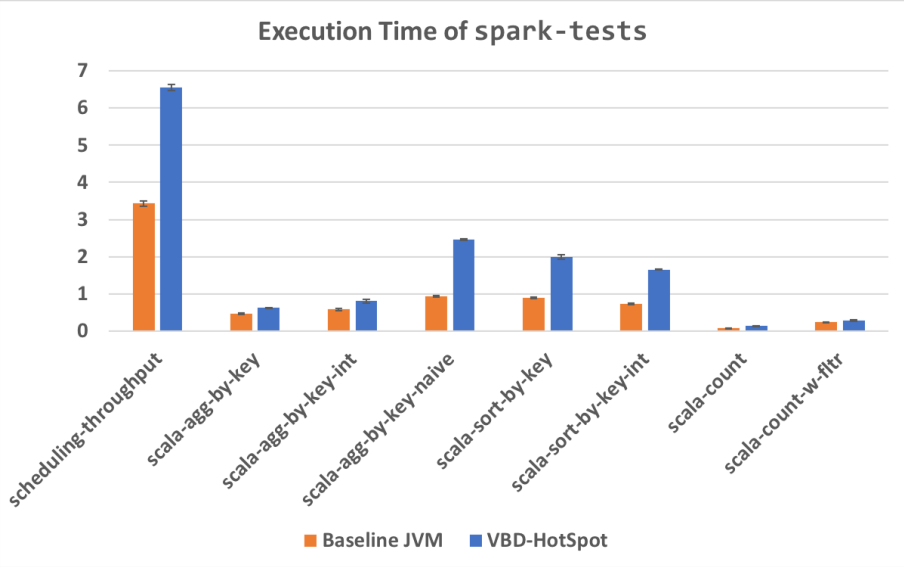

Figure A.1: Median execution time in seconds for `spark-tests`

As in the original paper, we ran Spark in standalone mode on a single machine. The executor memory is 4GB and the driver memory is 1GB, with a `scale_factor` of 0.05. The `spark-perf` framework runs each benchmark multiple times and calculates the median execution time. We ran the `spark-perf` framework for five invocations and calculated the average of the median execution time.

Figure A.1 shows the median execution times for the eight `spark-tests` benchmarks when run on the original HotSpot JVM as well as on VBD-HotSpot,

[1] Thanks to David Devecsery for identifying this issue when attempting to replicate our results.

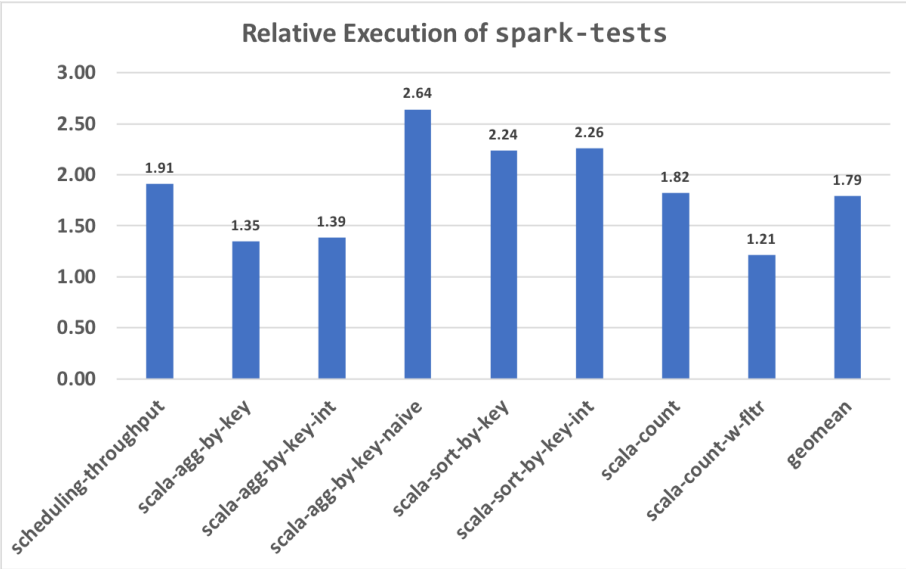

Figure A.2: Relative execution Time of VBD-HotSpot over the baseline JVM for `spark-tests`

with 95% confidence intervals. This figure replaces Figure 5 from the original paper. Figure A.2 shows the same results but as a relative execution time of VBD-HotSpot over the baseline HotSpot JVM. The geometric mean of the overhead of VBD-HotSpot is 79%, which is significantly higher than the average overhead of VBD-HotSpot on the Da Capo benchmarks (see Figure 4). We surmise that the large overhead for Spark benchmarks is due Spark's use of the *resilient distributed dataset* (RDD) abstraction, which is an in-memory, immutable data structure. Each Spark operation potentially incurs many memory operations in order to read its input RDDs and write its output RDDs.

Figures A.3 through A.6 show the results for the `mllib` benchmarks, replacing Figures 6-9 from the original paper. Figure A.7 shows these results in a histogram, replacing Figure 10 from the original paper. The geometric mean of VBD-HotSpot's relative execution time is 1.67, or a 67% slowdown. These results are consistent with those for the `spark-tests` benchmarks shown above.

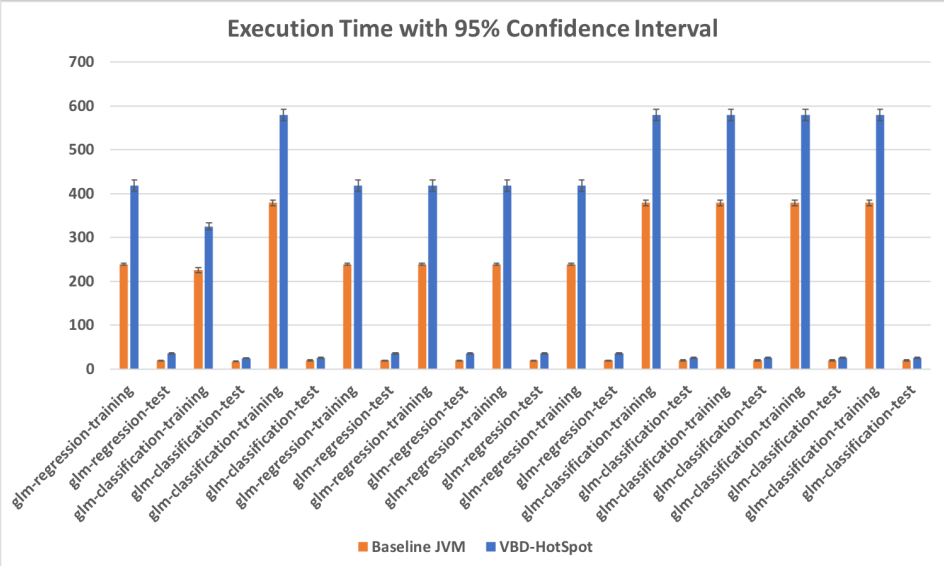

Figure A.3: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 1 of 4)

Figure A.4: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 2 of 4)

Figure A.5: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 3 of 4)

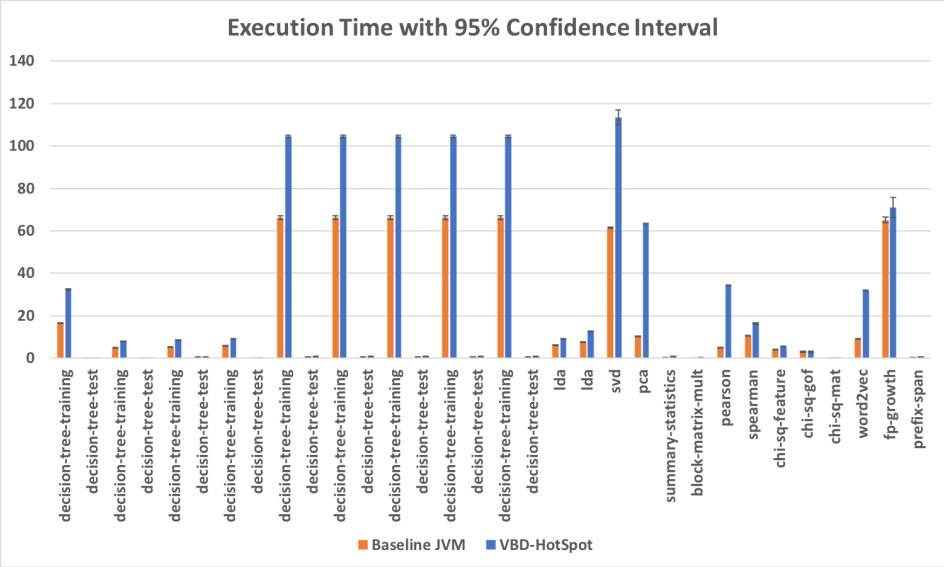

Figure A.6: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 4 of 4)

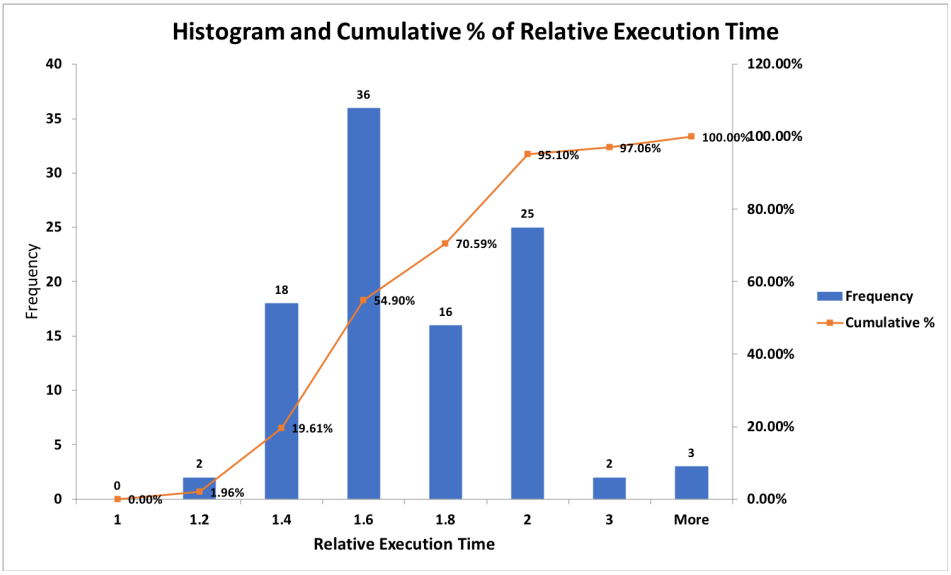

Figure A.7: Histogram and cumulative % of relative execution time for `mllib-tests`