Safe-by-default Concurrency for Modern Programming Languages

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

10

Modern "safe" programming languages follow a design principle that we call *safety by default* and *performance by choice*. By default, these languages enforce important programming abstractions, such as memory and type safety, but they also provide mechanisms that allow expert programmers to explicitly trade some safety guarantees for increased performance. However, these same languages have adopted the inverse design principle in their support for multithreading. By default, multithreaded programs violate important abstractions, such as program order and atomic access to individual memory locations to admit compiler and hardware optimizations that would otherwise need to be restricted. Not only does this approach conflict with the design philosophy of safe languages, but very little is known about the practical performance cost of providing a stronger default semantics.

In this article, we propose a safe-by-default and performance-by-choice multithreading semantics for safe languages, which we call `volatile`-*by-default*. Under this semantics, programs have *sequential consistency* (SC) by default, which is the natural "interleaving" semantics of threads. However, the `volatile`-by-default design also includes annotations that allow expert programmers to avoid the associated overheads in performance-critical code. We describe the design, implementation, optimization, and evaluation of the `volatile`-by-default semantics for two different safe languages: Java and Julia. First, we present VBD-HotSpot and VBDA-HotSpot, modifications of Oracle's HotSpot JVM that enforce the `volatile`-by-default semantics on Intel x86-64 hardware and ARM-v8 hardware. Second, we present SC-Julia, a modification to the just-in-time compiler within the standard Julia implementation that provides best-effort enforcement of the `volatile`-by-default semantics on x86-64 hardware for the purpose of performance evaluation. We also detail two different implementation techniques: a *baseline* approach that simply reuses existing mechanisms in the compilers for handling atomic accesses, and a *speculative* approach that avoids the overhead of enforcing the `volatile`-by-default semantics until there is the possibility of an SC violation. Our results show that the cost of enforcing SC is significant but arguably still acceptable for some use cases today. Further, we demonstrate that compiler optimizations as well as programmer annotations can reduce the overhead considerably.

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

Additional Key Words and Phrases: Memory consistency models, sequential consistency, just-in-time compilers

This work is supported in part by National Science Foundation under grants CCF-1527923 and CNS-1704336.
Authors' addresses: L. Liu and T. Millstein, UCLA Computer Science Department, 404 Westwood Plaza, Box 951596, Los Angeles, CA 90095-1596; emails: {lunliu93, todd}@cs.ucla.edu; M. Musuvathi, Microsoft Research, Redmond, Microsoft Building 99, 14820 NE 36th Street, Redmond, WA, 98052; email: madanm@microsoft.com.

0164-0925/2021/08-ART10 $15.00
https://doi.org/10.1145/3462206

ACM Reference format:
Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2021. Safe-by-default Concurrency for Modern Programming Languages. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 10 (August 2021), 50 pages.
https://doi.org/10.1145/3462206

1 INTRODUCTION

In common parlance, *safety* in programming languages almost always refers to memory safety and type safety. However, the notion of safety is more general and refers to the ability of programming languages to protect their abstractions [52]. For instance, a language is memory-safe if it maintains the illusion that each object is logically separate—an update to one cannot change the state of the other despite being allocated in the same address space. Similarly, a language is type safe if it enforces the abstraction of types by only allowing intended operations on objects, thereby encapsulating their implementation details. Because protecting language abstractions typically induces associated runtime costs, for example garbage collection, array-bounds checks, and runtime type checks, safe languages also provide mechanisms for expert programmers to selectively trade safety guarantees for performance, such as native code and `unsafe` code blocks.

We call this paradigm of language design *safety by default* and *performance by choice*. Today's mainstream "safe" languages, for example Java and Julia, implement this paradigm for type and memory safety, and for good reason. This paradigm allows all programs to enjoy important safety guarantees by default, while still providing flexibility to selectively give up these guarantees where necessary for performance. Any unsafe code can be carefully audited for correctness, and any safety violations that do occur can be localized back to those sections of code.

Unfortunately, these same languages have adopted the inverse paradigm of *performance by default* and *safety by choice* in their design for concurrency via multithreading. By default, multithreaded programs violate important abstractions that are relied upon from sequential programming. For example, the instructions within a thread can execute out of order, the effects of a function might not complete before it returns, objects can be accessed before being fully constructed, and accesses to certain primitive types, such as `long`s and `double`s, are not guaranteed to be atomic. Rather, these abstractions are only enforced by choice, namely if programmers appropriately annotate certain memory accesses, for example with the `volatile` keyword in Java. Only such *well annotated* programs are guaranteed to enjoy the benefits of the sequential abstractions above [42].

These problems arise from the *memory consistency models* adopted by mainstream programming languages. A memory consistency model (or simply *memory model*) defines the possible values that a shared-memory read may return. ***Sequential consistency* (SC)** [35] guarantees that all memory reads and writes occur in a global total order that is consistent with the per-thread program order. This standard "interleaving" semantics of threads is natural, as it enforces two crucial abstractions that underlie sequential reasoning of programs. First, even though instructions can execute out of order due to compiler and hardware optimizations, SC provides the illusion that instructions take effect in the order specified in the program text. Second, despite complex memory hierarchies orchestrated by sophisticated cache coherence protocols, shared memory behaves logically as a global map from addresses to values, with each primitive memory read or write taking effect atomically. However, due to concerns about the cost of enforcing SC, today's languages instead have opted for *relaxed* (or *weak*) memory models that allow these two abstractions to be violated by default.

1.1 The `volatile`-by-default Memory Model

In this article, we present a *safe by default* and *performance by choice* memory model called `volatile`-*by-default*. Our memory model is a conceptually simple change to the memory models

of today's languages. In our design, each memory access has sequentially consistent semantics by default; in Java this is akin to annotating every variable as `volatile`. In this way, sequential consistency is guaranteed by default for all programs. However, expert programmers can employ new annotations that we introduce, such as `relaxed`, to "turn off" the default guarantees and overhead in performance-critical code.[1]

We stress that the `volatile`-by-default memory model and those of mainstream languages like Java [42] are *equivalent*, both in terms of safety and in terms of performance. It is simply a *syntactic* difference regarding what accesses need to be annotated. As stated above, a Java programmer today can trivially obtain the safety guarantees of `volatile`-by-default by annotating every variable as `volatile`.[2] Similarly, under the `volatile`-by-default semantics a programmer can trivially obtain the performance guarantees of today's **Java memory model (JMM)** by annotating the entire program as `relaxed`.

But this syntactic change of default makes a huge difference! Under the JMM, programs can behave in unpredictable and unsafe ways by default, and the programmer has no easy way to know when sufficient annotations have been added to prevent them. Under `volatile`-by-default, programs obtain basic safety guarantees by default, and programmers can make informed choices about how and where to place annotations to obtain desired performance characteristics while minimizing the potential for error. Indeed, this is exactly analogous to how today's safe languages address memory and type safety.

What are the barriers to considering this change of default? First, maybe it is unnecessary, because programs in practice already enjoy the guarantees of sequential consistency. However, this does not seem to be the case. As we show (Section 2.1.2), programmers routinely fail to specify necessary (`volatile`) annotations, resulting in hard-to-find bugs. Given that this annotation task is akin to manually performing static data race detection, perhaps this result is not surprising. Further, it has turned out to be difficult for language designers to provide a reasonable semantics for programs that are not well annotated [10] while still admitting desired optimizations, and this continues to be an active area of research.

Second, it is possible that the annotation burden introduced by this syntactic change of default is unacceptable. For instance, if a large percentage of memory accesses need to be annotated as `relaxed` to get acceptable performance, then we would arguably not be better off by switching defaults in practice. However, our experimental results indicate that this is not the case. For example, with 20 or fewer `relaxed` method annotations the overhead of `volatile`-by-default for a standard Java benchmark suite on x86-64 is less than 20% (Section 3.3.2), and a straightforward usage of annotations for Julia code shows similar effectiveness (Section 4.3.3).

This conclusion might seem obvious: Since only small parts of the code are likely to be performance-critical, only those parts need to be tagged as `relaxed`. However, sequential consistency is often dismissed from consideration as a possible language-level memory model. For instance, Kaiser et al. [31] say that SC is "woefully unrealistic" due to "the significant performance costs ... on modern multi-core architectures," and Demange et al. [21] say that "SC would likely cripple performance of Java programs on all modern microprocessors." Our results indicate that sequential consistency *by default*, with `relaxed` annotations to selectively trade some guarantees for increased performance, can be a viable memory model for modern programming languages.

[1]The `volatile` keyword is specific to Java, but other languages have analogous keywords; throughout, we use the term `volatile`-by-default generically to refer to our memory model as applied to any programming language.
[2]Technically this is currently not possible in Java, as its syntax does not allow `volatile` annotations on array elements.

1.2 Contributions

This article describes the design, implementation, and evaluation of the `volatile`-by-default memory model for safe programming languages.[3]

Design. We discuss the design philosophy of the `volatile`-by-default memory model and how it resolves key challenges of today's memory models. To illustrate the general applicability of our design, we have instantiated it for two very different mainstream languages: Java and Julia. Java is a well-established statically typed language that is widely used for many purposes, including server-side and mobile applications. Julia is an up-and-coming dynamic language that is increasingly popular for array-heavy scientific computing.

Implementation. We have implemented the `volatile`-by-default semantics for Java as a modification to Oracle's HotSpot **Java virtual machine (JVM)**, which is the default implementation platform for Java. Our implementation supports both Intel x86-64 and ARM-v8 hardware architectures. We have also implemented the `volatile`-by-default semantics for Julia as a modification to Julia's default compiler, which is built on top of LLVM [36]. Our implementation supports the Intel x86-64 hardware architecture. Our implementation provides the `volatile`-by-default semantics for the LLVM IR and employs existing metadata capabilities in LLVM to transfer higher-level program information that is useful for optimizations. As such, our implementation can be reused to provide `volatile`-by-default semantics for other languages that target the LLVM IR.

While several prior works [32, 60, 62, 63] have proposed ways of implementing SC, they rely on either whole-program compilation or special hardware support to achieve good performance. Instead, we have developed two implementation strategies that are compatible with modern implementation techniques, based on virtual machines with **just-in-time (JIT)** compilation and dynamic class loading, and with stock hardware. The *baseline* implementation is a low-complexity approach that simply reuses the existing mechanisms for handling `volatile` and related accesses to treat all accesses as such. We have implemented this strategy for both Java and Julia. The *speculative* implementation is a more sophisticated approach that treats a newly created object as *safe*, meaning that its fields can be accessed without additional hardware fences. If the object ever becomes *unsafe*, meaning that it is possible for its accesses to violate SC, then any speculatively compiled code for the object is removed, and future JITed code for the object will contain the necessary fences to ensure the `volatile`-by-default semantics. We have implemented this strategy for Java on ARM-v8 hardware.

The source code of our baseline `volatile`-by-default JVM for x86-64, called VBD-HotSpot, can be found on GitHub at https://github.com/SC-HotSpot/VBD-HotSpot. The source code for both the baseline and the speculative implementations of the `volatile`-by-default JVM for ARM-v8, respectively, called VBDA-HotSpot and S-VBD, can be found on GitHub at https://github.com/Lun-Liu/schotspot-aarch64. Finally, the implementation of our Julia compiler, called SC-Julia, is also open source and can be found on GitHub at https://github.com/Lun-Liu/SC-by-Default-Julia.

Evaluation. We perform a comprehensive empirical evaluation of the `volatile`-by-default memory model for both Java and Julia. First, we describe results for the baseline implementation for both Java and Julia, on Intel x86-64 hardware and *with no* `relaxed` *annotations*. On the DaCapo benchmarks our baseline implementation has an average overhead of between 28% and 50%, depending on the specific machine tested. We additionally analyze various factors that affect the cost of `volatile`-by-default for both Java and Julia. Finally, we explore the potential for annotations to

[3]The Java-based work and results were originally described in two prior conference publications [38, 39]. The Julia-based work and results were reported in the first author's dissertation [37] but have not been otherwise published.

Table 1. Key Experiments and Findings in Article

Figure	Experiment	Main Findings
3, 4	Performance evaluation of VBD-HotSpot with DaCapo benchmarks	Geometric mean of overhead: 28% Maximum overhead 81%
5, 6	Performance evaluation of VBD-HotSpot with `spark-tests` benchmarks	Geometric mean of overhead: 79% Maximum overhead 164%
7	Performance evaluation of VBD-HotSpot with `mllib-tests` benchmarks	Geometric mean of overhead: 67% 54.9% of benchmarks have overhead <= 60% 95.1% of benchmarks have overhead <=100%
8, 9	Scalability experiments of VBD-HotSpot with DaCapo benchmarks	Relative overhead decreases with increased concurrency.
10	Cost of VBD-HotSpot with `relaxed` methods	20 `relaxed` method annotations each on 4 benchmarks reduces the geometric mean of overhead for DaCapo to 18% and the maximum overhead to 34%.
11	Performance evaluation of VBD-HotSpot on consumer PCs with DaCapo benchmarks	Geometric mean of overhead is between 36% and 50% for different machines.
12–14	Performance evaluation of VBDA-HotSpot with different fence instructions	For VBDA-HotSpot, using `dmbs` for `volatiles` is much faster than using `ldar`/`stlr`, while for the original HotSpot there is little performance difference.
15, 16	Performance evaluation of VBDA-HotSpot with DaCapo benchmarks	Geometric mean of overhead: 73% (machine A), 57% (machine B) Maximum overhead: 129% (machine A), 157% (machine B).
17	Scalability experiments of VBDA-HotSpot with DaCapo benchmarks	Relative overhead is flat or slightly decreases with increased concurrency
18	Performance evaluation of VBDA-HotSpot with `spark-tests` and `mllib-tests` benchmarks	Geometric mean of overhead: 103% (machine A), 85% (machine B)
19	Performance evaluation of SC-Julia with BaseBenchmarks	Average overhead: 76% (optimization level O2), 22% (O0). Maximum overhead: 25,642% (O2), 26,154% (O0).
20	Performance evaluation of SC-Julia with BaseBenchmarks, without fences in generated code	Most of the overhead of SC-Julia comes from the cost of hardware fences.
22	Performance evaluation of SC-Julia with `@drf` annotations	Simple usage of `@drf` annotations cuts performance overhead in half.
24	Performance evaluation of SC-Julia with different fence instructions	Using MFENCE instead of XCHG roughly doubles overhead.
28, 29	Performance evaluation of S-VBD with DaCapo benchmarks	Geometric mean of overhead: 51% (machine A), 37% (machine B). Maximum overhead: 78% (machine A), 73% (machine B).
30	Startup performance of VBDA-HotSpot and S-VBD with DaCapo benchmarks	As expected, the use of deoptimization causes S-VBD to have a significantly higher impact on startup performance than VBDA-HotSpot.
31	Performance evaluation of just the dynamic checks of S-VBD with DaCapo benchmarks	The cost of the checks required by the speculative approach is considerable, on its own incurring well over half of the overhead.
32	Performance evaluation of S-VBD with `spark-tests` and `mllib-tests` benchmarks	S-VBD provides little speedup for these benchmarks.

reduce the overhead. For instance, by adding 20 method annotations in the DaCapo benchmarks one can reduce the average overhead of enforcing `volatile`-by-default from 28% to 18% on x86-64, our server machine. Similarly, adding annotations to Julia programs in a straightforward manner cuts the average overhead in half.

Second, we describe results for the baseline and speculative implementations for Java, on ARM-v8 hardware. We demonstrate that overhead on ARM-v8 is roughly double that for x86-64, which is significant but somewhat modest given that ARM-v8's memory model is considerably weaker than that of x86-64. Further, our speculative implementation strategy reduces the performance overhead on ARM-v8 versus the baseline JVM by roughly one-third.

We summarize key findings of our experiments in Table 1.

1.3 Article Organization

The article is organized as follows: Section 2 provides necessary background information on memory models and language implementations. Section 3 describes the `volatile`-by-default semantics for Java as well as its baseline implementation and evaluation. Section 4 does the same for Julia. Section 5 describes and evaluates the speculative implementation of `volatile`-by-default for Java on ARM-v8. Section 6 discusses related work, and Section 7 concludes.

2 BACKGROUND

In this section, we provide necessary background on memory models, focusing on the memory models of Java and Julia. We then overview the implementations of the HotSpot JVM and Julia compiler, which we modify to support the `volatile`-by-default semantics for Java and Julia, respectively.

2.1 Memory Models

The design of a memory model involves an inherent programmability vs. performance tradeoff. Stronger memory models like **sequential consistency (SC)** admit fewer behaviors, so they are relatively easier for programmers to understand and provide important guarantees for all programs. However, the restrictions on allowed behaviors limit the scope of optimizations, which

has a performance cost. For example, SC must restrict compiler optimizations that may reorder memory accesses, such as ***common subexpression elimination (CSE)***, *code motion*, and *dead store elimination*, and similarly for hardware optimizations like *write buffers*. For this reason, mainstream languages today have memory models that are weaker than SC, thereby trading off some programmability and safety in favor of performance.

2.1.1 The Java Memory Model. The Java memory model was defined more than 15 years ago [42] 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" [42]. 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 [5], whereby SC is only guaranteed for *data-race-free* programs. We say that two memory accesses *conflict* if they access the same variable, occur on two different threads, and at least one of the accesses is a write. Informally, a program is considered to be data-race-free if all concurrent conflicting accesses are to variables that are declared `volatile`. The JMM guarantees SC semantics for `volatile` variables, thereby ensuring that conflicting accesses on them are not actually concurrent: They will occur in some sequential order. Doing so requires implementations to disable many compiler optimizations for these variables and to emit fence instructions or special "synchronizing" load and store 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 for most variables.

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" [42]. In the presence of data races, some compiler optimizations 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 [42]. The JMM's causality is known to disallow some optimizations that it was intended to allow, notably common subexpression elimination [19, 59]. 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.

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" [42]. 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 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. This ability to access partially constructed objects is a known security vulnerability.[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 [56] and Dekker's mutual exclusion algorithm.

2.1.2 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 20 here: SOLR-13465, YARN-10185, SHIRO-762, CASSANDRA-2490, HDFS-566, OAK-3638, YARN-8323, HDFS-4106, AMQ-6251, ARTEMIS-1945, SPARK-4282, SLIDER-101, SPARK-3567, LOG4J2-247, POOL-11, HDFS-1207, CASSANDRA-11984, AMQ-6495, APEXMALHAR-1887, OWB-529. Each bug contains the project name and the bug ID. Its details can be found at https://issues.apache.org/jira/browse/<ProjectName>-<BugID>. The 20 issues listed here are returned by a search in the issues tracker as of May 2020. These errors occur in popular systems such as the Cassandra database, the HDFS distributed file system, and the Spark system for big-data processing[6] and can thereby impact the applications that employ these systems.

There are many different kinds of concurrency-related programming errors, and they are often grouped together as *race conditions*. Examples 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. What distinguishes missing-annotation bugs from the others is that

[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html.
[5] https://wiki.sei.cmu.edu/confluence/display/java/TSM03-J.+Do+not+publish+partially+initialized+objects.
[6] Spark is implemented in Scala, which compiles to Java bytecode and inherits Java's memory model.

the *only* reason that they are bugs is because of the language's weak memory model: They are correct under SC. Yet as Boehm et al. [16] and our earlier examples point out, missing-annotation bugs are far from "benign" but rather can cause surprising and harmful behaviors.

2.1.3 The Julia Memory Model. The Julia programming language is a recent and increasingly popular language that aims to provide the flexibility of a dynamic language while retaining performance comparable to traditional static languages. It is open source and is under active development.

Recently, Julia announced the addition of composable multi-threaded parallelism to the language [13]. Currently, however, there is no explicit memory model for Julia. Instead, the memory model is implicitly determined by the optimizations that the Julia compiler as well as the underlying hardware perform. Because Julia's compiler is based on top of LLVM, in practice this means that Julia's memory model is similar to that of C++ [11]. Like Java, C++ provides SC for data-race-free programs, and it provides an annotation called `atomic` that plays a similar role as Java's `volatile` annotation. Unlike Java, however, a C++ program with data races is considered to have undefined behavior, with no guarantees whatsoever. As a result, the kinds of issues described above for Java also exist in Julia today, and likely Julia's implicit memory model is weaker than Java's.

2.2 The HotSpot Java Virtual Machine

Oracle's HotSpot JVM is an implementation of the Java Virtual Machine Specification [27]. It is widely used and part of the OpenJDK—the official reference implementation of Java SE.[7]

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 includes one interpreter and 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. In our implementation of the `volatile`-by-default semantics for Java, we have modified the interpreter and the C2 compiler (and do not use the C1 compiler at all).

2.2.1 The HotSpot JVM Interpreter. The HotSpot JVM uses a *template-based*, or *threaded code*, interpreter. Specifically, 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).[8] 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.

[7] Our work is based on OpenJDK 8u. All specific descriptions of any technical details in this article are based on this version.
[8] The labels (1) and (2) can be ignored and will be referenced in the next section.

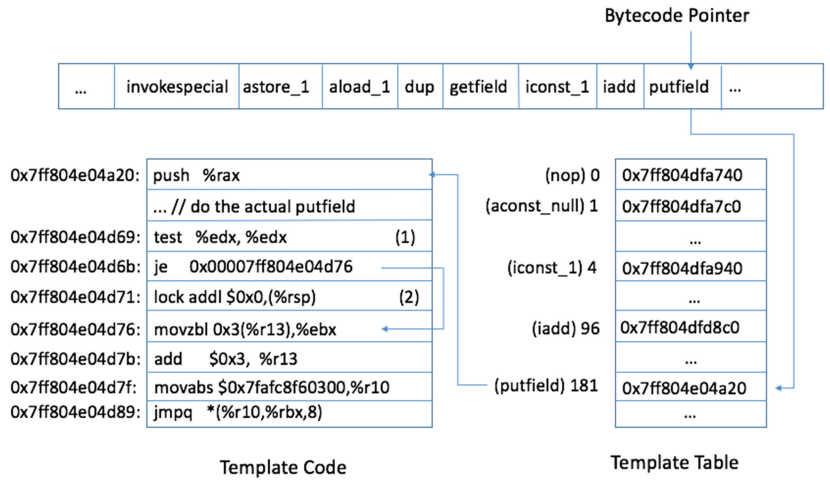

Fig. 1. Interpretation example of bytecode `putfield`.

2.2.2 The HotSpot JVM 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.

2.3 The Julia Compiler

Like Java, Julia also employs an interpreter along with JIT compilation. The Julia runtime uses simple heuristics to decide when code should be compiled. To perform that compilation, Julia employs the LLVM toolchain [36]. LLVM is a compiler infrastructure that provides a collection of industrial-strength compiler capabilities. The Julia compiler functions as an LLVM front-end: When it determines that some code should be compiled, the Julia compiler first generates a Julia-level IR and performs some simple optimizations. After that, the Julia compiler translates the Julia IR code into the LLVM IR. The LLVM execution engine then executes a sequence of LLVM *passes* to analyze, optimize, and generate the native instructions to execute. The subset of passes and their ordering are specified by the Julia compiler. The Julia compiler also includes some Julia-specific passes that are executed during this stage.

3 A VOLATILE-BY-DEFAULT JAVA VIRTUAL MACHINE

This section describes the design, implementation, and evaluation of our baseline `volatile`-by-default JVMs for both the x86-64 and ARM-v8 hardware platforms.

3.1 Design

As described in the previous section, 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. Our `volatile`-by-default semantics simply flips the default: All variables are treated as if they were declared `volatile`. Therefore, 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 methods, fields, 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 [4]. 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 optimizations 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 [43]. First, as mentioned above, all accesses to Java primitive values are atomic. 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.

It is worth noting that the `volatile`-by-default guarantees are only provided *by default*; a program that employs `relaxed` annotations may violate these properties. Further, in general it is difficult to understand the precise semantic impact of a particular `relaxed` annotation, since it has the effect of admitting weak behaviors that pertain to the interactions among multiple threads. However, we argue that the `volatile`-by-default approach provides important benefits despite these limitations.

First, this situation is exactly analogous to that for type and memory safety today. For example, almost every Java program will invoke functionality implemented in native code. In the presence of such code, it is possible that a Java program will violate type or memory safety. Yet the fact that the language enforces type and memory safety by default provides huge benefits to security and reliability in practice. Second, we can state a clear correctness requirement for `relaxed` code: that code must not participate in any data races. If all `relaxed` code satisfies this property, then the program as a whole will be data-race-free and hence will enjoy the guarantees of the `volatile`-by-default memory model. While in general determining whether a piece of code can participate in a data race is difficult, the JMM already requires programmers to do this if they want to ensure SC for their programs. Under the `volatile`-by-default memory model, however, this hard work only needs to be done for code that is declared `relaxed`. Third, no matter what, `relaxed` annotations cannot introduce behaviors that violate the JMM, as by design `relaxed` code respects the JMM (and so does non-`relaxed` code, as it has even stronger requirements than the JMM).

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.

3.2 Implementation

The most 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 [8]. But it was not clear how the empirical observations from such a JVM would translate to a production JVM implementation.

Therefore, we opted to instead implement the `volatile`-by-default semantics through a modification to 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 for both x86-64 and aarch64 (64-bit ARM-v8). The modified version for x86-64 is called VBD-HotSpot and the one for ARM-v8 is called VBDA-HotSpot. Both versions add a flag `-XX:+VBD` that allows users to obtain `volatile`-by-default semantics.

As mentioned in Section 2.2, we have modified the HotSpot interpreter as well as the HotSpot server compiler. Our modifications reuse the mechanisms already in place for handling `volatile` variables, as described next.

3.2.1 `volatile`*-by-default Interpreter.* Since the interpreter is platform-specific, different fence instructions are used for VBD-HotSpot and VBDA-HotSpot [29]. We will first talk about how we make the interpreter `volatile`-by-default on x86-64 and then present our modifications to the interpreter on ARM-v8.

x86-64. Figure 1 from the previous section shows how the HotSpot JVM handles accesses to `volatile` variables on x86-64. The SC semantics for `volatile` accesses is achieved by inserting the appropriate platform-specific fences before/after such accesses. In the case of x86-64, which has the relatively strong ***total store order*** **(TSO)** semantics [50], 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 [29]. In the figure, %edx is already loaded with the field attribute for `volatile`. Instruction (1) tests if the field is declared `volatile`. If so, then the `lock addl` instruction (2) will be executed, which acts as a StoreLoad barrier on x86-64; otherwise, the `lock addl` instruction is skipped.

To implement our `volatile`-by-default semantics for x86-64, we therefore modified the template for `putfield` to unconditionally execute the `lock addl` instruction. This is done by removing instruction (1) and the following jump instruction `je`. We also added the `lock addl` 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).

We manually inspected the template instructions for the interpreter's implementation of all bytecodes that read from or write to memory: `getfield`, `putfield`, `fast_xgetfield`,

`fast_xputfield`, and `fast_xaccess`. The latter three bytecodes are used internally by the HotSpot JVM as special, more efficient versions of `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 VBD-HotSpot, we elide the check of the `volatile` attributes and always add the necessary fences.

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 to unconditionally execute the `lock addl` 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 x86-64 implementations of these intrinsics and found that none of them contain writes to shared memory, so they already preserve SC.

ARM-v8. Similar to the case for x86-64, we manually inspected the template instructions in the ARM-v8 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. 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`. Additionally, we examined and modified bytecode-rewriting optimizations and intrinsics in the interpreter in VBDA-HotSpot the same way we treated VBD-HotSpot.

To implement the semantics of `volatile` on ARM-v8, the Java interpreter inserts 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 [29]. The interpreter uses ARM-v8's **`dmb` (data memory barrier)** instruction for this purpose. In particular, it uses 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.[9] We have fixed this bug and use the fixed version of the baseline HotSpot JVM in all of our experiments.

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

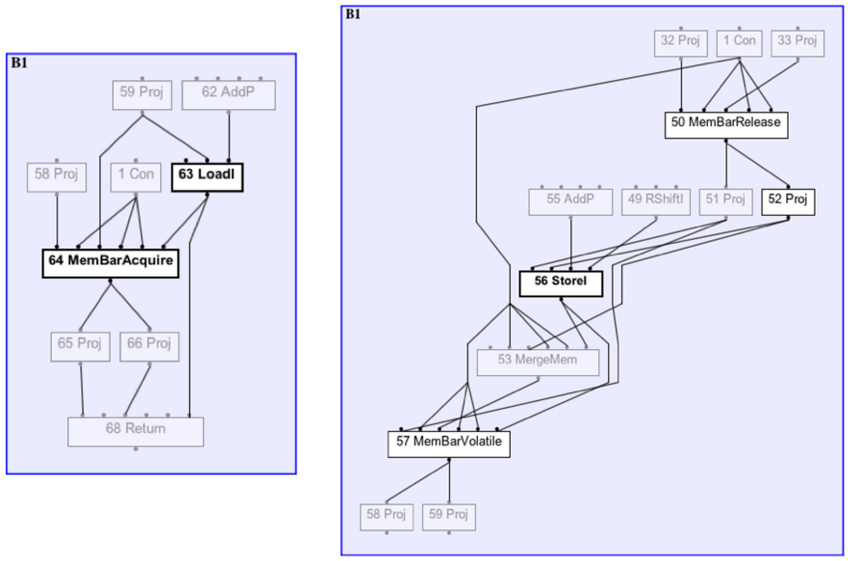

Fig. 2. Ideal graph sections for `volatile` loads (left) and stores (right).

3.2.2 `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 has several phases. First a hot spot's bytecode instructions are translated into a high-level graph-based **intermediate representation (IR)** called Ideal, which is platform-independent. The Ideal graph is then translated to a lower-level platform-specific IR, and finally machine code is generated. Optimizations are performed during all of these phases.

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.[10]

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 x86-64 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 to prevent the compiler from performing optimizations that violate their semantics. On ARM-v8, `MemBarAcquire` becomes a `dmb ish ld`, and both `MemBarRelease` and `MemBarVolatile` become `dmb ish`.

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

[10]On the POWER processor [41], which is not multi-copy atomic, a `MemBarVolatile` also precedes each `volatile` load, but this is not necessary for x86-64 or ARM-v8.

interpreter, this was done both for accesses to instance variables and to array elements. 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. Further, because the ideal graph is platform-independent, modifying the compiler at this stage suffices to handle both x86-64 and ARM-v8—we simply rely on the existing backends for these platforms to compile the memory-barrier nodes appropriately. The downside of this approach is that it is overly conservative. For example, under SC it is safe to eliminate a load that is immediately preceded by a store to the same memory location, but surrounding these accesses with memory-barrier nodes has the effect of preventing the optimization.

One 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 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.

3.2.3 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.[11]

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 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.

3.2.4 `relaxed` *Annotations.* To implement `relaxed` annotations, we added a flag to specify a list of methods that should be treated as `relaxed`. When a field access or an array access is compiled, we check that the compiled method is not on the list before enforcing SC semantics. We added similar flags to allow users to specify lists of fields and classes that should be treated as `relaxed`. We check the list of classes as well as the list of fields before enforcing SC semantics on a field access.

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

3.2.5 Correctness. Our main implementation technique, in both the VBD-HotSpot and VBDA-HotSpot interpreter and compiler, is to simply reuse the existing mechanisms for handling accesses to `volatile` variables. Therefore, the correctness of VBD-HotSpot and VBDA-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 more than 30 litmus tests that can exhibit non-SC behavior under the unmodified HotSpot JVM, including many existing litmus tests from the literature on weak memory models [55]. The litmus tests can be found on Github: https://github.com/Lun-Liu/schotspot-aarch64/tree/master/litmustests. We ran these litmus tests hundreds of thousands of times on the current VBD-HotSpot and VBDA-HotSpot compilers and they have never exhibited a non-SC behavior, which helps lend confidence in our implementation.

3.2.6 `volatile`*-by-default for Java and Scala.* Finally, we note that VBD-HotSpot and VBDA-HotSpot ensure `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 and tested the widely used `javac` compiler that is part of the OpenJDK, which compiles Java source to bytecode, and found no optimizations that can violate SC. This is not surprising, since by design `javac` performs only a few simple optimizations, which we detail below, deferring sophisticated optimizations to the bytecode JIT compiler. Hence compiling a Java program with `javac` and executing the resulting bytecode with VBD-HotSpot or VBDA-HotSpot provides `volatile`-by-default semantics for the original program.

Specifically, we manually inspected each pass in `javac`, and for each pass, we also wrote one or more small test programs to concretely see the bytecode produced as a result of the pass. There are two kinds of passes. First, there are passes that perform *desugarings* of Java language features. These include passes that eliminate usage of generic types, translate lambdas into methods, perform boxing/unboxing of primitives, and translate for-each loops into regular loops. Second, there are passes that prepare for bytecode generation by making several things explicit that are otherwise implicit. Most notably, inner classes must be hoisted to the top level, which requires many other changes: The class name is mangled and then updated everywhere it is used, the class is provided with extra fields to be able to refer to parent objects, and so on. Last, there is a final pass that generates bytecode in a straightforward manner.

In terms of optimizations, the `javac` compiler has no support for general dataflow analyses or transformations. Rather, the *only* optimizations are local ones pertaining to compile-time constants. Specifically, the translation of conditionals performs *branch folding* when the guard of the conditional is a known compile-time constant, the translation of logical operations like `||` and `&&` do the same, and an access to a variable that is a compile-time constant is replaced by its value. Here a compile-time constant is either a constant value (e.g., `true`), an expression involving only compile-time constants (e.g., `3+5`), or a `final` variable or field that is initialized with a compile-time constant. To potentially violate SC, an optimization must reorder accesses to heap memory. (Reordering stack accesses with one another, and even reordering a stack access with a heap access, cannot violate SC.) Of the `javac` optimizations, only the removal of a field access pertains to the heap. However, the restrictions on that field ensure that all reads in the original program must see the same constant value, so the optimization does not introduce any non-SC behaviors.

We similarly examined the `scalac` compiler that compiles Scala source to Java bytecode[12] and also found no optimizations that reorder memory accesses, so the same guarantees hold for Scala

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

programs running on VBD-HotSpot or VBDA-HotSpot. Like `javac`, the `scalac` compiler has no support for general dataflow analyses or transformations. In addition to optimizations on compile-time constants, `scalac` also includes a few peephole optimizations specifically related to boxing and unboxing. For example, an expression of the form `new Integer(3) == new Integer(4)` is simplified to `3 == 4`. We have manually inspected these optimizations and found that they do not violate SC. Specifically, whenever the optimizations apply to heap accesses, the associated objects are known to be *thread local*, as in the example shown above.

3.3 Performance Evaluation

We first present the performance evaluation of our x86-64 implementation VBD-HotSpot, and then we present the performance evaluation of our ARM-v8 implementation VBDA-HotSpot.

3.3.1 Benchmarks and Methodology. We have used two benchmark suites for our performance evaluation: the DaCapo benchmark and `spark-perf`.

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 [15]. We use the DaCapo 9.12 distribution for VBD-HotSpot evaluations. We exclude 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,[13] which VBD-HotSpot employs (see below); and *lusearch* has a known concurrency error that causes it to crash periodically. We use the latest maintenance release (9.12-MR1) of the DaCapo benchmarks from January 2018 for VBDA-HotSpot evaluations. 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 as described earlier; and *tomcat*, due to a problem unrelated to DaCapo.[14] 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.

For all DaCapo tests, we ran the DaCapo benchmarks on our server machine and used the default workload and thread number for each benchmark. We used no `relaxed` annotations whatsoever, including in the Java standard library, unless specified otherwise. Therefore, the experiments represent extreme points and cases of sorts for the `volatile`-by-default memory model, since in practice we expect that judicious usage of `relaxed` will be used to increase performance where appropriate. We used an existing methodology for Java performance evaluation [25]. 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 VBD-HotSpot evaluations and 5 JVM invocations for VBDA-HotSpot evaluations (since execution on our ARM-v8 machine is much slower) for each test and calculated the average execution time of the averages, and calculated the 95% confidence interval using the averages from the JVM invocations.

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 and VBDA-HotSpot on two benchmark suites for Apache Spark [65], 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[15]: `spark-tests` includes several big-data analytics applications, and `mllib-tests` employs Spark's

[13] https://bugs.openjdk.java.net/browse/JDK-8067708.
[14] https://bugs.openjdk.java.net/browse/JDK-8155588.
[15] 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.

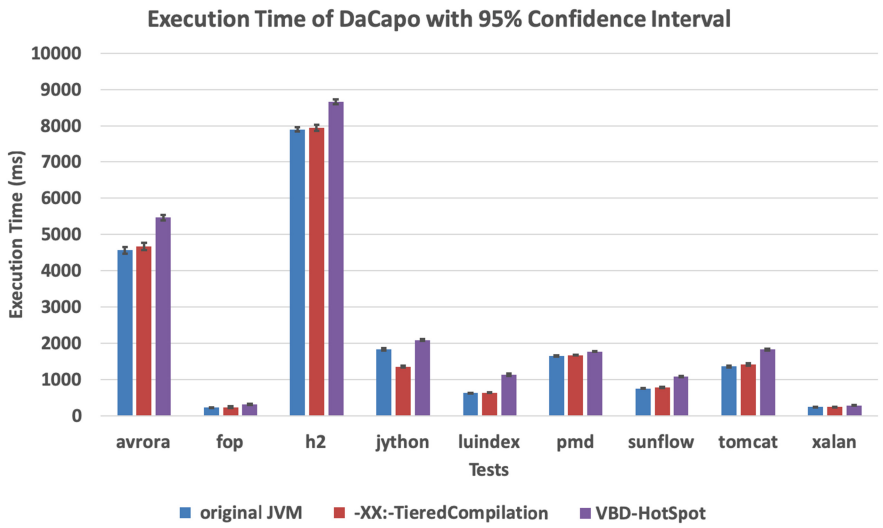

Fig. 3. Execution time in milliseconds of VBD-HotSpot on the DaCapo benchmarks. "original JVM" means running the baseline HotSpot JVM without additional flags; "`-XX:-TieredCompilation`" means running the baseline HotSpot JVM with `-XX:-TieredCompilation`; "VBD-HotSpot" shows results of running VBD-HotSpot.

MLlib library [46] 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.

For `spark-perf`, 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. Further, as with the DaCapo experiments, we use no `relaxed` annotations, in either the application code or the Spark library code. In our experiments, the executor memory is 4 GB and the driver memory is 1 GB. 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 for VBD-HotSpot and 5 invocations for VBDA-HotSpot and calculated the average of the median execution time.

3.3.2 VBD-HotSpot. In this section, we describe our experiments that provide insight into the performance cost of SC for JVM-based server applications on x86, 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 12-core machine with two Intel Xeon E5-2620 v3 CPUs (2.40 GHz) with hyperthreading, which provides 24 processing units in total.

DaCapo Benchmarks. We used the methodology mentioned in Section 3.3.1 for DaCapo benchmarks on our server machine. 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.

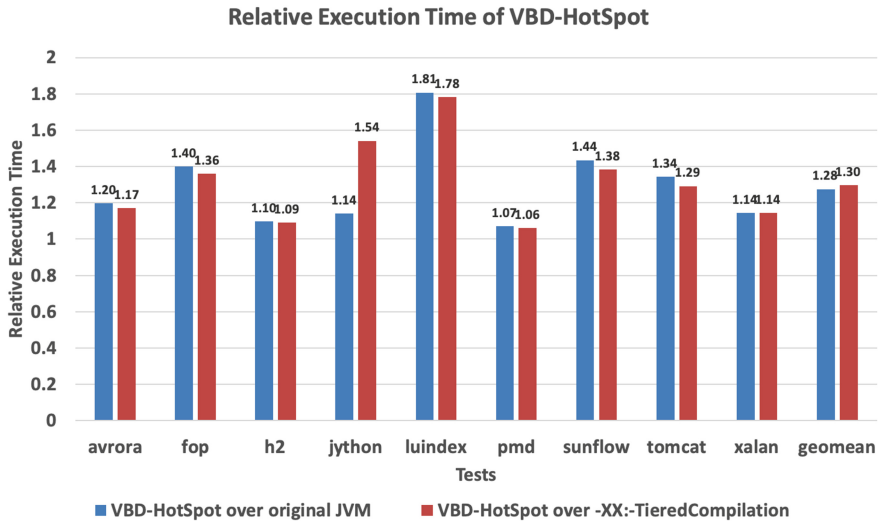

Fig. 4. Relative execution time of VBD-HotSpot on the DaCapo benchmarks.

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 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.[16] 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 multithreaded programs is due to the fact that multithreaded programs already must use synchronization, for example locks and `volatile` annotations, 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.

Of course, if the programmer is aware that their program is single threaded (or has limited concurrency such that it is obviously data-race-free), then they 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.

Spark Benchmarks. We used the methodology mentioned in Section 3.3.1 for `spark-perf` on our server machine. Figure 5 shows the median execution times for the eight `spark-tests`

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

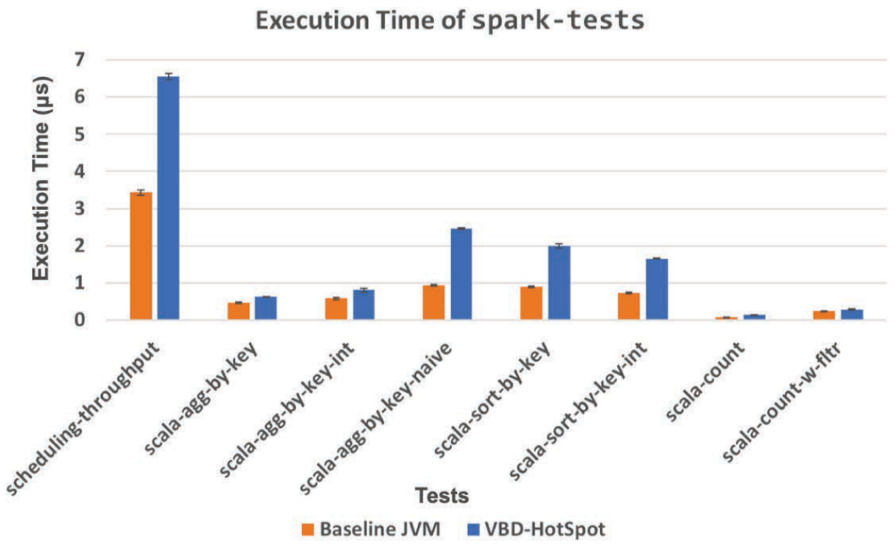

Fig. 5. Median execution time in seconds for `spark-tests`.

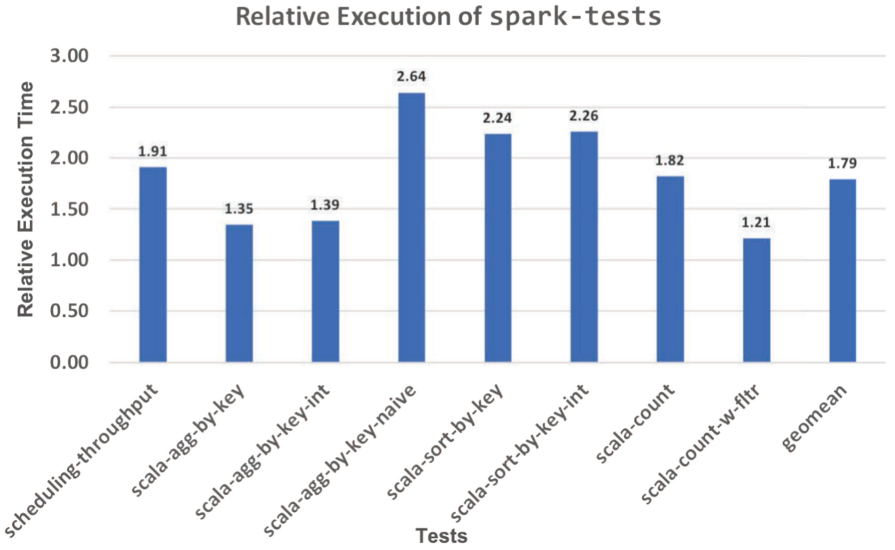

Fig. 6. Relative execution time of VBD-HotSpot over the baseline JVM for `spark-tests`.

benchmarks when run on the original HotSpot JVM as well as on VBD-HotSpot, with 95% confidence intervals. Figure 6 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 DaCapo benchmarks. We surmise that the large overhead for Spark benchmarks is due Spark's dataflow programming model using the ***resilient distributed dataset*** **(RDD)** abstraction, which is an in-memory, immutable data structure. Each Spark operation in a program potentially incurs many memory operations to read its input RDDs and write its output RDDs.

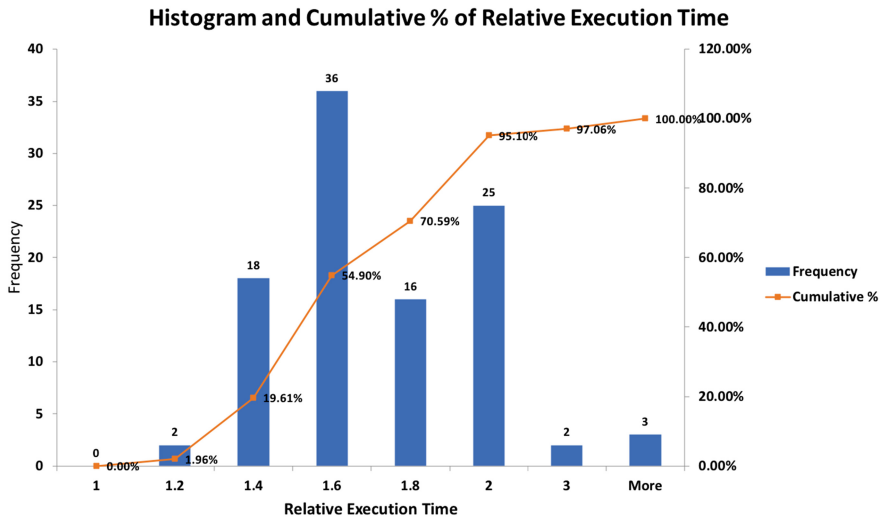

Fig. 7. Histogram and cumulative % of relative execution time for `mllib-tests`.

Figure 7 shows the results for the `mllib-tests` benchmarks, as a histogram. For example, the third bar from the left indicates that there are 18 benchmarks that incur a relative execution time between 1.4 (exclusive) and 1.6 (inclusive). Cumulatively, we see that 54.90% of the benchmarks incur an overhead of 60% or less, and 95.10% of the benchmarks incur an overhead of 100% or less. 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 1.67, or a 67% slowdown. These results are consistent with those for the `spark-tests` benchmarks shown above.

Scalability Experiments. We performed an experiment to understand how the cost of the `volatile`-by-default semantics changes with the number of threads/cores available. Since committing a memory operation involves inter-core communication (e.g., cache 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. But, in fact, we find that this is not the case.

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. For this experiment, 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 3.3.2. It also does not apply to *avrora* and *pmd*—though these benchmarks use multithreading internally, they always use a single driver thread. Therefore, our experiments only employed the remaining four DaCapo benchmarks.

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 8. 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 9 provides the results

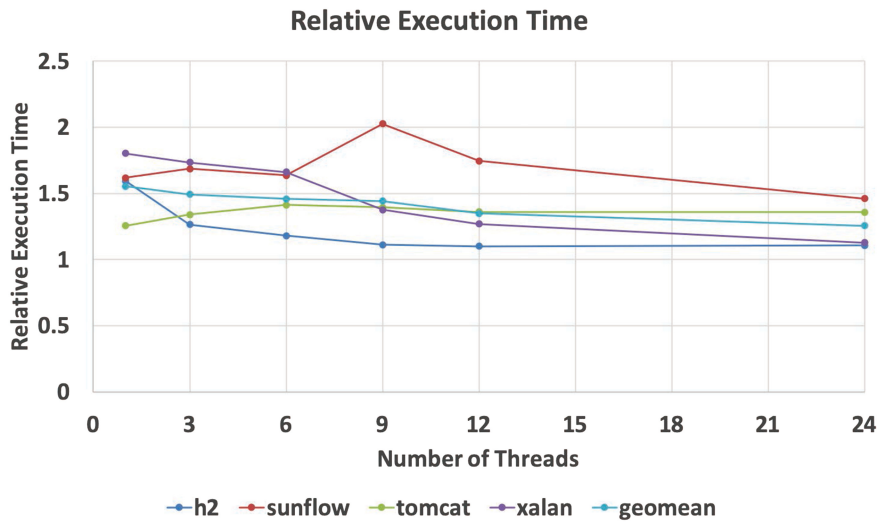

Fig. 8. Relative cost of VBD-HotSpot with different numbers of threads/cores.

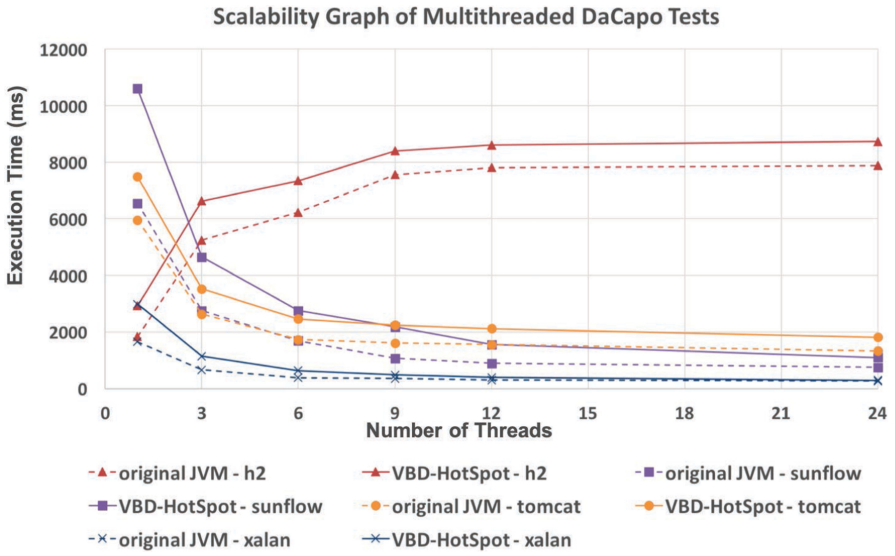

Fig. 9. Scalability graph with different numbers of threads/cores.

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 imply that SC performance does not suffer with increased concurrency. They also accord with an experiment by a previous work [17] showing that a lock-based version of a particular parallel algorithm scales better than a version with no synchronization.

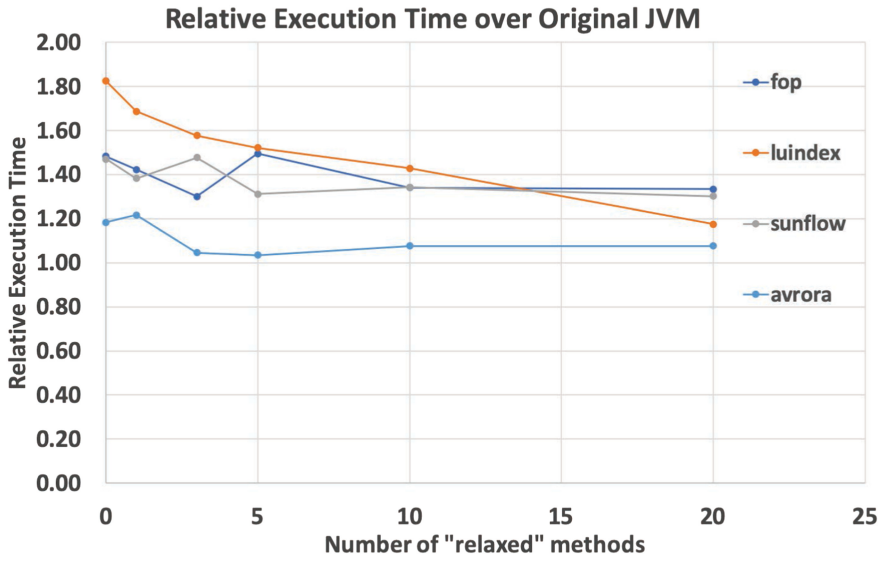

Fig. 10. Cost of VBD-HotSpot with `relaxed` methods.

Relaxed Execution. We also performed experiments to gauge the potential for usage of `relaxed` annotations to improve the performance of VBD-HotSpot. We profiled four of the five DaCapo benchmarks that incur the most overhead for VBD-HotSpot[17] to determine the methods in which each benchmark spends the most execution time. Figure 10 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. One benchmark has particularly dramatic reductions in overhead: *luindex* reduces from 1.82 to 1.17. Many of the top methods are in the Java standard library and so could be declared `relaxed` once and then used by many applications. Overall, with 20 `relaxed` annotations each on the five benchmarks in Figure 10, the geometric mean of VBD-HotSpot's overhead reduces to 18% for the entire DaCapo suite (with a max overhead of 34% for *tomcat*). These results indicate that targeted usage of `relaxed` annotations in performance-critical code can be a valuable tool in making the `volatile`-by-default semantics a practical choice for programmers today.

Consumer PCs. Finally, we also ran our benchmarks on several consumer PC machines, in addition to our server machine. PC1 is a six-core machine with an Intel Core i7-3930K CPU (3.20 GHz), which was released in the fourth quarter of 2011. PC2 is a four-core machine with an Intel Core i7-4790 CPU (3.20 GHz), which was released in the second quarter of 2014. PC3 is a four-core machine with an Intel Core i7-6700 CPU (3.40 GHz), which was released in the third quarter of

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

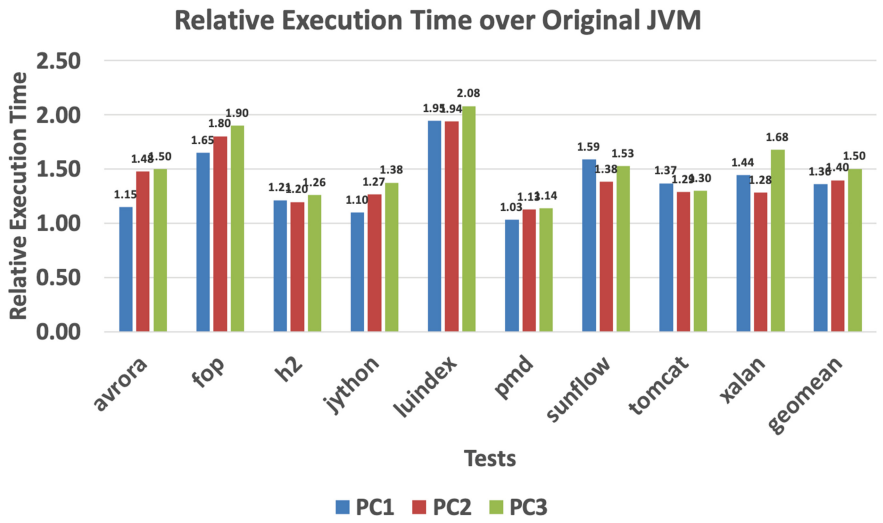

Fig. 11. Relative execution time of the DaCapo benchmarks.

2015. Hyperthreading is enabled on all three machines. Therefore, we have 12, 8, and 8 processing units for PC1, PC2, and PC3, respectively.

We ran the DaCapo benchmarks on these machines using the same setup as in Section 3.3.1. Figure 11 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 28% overhead of VBD-HotSpot on our server machine (Section 3.3.2).

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 is 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.

3.3.3 VBDA-HotSpot. We also 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-v8 servers: Machine A has eight Cortex A57 cores, 16 G memory, and is running openSUSE Tumbleweed; machine B has two Cavium ThunderX CN8890 CPU (96 cores in total), 128 G memory, running Ubuntu 16.04.

DaCapo Benchmarks. We used the methodology mentioned in Section 3.3.1 for DaCapo benchmarks. 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, which, respectively, perform a load with acquire semantics and a store with release semantics. Indeed, these *one-way* fence instructions [9] were introduced in ARM-v8 in part to support `volatile` accesses more efficiently. If the backend cannot identify that a memory-barrier node is part of a `volatile` read or write, then it employs the regular *two-way* fence instruction (`dmb`) for that

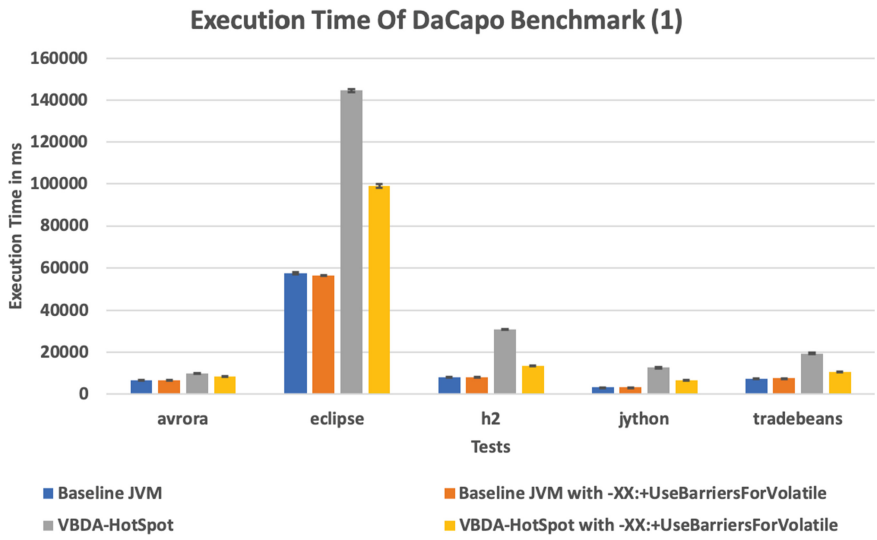

Fig. 12. Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

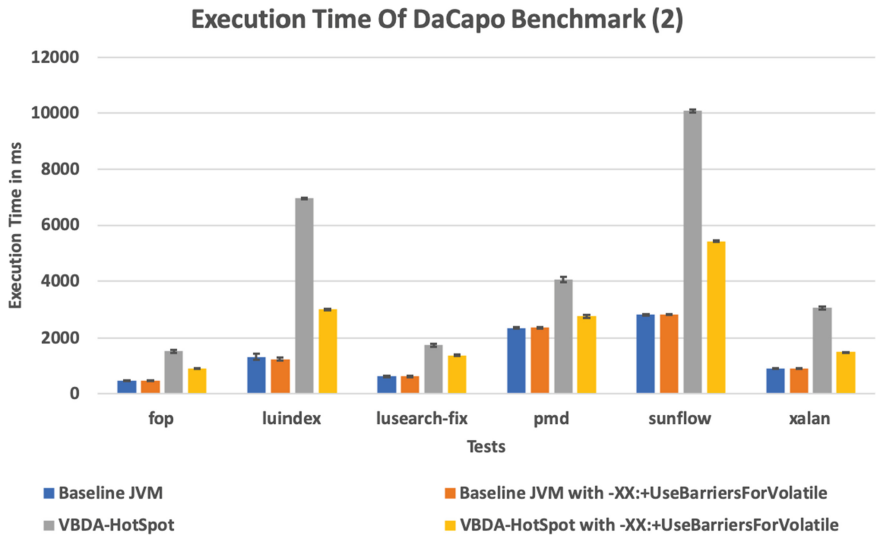

Fig. 13. Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

node, as we do for the VBDA-HotSpot interpreter (Section 3.2.1). 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 employ and evaluate both approaches.

Figures 12 and 13 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 to turn off tiered compilation, as described earlier for the VBD-HotSpot experiments. We have verified that for the baseline HotSpot JVM, there is very little performance difference with and without tiered compilation on the DaCapo benchmarks.

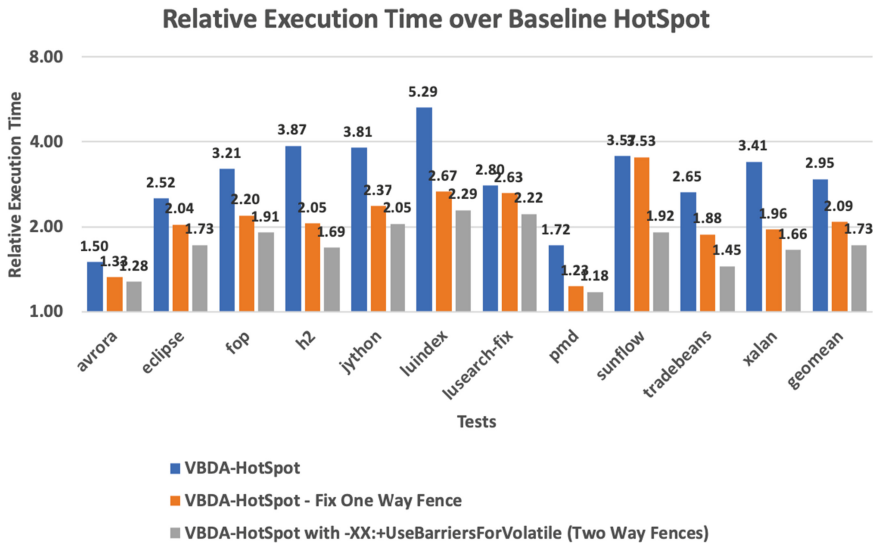

Fig. 14. Relative execution time of VBDA-HotSpot, VBDA-HotSpot with bug fix for one way fences, VBDA-HotSpot with two way fences on machine A, y-axis in logarithmic scale.

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. So it appears that the ARM-v8 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.[18] Fixing those heuristics as suggested in the bug report makes a dramatic difference, as shown in Figure 14, 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 article, we report numbers with the `-XX:+UseBarriersForVolatile` flag for VBDA-HotSpot.

The first series in Figures 15 and 16, 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 14). 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

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

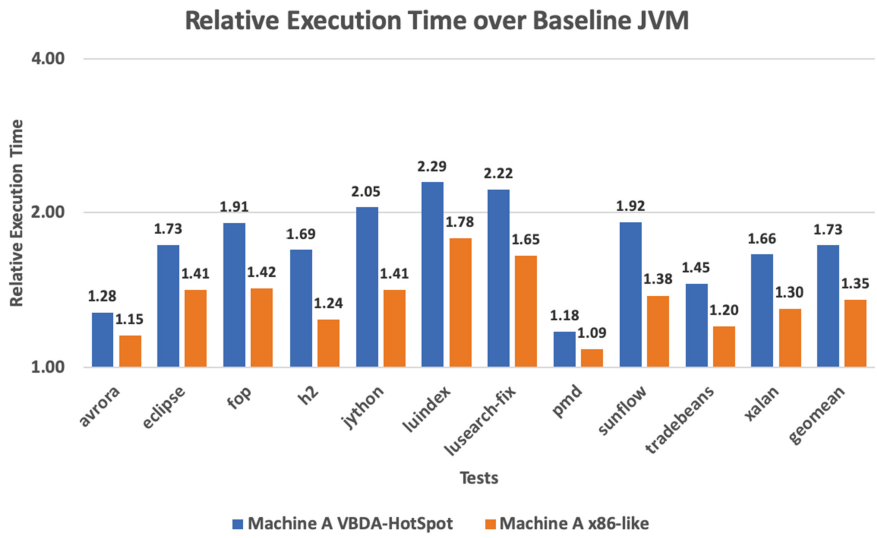

Fig. 15. Relative execution time of VBDA-HotSpot and x86-like VBDA-HotSpot over baseline JVM for DaCapo on machine A, y-axis in logarithmic scale.

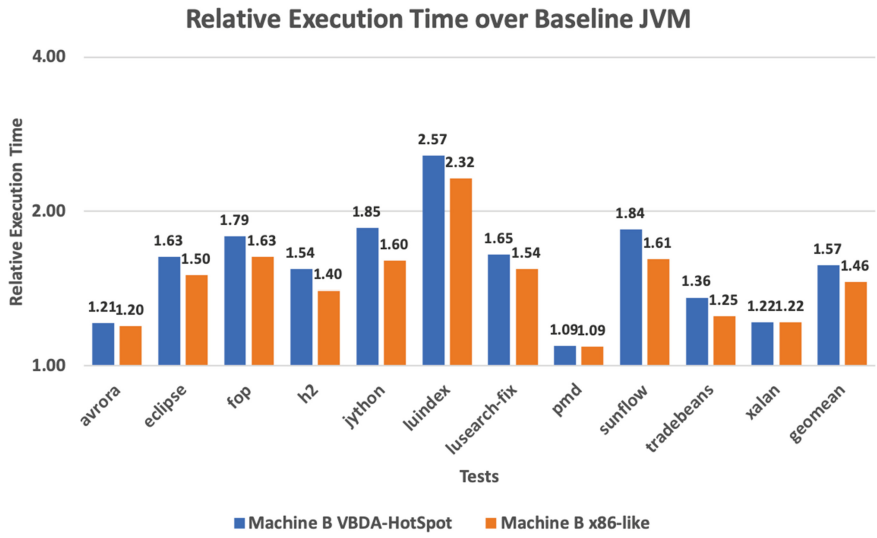

Fig. 16. Relative execution time of VBDA-HotSpot and x86-like VBDA-HotSpot over baseline JVM for DaCapo on machine B, y-axis in logarithmic scale.

compiler. This version of VBDA-HotSpot does not guarantee SC and so is only introduced as a way to better understand the costs of the full VBDA-HotSpot implementation. The second series in Figures 15 and 16, respectively, shows the 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

Fig. 17. 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.

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.

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 we did for VBD-HotSpot above. There, we saw that the relative overhead of `volatile`-by-default decreases or stays the same as the number of cores increases on x86-64, 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-v8 platform.

Machine A has four sockets, each with 2 cores. Machine B has two sockets, each with 48 cores. 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 the execution the same way we did for the scalability experiments in Section 3.3.2. We choose the benchmarks in DaCapo which exhibit external concurrency, namely, `h2`, `lusearch-fix`, `sunflow`, `xalan`.

Experiments on both machines show a similar trend as for VBD-HotSpot: 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 17; the results for machine A are similar. The figure shows how the relative execution time of VBDA-HotSpot changes with different numbers of driver threads. Interestingly, VBDA-HotSpot is faster than the baseline HotSpot JVM for the `xalan` benchmark at 36 and 48 threads. By examining the absolute execution times, we observe the reason: The baseline JVM stops scaling for `xalan` at 24 threads, with performance remaining flat or slightly degraded after that, while VBDA-HotSpot continues scaling until 48 threads on the benchmark.

Spark Benchmarks. Finally, we tested VBDA-HotSpot's performance on the `spark-tests` and `mllib-tests` benchmarks for Apache Spark, as we did for VBD-HotSpot. Again, we used the same methodology.

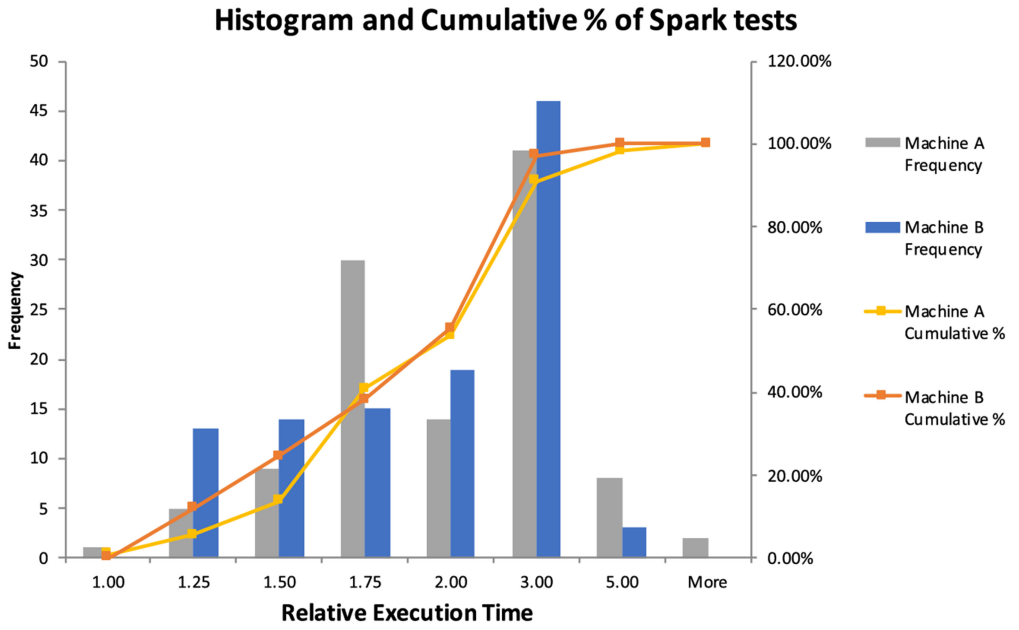

Fig. 18. Histogram and cumulative % of relative execution time of VBDA-HotSpot for Spark benchmarks.

Figure 18 summarizes the results in a histogram. For example, the second gray bar from the left indicates that on Machine A there are five 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 on x86-64. As mentioned earlier, we suspect that the high overhead for Spark tests versus the DaCapo benchmarks is due to the many memory reads and writes that are necessary to implement Spark's key data structure, the **resilient distributed dataset (RDD)** [65], which is implemented using arrays. 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%.

3.3.4 Performance Evaluation Summary. These results show that, while there is a non-trivial cost to enforcing sequential consistency by default, it is arguably much less than has been assumed until now. Even with zero `relaxed` annotations, which represents an extreme point in the way that the `volatile`-by-default memory model could be used, overheads on even the very weak ARM-v8 hardware platform are roughly a factor of two on average for our worst-performing benchmark suite. In practice, `relaxed` annotations would provide users with fine-grained control over the performance vs. safety tradeoff. Finally, on both x86-64 and ARM-v8, we see that concurrent code scales as well or better under `volatile`-by-default as it does under the Java memory model.

4 VOLATILE-BY-DEFAULT FOR JULIA

Thanks to its high-level syntax, script-like features such as dynamic typing, and good performance through aggressive optimization, the Julia language [12] is gaining popularity in fields such as scientific computation and data analysis. To illustrate the generality of our proposed

`volatile`-by-default memory model, as well as to obtain more data on its performance, we have built a `volatile`-by-default version of Julia as a modification to its compiler.[19]

As mentioned earlier, Julia does not have a memory model for its recently added multi-threading support [13]. However, Julia uses LLVM as the backend for the compiler and does not put additional work into enforcing a stronger memory model. Since the LLVM memory model is inspired by the C++0x memory model [2], it is reasonable to conclude that the Julia's memory model is weaker than SC. In fact, simple litmus tests like Dekker's algorithm do exhibit SC-violating behaviors in Julia.

However, like Java, we believe that Julia's design philosophy makes it an excellent candidate for the `volatile`-by-default memory model. Specifically, Julia is designed to be an accessible and easy-to-use language for programmers of varying backgrounds, and it already employs a *safe by default, performance by choice* approach to other forms of safety. For example, Julia performs bounds checking by default to ensure memory safety when accessing arrays. However, programmers can annotate critical loops with the `@inbounds` macro, indicating that all array accesses within these loops should be treated as in-bounds, and hence no bounds checks will be performed during execution.

4.1 Design

Our design of the `volatile`-by-default memory model for Julia largely follows that of Java, as described in the previous section. Since Julia employs the LLVM compiler infrastructure to compile functions, our memory model provides sequential consistency by default at the level of the LLVM **intermediate representation (IR)**. That is, by default all memory operations in the LLVM IR are sequentially consistent, in the same way that our design for Java ensures SC at the level of Java bytecode.

Following the *performance by choice* part of our philosophy, we also introduce a linguistic mechanism for programmers to trade off some safety guarantees for increased performance. Following the existing Julia style, we introduce a new macro called `@drf`, which a programmer can use to indicate that some portion of code should be considered data-race-free. A `volatile`-by-default version of Julia can safely perform all of the optimizations that the current Julia compiler performs for `@drf`-annotated code, and similarly there is no need to insert additional hardware fences during code generation. Our implementation supports `@drf` annotations on loops, functions, and entire modules in Julia.

4.2 Implementation

We have implemented a `volatile`-by-default version of the just-in-time compiler within the standard Julia implementation, with the x86-64 backend, and we call our version of this compiler SC-Julia. Our implementation is based on the v1.4.1 release of the Julia runtime on GitHub [30]. Additionally, we have changed the LLVM version used to 9.0.1 because of a patch needed to handle atomic operations correctly.[20]

Our implementation of SC-Julia has a few caveats that can potentially cause Julia programs to violate SC by default. First, we have modified Julia's just-in-time compiler to ensure SC by default, but we have not modified its interpreter. Second, our approach does not prevent optimizations within the Julia compiler itself, before translation to LLVM, from violating SC, and we have not inspected that part of the compiler. Despite these limitations, we believe that our implementation

[19]The source code of the compiler can be found on GitHub: https://github.com/JuliaLang/julia.
[20]https://github.com/llvm/llvm-project/commit/c5830f5f05a4ecb6ae0db0aa386af733f6113b77#diff-77f3e79090addeb629c840b569aee5803ca93afe78e2e7e7a40c6bdba7be59b6.

is useful for understanding the cost of SC for Julia. In particular, Julia uses simple JIT heuristics and so most of the code ends up being compiled rather than interpreted, and we expect that, as is the case with `javac` and `scalac` (see Section 3.2.6), there are few if any SC-violating transformations outside of LLVM.

As in our Java-based implementations of the `volatile`-by-default memory model, our main task is to transform all loads and stores in the LLVM IR to have SC semantics. To implement this transformation, we have created a new LLVM pass that rewrites each load and store to ensure sequential consistency. We considered instead performing this transformation inside the Julia compiler itself, when it generates the LLVM IR from its lowered-form IR. The potential advantage of that approach is access to higher-level program information that could be used for optimization. However, in turn, that approach would not be able to leverage the existing LLVM analyses for optimization. Further, as we describe later, the Julia compiler preserves some semantic information as metadata in the LLVM IR, which our pass can still exploit. Finally, our approach is more modular and hence less invasive, and it can potentially apply to other languages that employ LLVM as a backend.

In the rest of this subsection, we describe our implementation in detail. We first overview the structure of our new LLVM pass. Then, we discuss the LLVM instructions that we use to rewrite loads and stores to ensure SC. Finally, we describe optimizations that we implemented to reduce the performance cost.

4.2.1 LLVM Pass. We implemented our own LLVM pass called "`AddSC`." We install this pass as the first optimization pass in the pipeline specified by the Julia JIT compiler. This ensures that the following passes will respect the SC semantics when trying to optimize the IR. `AddSC` iterates through every basic block and rewrites each load and store instruction within each basic block. We detail the rewriting of loads and stores below. Our implementation of `@drf` annotations introduces metadata into the LLVM IR, which we use to skip rewriting for basic blocks that are within loops, functions, and modules that have been declared `@drf`.

4.2.2 Rewrite Loads and Stores. By default, LLVM loads and stores provide no guarantees beyond those of a weak memory model like that of C++: If there is a race on a given memory location, then loads from that location can have undefined behavior [1]. Standard single-threaded compiler and hardware optimizations are allowed on these memory accesses, which can lead to non-SC behaviors. The Julia compiler generates these default LLVM loads and stores for ordinary reads and writes in Julia programs and hence inherits this weak semantics.

However, to implement various forms of synchronization, LLVM supports multiple forms of *atomic* memory accesses [1], which provide different guarantees. For example, *Acquire* atomicity ensures that a load has lock-acquire semantics, *Release* atomicity ensures that a store has lock-release semantics and makes a store to have lock-release semantics, *SequentiallyConsistent* atomicity provides both *Acquire* semantics for loads and *Release* semantics for stores, and it also guarantees a total order among all *SequentiallyConsistent* accesses.

The obvious way to enforce the `volatile`-by-default semantics is then simply to rewrite every load and store in the LLVM IR to be *SequentiallyConsistent* atomic. And indeed, this is what our `AddSC` pass does. Unfortunately, it is not that simple, as some loads and stores in the LLVM IR are not allowed to be declared atomic [2]. For example, the pointee of atomic loads and stores must either be an integer, a pointer, or a floating-point, while non-atomic loads and stores are additionally allowed to point to aggregate types such as arrays and structures. The LLVM code generation process will fail if such accesses are declared to be atomic.

Hence, we need a new approach to ensure that loads and stores that cannot be declared atomic nonetheless have sequentially consistent behavior. We still mark these loads and stores as

SequentiallyConsistent during the `AddSC` pass, to ensure that downstream compiler optimizations treat them as such. In addition, we have implemented a second LLVM pass called `SCExpand`, which we install as the last optimization in the pipeline, right before code generation. During `SCExpand`, every load and store that cannot be declared atomic is converted back into a non-atomic access.

However, we are still left with the problem of ensuring that the hardware respects SC for these accesses. Therefore, during `SCExpand`, we add an *Acquire* fence after each non-atomic load, a *Release* fence before each non-atomic store, and a *SequentiallyConsistent* fence after each non-atomic store. This approach is analogous to the memory-barrier nodes that we inserted in the Java compiler to ensure SC, as detailed in Section 3.2.2. On x86-64 hardware, the *Acquire* and *Release* fences will become no-ops, and the *SequentiallyConsistent* fence will become an `MFENCE` instruction to prevent loads from being reordered before stores.

Finally, there is one last wrinkle. According to the specification of LLVM [2], compiler optimizations do not have to respect the semantics of fences for regular (non-atomic) loads and stores. To ensure that the code generation pass nonetheless does not perform any optimizations on these memory accesses, during `SCExpand`, we also annotate all of these accesses as `volatile`, which has a similar semantics to the `volatile` keyword in C/C++ (and is notably different from `volatile` in Java). In particular, the compiler will not perform any optimizations on `volatile` memory accesses. This approach is conservative, as some optimizations that are allowed under SC will not be performed on `volatile` accesses.

An alternative approach to addressing this problem of non-atomic loads and stores is to rewrite the Julia compiler to only generate LLVM memory accesses that are able to be declared *SequentiallyConsistent*. For example, a load of a structure could instead be compiled to individually load each field. Of course, there is potentially a performance cost with this approach, and it also requires invasive changes to the Julia compiler, which our approach avoids.

4.2.3 Optimizations. Simply rewriting every load and store in the LLVM IR to have SC semantics will lead to huge slowdown. However, many loads and stores are already guaranteed to have SC semantics and can be safely ignored by our transformation pass.

If the data being loaded is immutable, then it cannot participate in a data race, so it is already guaranteed to have SC semantics. Our pass accordingly does not transform loads of immutable data. Stores to immutable data (i.e., its initialization) need not be declared *SequentiallyConsistent*, but they must use some form of atomicity to ensure that they will respect later fences and hence will not be reordered after the write of the pointer to the data. Our implementation declares these stores as *Release* atomic.[21]

If the data is only visible to the runtime, it will not affect the semantics of the language and so can be ignored by our pass. The compiler developers already have to ensure there is no data race for such data to have a correct implementation of the runtime. For example, Julia optimizes the implementation of `Union` by storing small unions inline. It uses a "type tag byte" to signal the type of the actual value stored inline. It is the compiler developers' responsibility that reading and writing this byte will not cause data races, and thus we do not need to rewrite those loads and stores.

If the data is on the stack, then it is already guaranteed to have SC semantics. The Julia compiler performs an optimization to stack-allocate objects that it can determine are either immutable or non-escaping, and we leverage this optimization to avoid unnecessary fences. Theoretically, it is possible to get a pointer to an object on the stack in Julia, but this is clearly an

[21] However, it should suffice to use a lower level of atomicity.

Table 2. A List of Tbaa Types in Julia and if They Need SC Rewriting

tbaa name	what it represents	SC rewriting?
jtbaa	Everything	Yes
jtbaa_gcframe	GC frame; exists after FinalLowerGC pass	
jtbaa_stack	stack slot	
jtbaa_data	Any user data that "pointerset/ref" are allowed to alias	Yes
jtbaa_binding	a Julia binding to a global variable	Yes
jtbaa_value	Runtime data structure of a Julia value that is not an array	Yes
jtbaa_mutab	mutable type	Yes
jtbaa_immut	immutable type	
jtbaa_ptrarraybuf	Data in an array of boxed values	Yes
jtbaa_arraybuf	Data in an array of POD	Yes
jtbaa_unionselbyte	a selector byte in isbits Union struct fields	
jtbaa_array	Runtime data structure of a Julia array (jl_array_t)	
jtbaa_arrayptr	The pointer inside a jl_array_t	
jtbaa_arraysize	A size in a jl_array_t	
jtbaa_arraylen	The len in a jl_array_t	
jtbaa_arrayflags	The flags in a jl_array_t	
jtbaa_arrayoffset	The offset in a jl_array_t	
jtbaa_arrayselbyte	a selector byte in a isbits Union jl_array_t	
jtbaa_const	Memory that is immutable by the time LLVM can see it	

"unsafe" operation, so we ignore it. This operation also does not play well with Julia's **garbage collection (GC)**.

To implement these optimizations, we leverage the ***type-based alias analysis*** **(tbaa)** metadata of memory in LLVM IR. Memory in LLVM IR does not have types, so LLVM relies on the tbaa metadata in the IR to do type-based alias analysis. Such metadata describes the type system of the higher-level language [3]. When the Julia compiler generates the LLVM IR, it will "decorate" each load and store instruction with a *MetaData* node that specifies the tbaa type of the address accessed accessed by the load or store instruction. We manually examined all tbaa types that Julia uses and identified the types that are "safe" for SC, following the above description. During our transformation pass, for each candidate load or store instruction, we check the tbaa information and ignore the ones that are accessing "safe" locations. Table 2 lists all the tbaa types used by Julia and if they need SC rewriting.

However, the tbaa information sometimes is overly conservative. Specifically, not all stack-allocated objects will be identified as such in the tbaa information. Hence, we implement an additional optimization to identify loads and stores to stack locations. This is not immediately obvious in the LLVM IR, as memory accesses to both stack and heap locations use the same instructions. Luckily, the Julia compiler uses LLVM's custom *address spaces* to identify GC-tracked pointers and their uses, and we can use this information to identify stack locations. Specifically, stack locations are allocated in address space 0, and no other locations use that address space. Therefore, our rewriting pass ignore loads and stores for addresses in address space 0.

Finally, some Julia functions are implemented directly as "builtins" in C++. While these functions are also compiled by LLVM, their loads and stores do not have Julia specific tbaa information. We manually inspected the C++ source code of builtin functions and rewrote them to ensure SC, using C++ `atomic` accesses where necessary.

Sometimes doing this rewriting required judgment calls on our part regarding the intended semantics. A good example are the `memset`, `memmov`, and `memcpy` functions. These functions do not guarantee that their memory accesses to move/copy bytes will occur in order. They also do not prevent earlier/later memory operations in the thread from being reordered with their memory operations. In our versions of these functions, we chose to still not provide any ordering guarantee among the bytes being moved/copied, as arguably callers should not expect such guarantees. However, we would like all of these accesses to be "atomic" relative to memory accesses before or after them in the program, to preserve the per-thread program order when viewing these calls as single instructions. To achieve this semantics, we rewrote all *memset*, *memmov*, or *memcpy* calls to our own versions that perform the memory operations in a loop. We inserted fences before and after the loop to prevent reorderings with code outside of the function. Further, each memory operation in the loop is declared to be *Unordered* atomic, which is the lowest level of atomicity in LLVM [1], to ensure that it will respect the fences.

4.3 Experiments

We performed several performance experiments for our SC-Julia compiler on x86-64 hardware. We ran all the evaluations on a eight-core (four physical cores with hyper-threading enabled) Intel(R) Core(TM) i7-6700 machine. We compared the performance of our implementation to that of the original v1.4.1 Julia but switched its LLVM version to v9.0.1.

4.3.1 BaseBenchmarks. The BaseBenchmarks benchmark suite is a collection of single-threaded Julia benchmarks available for CI tracking, available on GitHub[22] and used by the Julia developers to track the performance of the Julia language. There are more than 3,000 benchmarks in the benchmark suite categorized into different groups. Many of the benchmarks in BaseBenchmarks are microbenchmarks, but there are also some larger benchmarks.

The BaseBenchmarks benchmark suite uses Julia's BenchmarkTools benchmarking framework. The framework will tune each test to find the correct test parameters for each test, run the test with the tuned test parameters, and record the results. We calculate the overhead of the tests using the median execution time reported for each test. We then compute the relative execution time of SC-Julia over the baseline Julia for each test and finally calculate the geometric mean of relative execution times for the tests in each test group as well as for the whole benchmark suite.

Figure 19 shows the results. The x-axis divides the tests into different groups, and the y-axis shows the geometric mean of the relative execution time of tests in each group. The numbers in parenthesis are the numbers of tests in the test groups. Figure 19 also shows the results of two different test settings. The orange bars on the right show the results when both the baseline and our SC-Julia are running with the O2 optimization level, which is the default optimization level for the Julia compiler, while the blue bars on the left show the results when both are running with O0 optimization level for the Julia compiler.

From the figure, we can see that with the O2 optimization level, 10 out of 18 groups of tests have an average overhead of more than 100%. The average overhead across the whole benchmark suite is 76%, with the maximum overhead of any single test being 25,642%. But with the O0 optimization level, all groups have an average overhead of less than 100%. The average overhead across the whole benchmark suite is 22%, with the maximum overhead of any single test being 26,154%. While it may be consistent with people's intuition that SC would have a higher cost for a higher optimization level, we performed an experiment to better understand why, discussed next.

[22] https://github.com/JuliaCI/BaseBenchmarks.jl.

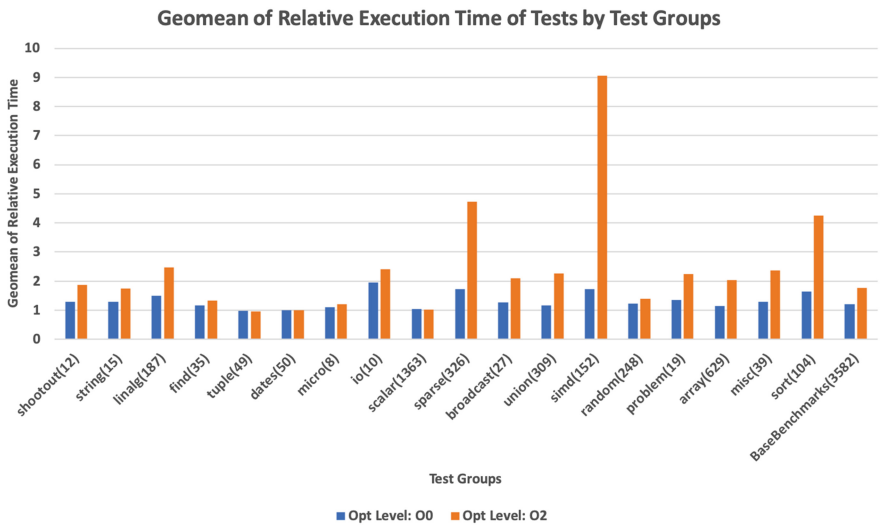

Fig. 19. Geometric mean of relative execution time of SC-Julia over the baseline Julia. Each bar represents a test group in BaseBenchmarks, the number in the parentheses after each test group name is the number of tests in that group. The last bar represents the whole benchmark suite.

4.3.2 Understanding the Performance Impact of SC. At a high level, enforcing SC has two sources of performance overhead, respectively, due to lost compiler optimizations and the need to execute additional fence instructions in the hardware. To isolate these costs, we modified LLVM such that all fence instructions in the IR will turn into no-ops during code generation. For this experiment alone, we also modified the implementation of SC-Julia so all atomic loads and stores are translated just before code generation to `volatile` accesses with associated fences (see Section 4.2.2), so SC is enforced in the hardware solely through fence instructions.

Figure 20 shows the results. With fence instructions being in the IR but not generated to hardware fences, the average cost of SC is negligible for all test groups with O0 optimization level, and the maximum overhead of any single test is also reduced to 1,633%. For O2, the cost for all test groups is also much lower than SC-Julia. The average cost with O2 is 23%, and the maximum overhead of any single test is 6,746%. Comparing with the results in Section 4.3.1, we conclude that most of the cost of enforcing SC comes from the cost of the hardware fences. Hence, the main reason for the higher relative cost of SC with O2 versus O0 is that the baseline execution of a program with O2 is much faster than the baseline execution with O0, so the relative cost of hardware fences under O2 is higher than under O0.

4.3.3 `@drf` annotations. The experiments above constitute a worst-case, and unrealistic, usage of the `volatile`-by-default memory model, as there are no `@drf` annotations whatsoever. And indeed, certain tests (mostly microbenchmarks) in the benchmark suite have a very high overhead with SC-Julia. For example, the benchmarks `["array", "bool", "boolarray_true_fill!"]` and `["array", "bool", "bitarray_true_fill!"]` test the performance of calling `fill!` to set all 1,000,000 elements in a bool or bit array to `true`, and they have overheads of 256.42× and 41.24×, respectively.

Figure 21 illustrates how `@drf` annotations can help. The majority of the overhead in these benchmarks comes from the fact that the existence of the atomic instructions/fence instructions in the IR prevents the compiler to optimize the whole loop to a `memset` call. Adding the `@drf`

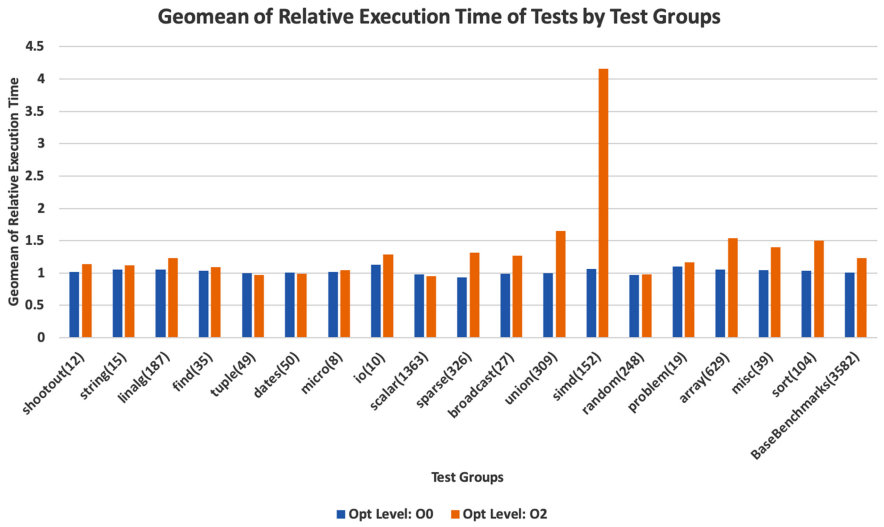

Fig. 20. Geometric mean of relative execution time of SC-Julia without hardware fences over the baseline Julia without hardware fences. Each bar represents a test group in BaseBenchmarks, the number in the parentheses after each test group name is the number of tests in that group. The last bar represents the whole benchmark suite.

```
1  @noinline function fill!(dest::Array{T}, x) where T
2      xT = convert(T, x)
3      @drf for i in eachindex(dest)
4          @inbounds dest[i] = xT
5      end
6      return dest
7  end
```

Fig. 21. Implementation of the library function `fill!` in Julia and with `@drf` annotation.

annotation to the loop body tells SC-Julia not to rewrite any load and store in the loop body, thereby allowing the `memset` optimization. Accordingly, the performance cost goes away, but in exchange now the program may not have SC semantics if there is a data race on the array elements during a call to `fill!`.

To gauge the potential effectiveness of `@drf` annotations in reducing the performance cost of SC-Julia, we added a few `@drf` annotations to the BaseBenchmarks tests. First, we treat loops that already have been annotated as `@simd` as if they are also annotated as `@drf`. The `@simd` annotation in Julia tells the compiler to use special SIMD instructions to implement the loop, implicitly promising that the loop iterations are independent and may be reordered. It is also unlikely that such an optimization would preserve behavior in the presence of data races. Hence, these loops are also naturally candidates for the `@drf` annotation. Additionally, we marked the `Base` module in Julia, which contains standard library functions such as functions and macros appropriate for performing scientific and numerical computing, as `@drf`.

Figure 22 shows the average overhead of SC-Julia over the baseline Julia on BaseBenchmarks with these `@drf` annotations. The average overhead reduces by roughly 50%. Specifically, the average overhead for BaseBenchmarks with O0 is reduced from 22% to 13%, and with O2 it is reduced from 76% to 36%. These results illustrate the potential for `@drf` annotations to significantly reduce the overhead of the `volatile`-by-default memory model.

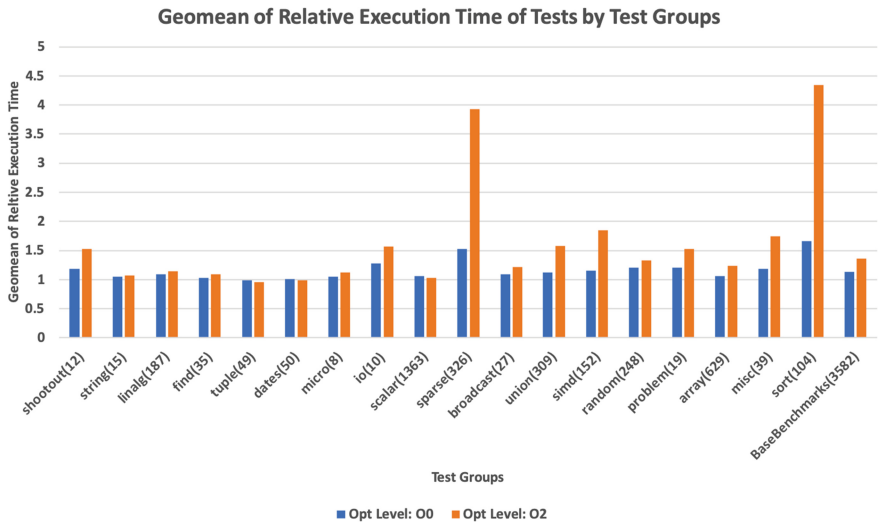

Fig. 22. Geometric mean of relative execution time of SC-Julia over the baseline Julia for BaseBenchmarks, with `@drf` behaviors for `@simd` and `Base` module. O0 and O2 represent different optimization levels.

```
1  using BenchmarkTools
2
3  arr = zeros(Int32, 100000)
4  function perf_loop(arr)
5      for i = 1:100000
6          arr[i] = i
7      end
8  end
9
10 @btime perf_loop(arr)
```

Fig. 23. A microbenchmark that uses a loop to set every element in an array.

4.3.4 Hardware Instruction Selection. Another interesting finding we discovered is that the hardware barrier instruction used greatly affects the cost of SC. On x86-64, both a sequentially consistent load and a non-SC load are mapped to a `MOV` instruction. A sequentially consistent store could be mapped to either an `XCHG` instruction or a sequence of `MOV; MFENCE` on x86-64.

In LLVM, if a store has the *SequentiallyConsistent* atomicity level, then it will be translated into an `XCHG` instruction. To measure the impact different hardware barriers have on the cost of SC, we implemented another version of SC-Julia and call it mSC-Julia. In this alternate implementation, instead of rewriting loads and stores to have the *SequentiallyConsistent* atomicity level when possible, we always rewrite loads and stores to be surrounded by fence instructions. Therefore, at the hardware level, SC-Julia will try to use `XCHG` for SC stores whenever possible, while mSC-Julia will always use the sequence of `MOV; MFENCE`.

We first tested the performance with a microbenchmark that uses a loop to set every element in an array, as shown in Figure 23. The execution time is 52.795 μs using the baseline Julia, 511.825 μs using SC-Julia, and 1,300 μs using mSC-Julia. In other words, using the `MFENCE` instruction rather than an `XCHG` instruction more than doubles the overhead of enforcing SC for a tight loop. Figure 24 shows the performance overhead of mSC-Julia over the baseline Julia for the BaseBenchmark suite.

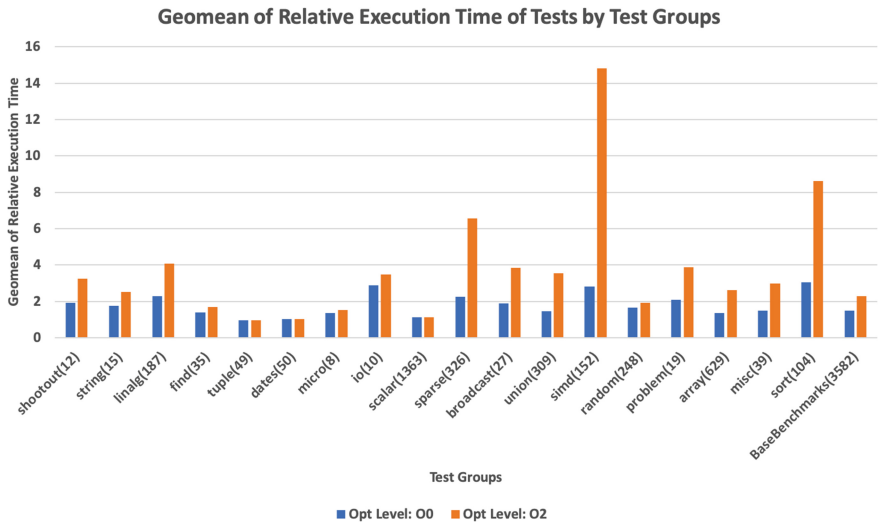

Fig. 24. Geometric mean of relative execution time of mSC-Julia over the baseline Julia. O0 and O2 represent different optimization levels.

The mSC-Julia compiler has an average overhead of 50% for O0 and 130% for O2, which again is roughly twice the overhead of SC-Julia.

This experiment illustrates the importance of hardware fence performance on the cost of SC. We also saw the importance of fences in the experiment described in Section 4.3.2. Hence, future improvements in the hardware implementation of fences have the most potential for reducing the cost of the `volatile`-by-default memory model.

5 OPTIMIZING THE IMPLEMENTATION OF `VOLATILE`-BY-DEFAULT SEMANTICS

The baseline implementation of the `volatile`-by-default semantics, described in the previous two sections, may have less overhead than is generally expected, but it is still significant, particularly for Java on the weak ARM-v8 platform and for the dynamic, array-heavy Julia language. While we showed that judicious choices of `relaxed` and `@drf` annotations can significantly reduce the overhead, the results also argue for the need to explore optimization techniques for language-level SC on stock hardware.

We take the first steps in this direction by proposing a *speculative* approach to enforcing the `volatile`-by-default semantics. 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, then any speculatively compiled code for the object is removed, and future JITed code for the object will include the necessary fences to ensure SC. We have implemented this approach as a modification to our VBDA-HotSpot JVM for ARM-v8, and we call this version S-VBD.

5.1 Design Decisions

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 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 decompilation of this method (see below). 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, then 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.

```
1  compile(m) {
2    if(m.class.mode==fast) {
3      compile fast_version(m);
4      atomic {
5        if(m.class.mode != fast)
6          abort_compilation();
7        else
8          register_compiled_method();
9      }
10   }
11   else {
12     compile slow_version(m);
13     register_compiled_method();
14   }
15 }
```

Fig. 25. Just-in-time compilation of a method.

5.2 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.

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. 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 that 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 25 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, then 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, then 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 26 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 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)

```
1  slow_version(m) {
2    vbd(m.body);
3  }
4
5  fast_version(m) {
6    if (curr_thread == this.creator_thread)
7      m.body;
8    else
9      switch_to_slow(this.class);
10 }
11
12 switch_to_slow(C) {
13   atomic {
14     if(C.mode == slow)
15       return;
16     C.mode = slow;
17     deoptimize_to_slow(C);
18   }
19 }
```

Fig. 26. The *slow* and *fast* versions of a method.

```
1  interpreter_version(m) {
2    if (this.class.mode == fast &&
3        curr_thread != this.creator_thread) {
4      switch_to_slow(this.class);
5    }
6    slow_version(m);
7  }
```

Fig. 27. The interpreted version of a method.

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 it can safely invalidate compiled methods. Also, we make the `switch_to_slow` function shown in Figure 26 atomic to prevent multiple threads from deoptimizing the same methods and to prevent the `compile` function in Figure 25 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 27.

5.3 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 (i.e., for fields that not declared `private`). For objects other than `this` S-VBD performs safety checks on a per-field-access basis. This subsection describes how such checks are implemented.

Table 3. The Implementation for `volatile` Accesses on ARM-v8 in HotSpot (First Two Rows) and an Optimized Implementation for Memory Accesses on ARM-v8 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`

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.

5.4 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-v8. The first two rows of Table 3 show the memory barriers required before and/or after a `volatile` memory access in Java, as described in the JMM Cookbook [29], and the corresponding ARM-v8 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-v8.

The baseline implementation of VBDA-HotSpot, which uses the approach of VBD-HotSpot, 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 3 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

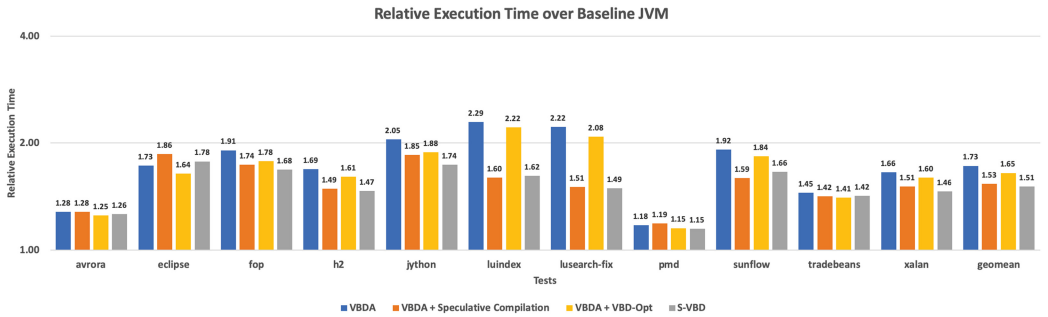

Fig. 28. Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine A, y-axis in logarithmic scale.

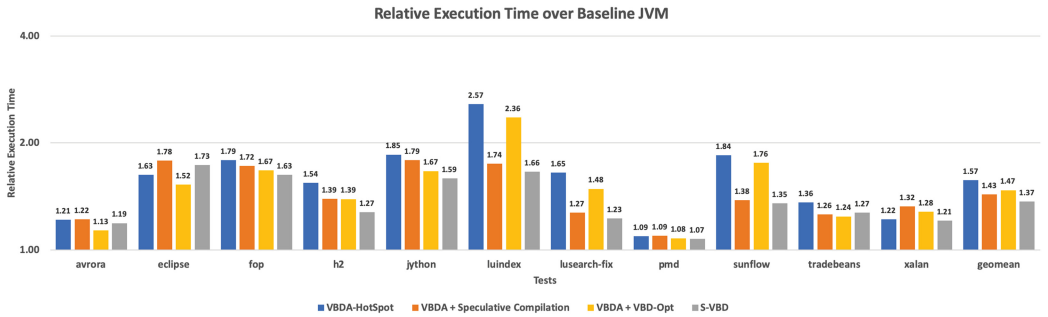

Fig. 29. Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine B, y-axis in logarithmic scale.

VBD store, the corresponding implementation of the required barriers in ARM-v8 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.

5.5 Performance Evaluation

5.5.1 DaCapo Benchmarks. We measured the peak performance of S-VBD for the DaCapo benchmarks using the same methodology as for the earlier experiments for Java. The fourth series in Figures 28 and 29 shows the overhead of our approach over the baseline HotSpot JVM on machine A and machine B. The geometric mean overhead of the S-VBD approach is, respectively, 51% and 37% for the two machines, which is a significant improvement over the geometric mean 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 28 and 29 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

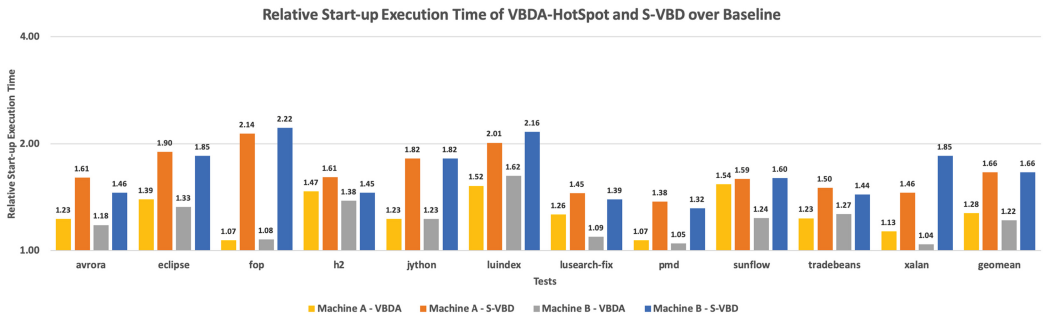

Fig. 30. Relative startup execution time of VBDA-HotSpot and S-VBD over the baseline JVM, y-axis in logarithmic scale.

more effective optimization. As the fourth series shows, together they are even 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. The first iterations of the same benchmark within a single JVM invocation are slower than later iterations, as they need to perform class loading and JIT compilation as well as deoptimization/recompilation before reaching steady-state performance. Our speculative compilation scheme introduces more deoptimizations and therefore can further slow down the initial iterations. We measured the startup performance of both VBDA-HotSpot and S-VBD using an existing methodology [25]. 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 30. 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.

5.5.2 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 31 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.

5.5.3 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 32 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

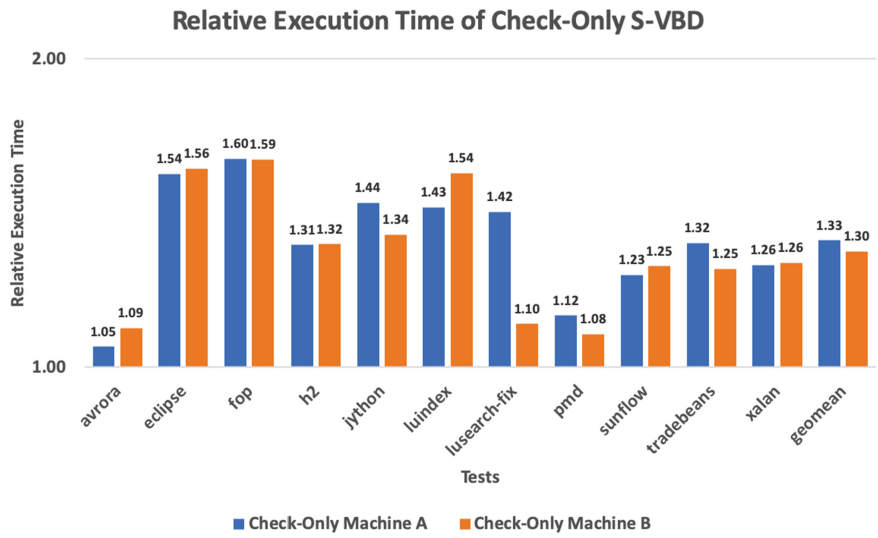

Fig. 31. Relative execution time of check-only S-VBD over baseline JVM on machine A and machine B, y-axis in logarithmic scale.

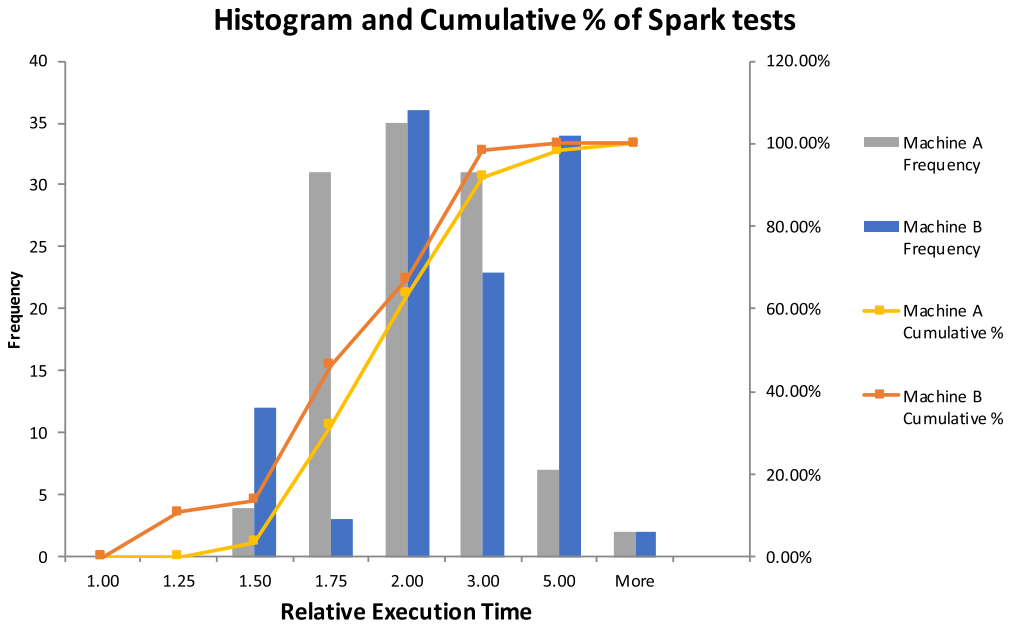

Fig. 32. Histogram and cumulative % of relative execution time of S-VBD for Spark benchmarks.

ones for VBDA-HotSpot from Figure 18, we see that our speculative compilation strategy provides little to no benefit 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.

5.5.4 Performance Evaluation Summary. Our experimental results indicate that there is significant scope for optimizing the performance overhead of the `volatile`-by-default memory model. On the DaCapo benchmarks for Java, our speculative implementation technique substantially reduces the average cost of enforcing `volatile`-by-default on the weak ARM-v8 platform, and it has an even more dramatic reduction to the maximum cost. Further, our implementation is fully compatible with modern just-in-time compilation techniques as well as features such as dynamic class loading. We are optimistic that, just as the costs of enforcing type and memory safety have reduced over time due to better optimizations, language implementers and researchers will continue to identify ways to reduce the cost of enforcing sequential consistency by default.

6 RELATED WORK

Language-level Sequential Consistency. We have proposed the `volatile`-by-default memory model for multithreading and have implemented and evaluated it for both Java and Julia. There have been other implementations of SC for Java (see below), but ours is the first in the context of a production JVM and hence the first to provide a realistic performance evaluation. Finally, ours is, to the best of our knowledge, the first comprehensive study of the cost of providing SC for any language on ARM-v8, which is a much weaker memory model than x86-64.

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 than our approach but is incompatible with modern JVMs. Shasha and Snir [60] propose a whole-program delay-set analysis for determining the barriers required to guarantee SC for a given program. Sura et al. [63] implement this technique for Java, and Kamil et al. [32] do the same for a parallel variant of Java called Titanium. These works demonstrate low performance overhead for SC on both x86-64 and POWER. Alglave et al. [7] implemented SC for C programs in a similar manner.

Other work has achieved language-level SC guarantees for Java [6, 20] and for C [45, 62] 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. [62] 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.

Vollmer et al. [64] implement the SC semantics for the Haskell programming language and demonstrate negligible overheads on x86-64. They also demonstrated low overheads for some benchmarks on ARM-v8 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.

Marino et al. [45] created an *SC-preserving* version of the LLVM compiler, meaning that the compiler preserves SC if the resulting assembly code is run on SC hardware. They did so by inspecting the various optimizations in LLVM to classify them as SC-preserving or not, and either disabling or modifying the ones that can violate SC. Their experimental results indicate that the cost of SC due to lost compiler optimizations is very low—4% on average on several C benchmark suites—and this result was independently confirmed by others subsequently [22]. Our work similarly indicates that the cost of hardware fences dominates the overhead of ensuring SC.

Finally, several works demonstrate testing techniques to identify errors in Java and C code that can cause non-SC behavior (e.g., References [24, 26]). However, such techniques are inherently incomplete and so do not provide any guarantees that a program execution is sequentially consistent.

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 [14, 57, 58, 66]. These approaches are implemented in the Jikes research virtual machine [8] and evaluated only on x86-64. Work on region serializability for C has achieved good performance either through special-purpose hardware [40, 44, 61] or by requiring $2N$ cores to execute an application with N threads [49].

Memory Model Safety. The notion of "safety" in the JMM disallows out-of-thin-air values [42], but it has proven difficult to ensure while also admitting desired optimizations [10]. Several recent works have defined new memory models that attempt to resolve this tension [18, 28, 33, 34, 48, 51]. 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 provides an empirical evaluation [48]; they demonstrate low overheads for C/C++ programs running on ARM-v8 hardware. Our work adopts and empirically evaluates a significantly stronger notion of safety for Java than these works [43], as it additionally preserves the program order of instructions and the atomicity of primitive types.

Compiler Testing. We incidentally found compilation errors in the JVM during the creation of our `volatile`-by-default version. In contrast, prior work has developed techniques to automatically identify compilation errors related to concurrency. Morisset et al. [47] create a theory of sound optimizations for the C11/C++11 memory model and use it to build a tool that finds compiler errors through a form of differential random testing. An earlier work by Eide and Regehr also used random testing, in that case to identify miscompilations of the `volatile` keyword in C [23], which was used to communicate between threads before the advent of an official memory model for C.

Weak Memory Model Performance for Java. Demange et al. [21] define an x86-like memory model for Java. They present a performance evaluation that uses the Fiji real-time virtual machine [53] to translate Java code to C, which is then compiled with a modified version of the LLVM C compiler [45] and executed on x86-64 hardware. Ritson and Owens [54] modified the HotSpot compiler's code-generation phase for both ARM-v8 and POWER to measure the cost of different instruction sequences to implement the JMM.

7 CONCLUSION

Languages like Java and Julia follow the principle of *safety by default* and *performance by choice* when it comes to type and memory safety. However, today their memory consistency models, which define the semantics of multithreading, follow the opposite approach, providing a complex and error-prone semantics that breaks fundamental programming abstractions by default, in the name of performance. Our `volatile`-by-default memory model is a conceptually simple alternative that follows the safety-by-default and performance-by-choice principle. We have provided a low-complexity implementation strategy for the `volatile`-by-default semantics and instantiated it for both Java and Julia. We have also demonstrated the potential for optimized implementation strategies to improve performance through our novel speculative technique.

Our performance results show that the cost of enforcing sequential consistency by default is perhaps less than has been previously assumed. At the same time, it is still significant. The `volatile`-by-default memory model explicitly allows programmers to trade off safety guarantees for

performance via annotations such as `relaxed` and `@drf`, in the same manner that they make these tradeoffs today for type and memory safety. Finally, our experimental results demonstrate that the lion's share of the cost of enforcing SC is the overhead of hardware synchronization instructions such as fences. Hence, future architectural improvements for fences and for special synchronized load and store instructions, as well as improvements in the compiler's usage of these instructions, can potentially have a dramatic positive impact on the cost of stronger language-level memory models.

ACKNOWLEDGMENTS

Thanks to Nobuko Yoshida and the anonymous reviewers for their constructive feedback on this article; the Works on ARM team, especially Edward Vielmetti, for setting up and providing access to an ARM server; Xiwei Ma for help implementing our litmus tests; and Jeff Bezanson and Jameson Nash for discussion and pointers to Julia benchmarks.

REFERENCES

[1] [n.d.]. LLVM Atomic Instructions and Concurrency Guide: Atomic orderings. Retrieved on June 2021 from https://llvm.org/docs/Atomics.html#atomic-orderings.

[2] [n.d.]. LLVM Language Reference Manual. Retrieved on June 2021 from https://releases.llvm.org/3.3/docs/LangRef.html.

[3] [n.d.]. "tbaa" Metadata. Retrieved on June 2021 from https://llvm.org/docs/LangRef.html#tbaa-metadata.

[4] Sarita V. Adve and H.-J. Boehm. 2010. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (Aug. 2010), 90–101. DOI: https://doi.org/10.1145/1787234.1787255

[5] S. V. Adve and M. D. Hill. 1990. Weak ordering-A new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*. ACM, 2–14.

[6] 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 *Proceedings of the 42nd International Symposium on Microarchitecture*.

[7] 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 *Proceedings of the 26th International Conference on Computer-aided Verification*. 508–524.

[8] 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 Syst. J.* 44, 2 (2005), 399–418.

[9] ARMv8 2018. ARM Cortex-A Series Programmer's Guide for ARMv8-A Version: 1.0, Section 13.2.1. Retrieved on June 2021 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CHDCJBGA.html.

[10] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *Programming Languages and Systems24th European Symposium on Programming (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 283–307.

[11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. *SIGPLAN Not.* 46, 1 (Jan. 2011), 55–66. DOI: https://doi.org/10.1145/1925844.1926394

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Rev.* 59, 1 (2017), 65–98. DOI: https://doi.org/10.1137/141000671

[13] Jeff Bezanson, Jameson Nash, and Kiran Pamnany. [n.d.]. Announcing Composable Multi-threaded Parallelism in Julia. Retrieved on June 2021 from https://julialang.org/blog/2019/07/multithreading/.

[14] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, 241–259.

[15] 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 *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, 169–190.

[16] 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.
[17] Hans-J. Boehm. 2012. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES'12)*. ACM, 9–14.
[18] 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.
[19] 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.
[20] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture*. 278–289.
[21] 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. ACM, New York, NY, 329–342.
[22] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. 2013. WeeFence: Toward making fences free in TSO. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*, Avi Mendelson (Ed.). ACM, 213–224. Retrieved on June 2021 from http://dl.acm.org/citation.cfm?id=2485922.
[23] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the International Conference on Embedded Software (EMSOFT'08)*, Luca de Alfaro and Jens Palsberg (Eds.). ACM, 255–264.
[24] 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.
[25] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. ACM, 57–76.
[26] Mohammad Majharul Islam and Abdullah Muzahid. 2016. Detecting, exposing, and classifying sequential consistency violations. In *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE'16)*. IEEE Computer Society, 241–252. DOI: https://doi.org/10.1109/ISSRE.2016.48
[27] Java Virtual Machine Specification 2017. Retrieved on June 2021 from https://docs.oracle.com/javase/specs/jvms/se8/html.
[28] Alan Jeffrey and James Riely. 2016. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st ACM/IEEE Symposium on Logic in Computer Science (LICS'16)*. ACM, New York, NY, 759–767. DOI: https://doi.org/10.1145/2933575.2934536
[29] JSR133 2018. JSR-133 Cookbook for Compiler Writers. Retrieved on June 2021 from http://g.oswego.edu/dl/jmm/cookbook.html.
[30] JuliaLang. 2020. The Julia Language. Retrieved on June 2021 from https://github.com/JuliaLang/julia/commits/v1.4.1.
[31] 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 *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP'17) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. 17:1–17:29.
[32] A. Kamil, J. Su, and K. Yelick. 2005. Making sequential consistency practical in Titanium. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society.
[33] 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'17)*. ACM, 175–189.
[34] 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'17)*. ACM, New York, NY, 618–632. DOI: https://doi.org/10.1145/3062341.3062352
[35] L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 100, 28 (1979), 690–691.
[36] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society.
[37] Lun Liu. 2020. *Safe and Efficient Concurrency for Modern Programming Languages*. Ph.D. Dissertation, University of California, Los Angeles.
[38] Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A volatile-by-default JVM for server applications. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017).

[39] Lun Liu, Todd D. 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)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 16–30.

[40] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict exceptions: Providing simple parallel language semantics with precise hardware exceptions. In *Proceedings of the 37th International Symposium on Computer Architecture*.

[41] 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 *Proceedings of the 24th International Conference on Computer-aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 495–512.

[42] J. Manson, W. Pugh, and S. Adve. 2005. The Java memory model. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 378–391.

[43] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. 2015. The silently shifting semicolon. In *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL'15) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. 177–189.

[44] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFx: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 351–362.

[45] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[46] 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).

[47] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 187–196.

[48] Peizhao Ou and Brian Demsky. 2018. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). DOI: https://doi.org/10.1145/3276506

[49] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. ...And region serializability for all. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar'13)*, Emery D. Berger and Kim M. Hazelwood (Eds.). USENIX Association.

[50] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09) (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 391–407.

[51] 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, NY, 622–633. DOI: https://doi.org/10.1145/2837614.2837616

[52] Benjamin Pierce. 2002. *Types and Programming Languages*. The MIT Press. Retrieved on June 2021 from http://www.cis.upenn.edu/~bcpierce/tapl/index.html.

[53] 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.

[54] 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)*.

[55] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. *SIGPLAN Not.* 46, 6 (June 2011), 175–186. DOI: https://doi.org/10.1145/1993316.1993520

[56] 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.

[57] 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 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 561–575.

[58] 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'15)*, Ryan Stansifer and Andreas Krall (Eds.). ACM, 65–75.

[59] Jaroslav Sevcík and David Aspinall. 2008. On validity of program transformations in the Java memory model. In *Proceedings of the 31st European Conference on Object-oriented Programming (ECOOP'08)*. 27–51.

[60] D. Shasha and M. Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Prog. Lang. Syst.* 10, 2 (1988), 282–312.

[61] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, 53–66.

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

[63] 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 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2–13.

[64] 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.

[65] 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.

[66] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. 2017. Avoiding consistency exceptions under strong memory models. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'17)*. ACM, 115–127.

Received September 2020; revised March 2021; accepted April 2021