UNIVERSITY OF CALIFORNIA

Los Angeles

Safe and Efficient Concurrency
for Modern Programming Languages

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Lun Liu

2020

ABSTRACT OF THE DISSERTATION

Safe and Efficient Concurrency
for Modern Programming Languages

by

Lun Liu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Todd D. Millstein, Chair

Safe languages provide programming abstractions, like type and memory safety, to improve programmer productivity. Unfortunately, in the domain of concurrent programming, mainstream safe languages instead choose to adopt complex and error-prone semantics by default in the name of performance. A *memory consistency model* defines the possible values that a shared-memory read may return in a multithreaded programming language. Languages today like Java and Julia have adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The downside, however, is that programmers are exposed to a complex and unintuitive semantics and must carefully declare certain variables as volatile in order to enforce program orderings that are necessary for proper behavior. Not only does this *performance-by-default* approach conflict with the design philosophy of "safe" languages, but also surprisingly, very little is actually known about the practical performance cost of a stronger memory model.

In this thesis, I define the safe-by-default and performant-by-choice `volatile`-by-default semantics for Java and other JVM-based languages. `volatile`-by-default provides sequential consistency (SC) by default, while still allows expert programmers to avoid fence overheads on performance-critical libraries. I present VBD-HotSpot and VBDA-HotSpot, modifications of Oracle's HotSpot JVM that enforce the `volatile`-by-default semantics on Intel X86 hardware and ARM hardware. I then present a novel *speculative* technique to optimize language-level SC. Finally, I have implemented an SC enforcing Julia compiler that guarantees SC for Julia by default. Through evaluations of the above projects, I show that it is possible to measure the practical cost of strong and simple semantics for concurrent programming, and with optimizations, it is possible to have a simpler semantics for concurrent programming at a reasonable cost for certain languages.

The dissertation of Lun Liu is approved.

Harry Guoqing Xu

George Varghese

Jens Palsberg

Madan Musuvathi

Todd D. Millstein, Committee Chair

University of California, Los Angeles

2020

To my mother, for showing me the strength of a woman,

and supporting me every step of the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisor, Todd Millstein, for his continuous support and guidance during my PhD. His passion and excitement during our discussions help shape the way I think about research, while his insights and encouragement help me break through obstacles in my research. He is caring, understanding, and always helpful. I have been, and will be keeping telling people how great an advisor he is.

I would like to offer my special thanks to Madan Musuvathi, who is also a great "advisor" to me. He provides so much inspiration and great ideas for each one of my PhD research project, and gives me heartfelt advice on my career plans. I can't be more grateful to the help and opportunities he gives me.

I would like to thank the rest of my committee members, Professor Jens Palsberg, Professor Harry Xu, Professor George Varghese, for serving on my committee. Their feedback has been a great help in completing my dissertation.

I would also like to thank my colleagues in the lab for the discussions, on research and on everything. It's just unfortunate that we still don't have an official name for the lab yet.

It's never possible to have a PhD without the support of family and friends. I owe a very important debt to my parent, who are the reason I stand where I am. None of it would have happened without them. Special thanks also to Jingwen, who is not only a best friend, but someone who is always supportive and believes in me, regardless of my grumpiness or moodiness when I am in "deadline mode". Thank you, too, to those who helped me settled in LA, and my friends who have been talking to me through ups and downs in this journey.

For my work in Chapter 3, I thank Professor Tyson Condie and his student Joseph Noor for permission and help to use their server as well as Ari Fogel and Saswat Padhi for permission to use their machines for performance evaluations of VBD-HotSpot. I would also like to express my gratitude to WorksOnArm and Ed Vielmetti for providing and setting up ARM server for performance evaluations of VBDA-HotSpot.

For my work in Chapter 4, I still need to thank WorksOnArm and Ed Vielmetti for

providing and setting up ARM server for the performance evaluations.

For my work in Chapter 5, I am grateful for the discussion and feedback given by Jameson Nash and Jeff Bezanson from Julia.

VITA

2015	B.Eng in Software Engineering Shanghai Jiao Tone University Shanghai, China
2015-2020	Graduate Student Researcher Computer Science Department University of California, Los Angeles
2017	Research Intern Microsoft Research Redmond, Washington
2017	Teaching Assistant CS 97: Principles and Practices of Computing Computer Science Department University of California, Los Angeles
2018	Software Engineer Intern Uber San Francisco, California
2019	Software Engineer Intern Facebook Menlo Park, California

PUBLICATIONS

Lun Liu, Todd Millstein, and Madan Musuvathi, *Accelerating Sequential Consistency for Java with Speculative Compilation*, PLDI 2019. Jun 2019.

Lun Liu, Leland Takamine, and Adam Welc, *Profiling Android Applications with Nanoscope*, VMIL 2018. Nov 2018.

Lun Liu, Todd Millstein, and Madan Musuvathi, *A Volatile-by-Default JVM for Server Applications*, OOPSLA 2017. Oct 2017.

CHAPTER 1

Introduction

Safe languages like Java and Julia provide programming abstractions, like type and memory safety, to improve programmer productivity. While some unsafe languages also provide such abstractions, safe languages *protect* these abstractions, ensuring that an unintentional programmer error cannot violate them. Doing so usually requires safe languages to pay the associated runtime costs, such as compiler-inserted bounds-checks, by default. Performance-minded programmers still have the flexibility to selectively use *unsafe* code fragments where the programmer, not the language, is responsible for ensuring safety.

Unfortunately, in the domain of concurrent programming, mainstream safe languages instead choose to adopt complex and error-prone semantics by default in the name of performance, breaking fundamental programming abstractions and failing to provide baseline guarantees for programmers. In this thesis, I focus on the problems with *memory consistency models*.

1.1 Problems with Memory Models of Today's Languages

A *memory consistency model* (or simply *memory model*) of a programming language defines the possible values that a shared-memory read may return in that language. One of the most intuitive strong memory models is *sequential consistency* (SC) [Lam79]. It guarantees that all memory reads and writes agree on a global order, and the global order is consistent with the per-thread program order. With SC, programmers can always view the execution of their program as a sequential interleaving of the instructions from each thread.

Most mainstream languages today either have a memory model weaker than SC by default, or don't even have a memory model. For example, Java [MPA05] only guarantee SC for certain "well- annotated" programs, and Julia does not have a well-defined memory model. The reasoning behind such decisions usually points to *performance.* Achieving SC requires the compiler to disable some common optimizations, and on current hardware platforms it further requires the compiler to restrict hardware optimizations by emitting expensive fence instructions on shared-memory accesses.

The downside of having a weak memory model, however, is that programmers must carefully annotate/write their programs in order to enforce the per-thread program orderings that are required for proper behaviors, otherwise programs are exposed to the relaxed/undefined semantics of the memory models, which is complex, unintuitive, and can violate critical program invariants. For example, under the current Java Memory Model (JMM) [MPA05] an object can be accessed before its constructor has completed, accesses to `long`s and `double`s are not guaranteed to be atomic, and some programming idioms, such as double-checked locking [SH97, BBB17], are not guaranteed to behave correctly. This possibility is not just theoretical: errors caused by Java's weak memory model can be found across a range of widely used Java applications.

This *performance-by-default* approach to memory models is acceptable for and consistent with the design philosophy of languages like C and C++ [BA08]. However, I argue that such an approach conflicts with the design philosophy of "safe" languages like Java and Julia. Indeed, when it comes to type and memory safety, they instead employ a *safe-by-default* and *performance-by-choice* approach: in Java, for example, type and memory safety are provided by default, but there is a library of type-unsafe operations meant for use by expert programmers in performance-critical code [MPM15]; Julia uses bounds checking to ensure memory safety when access arrays, but `@inbounds` macro can be used to elide such checks in a tight loop.

Not only does the *performance-by-default* approach conflict with the design philosophy of "safe" languages, but also surprisingly, very little is actually known about the performance cost of a stronger memory model. Surely, there have been studies regarding such costs, but they are in unrealistic settings, like whole-program compilation [SS88, SFW05, KSY05]. Nobody knows the cost of a practical implementation of SC for a modern language.

Considering the lack of measurement of the cost of a safer semantic, the decision to choose performance over programmability needs to be revisited. Essentially, choosing a memory model for a language involves an inherent performance-programmability trade-off, and such decision shouldn't be made without a better sense of what the cost of a stronger memory model like SC really is on *modern* hardwares for *modern* programming languages.

1.2 Thesis Statement

My thesis states the following:

Thesis statement It is possible to measure the cost of strong and simple semantics for concurrent programming on *modern* hardwares for *modern* languages with *modern* language implementation techniques, and with optimizations, it is possible to have a simpler semantics for concurrent programming at a reasonable cost for certain languages.

I support the above statement by 1) implementing and evaluating the cost of SC for two different modern programming languages, and 2) developing language VM and compiler techniques to reduce such cost.

I propose a simpler and stronger memory model called `volatile`-by-default for Java. `volatile`-by-default provides SC for all programs by default, while at the same time, expert programmers retain the freedom to build performance-critical libraries that violate the SC semantics. I have implemented the `volatile`-by-default semantic on Oracle's HotSpot JVM and demonstrate through comprehensive empirical evaluation that the `volatile`-by-default semantics is arguably acceptable for a predominant use case for Java today — server-side applications running on Intel X86 architectures. I have also shown that the overhead of `volatile`-by-default semantics is much less than people have expected for applications running on a weaker architecture like ARM.

I also present a *speculative* approach to implement `volatile`-by-default which further reduces the average and maximum overhead of `volatile`-by-default on ARM. Additionally, I have devised an orthogonal optimization that reduces the number of fences required to enforce the `volatile`-by-default semantics for ARM. Those two optimizations reduce the overhead of enforcing the `volatile`-by-default semantics by roughly 1/3 combined.

Besides Java, I have explored ways to provide SC for Julia. I have implemented an SC enforcing Julia compiler that guarantees SC and measured the cost of SC for Julia. I have further analyzed different factors that might affect the cost of SC and explored possibilities of using annotations to reduce the cost of SC.

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 provides some background information related to the projects. Chapter 3 focuses on `volatile`-by-default semantics that provides SC for Java by default. It also covers how I implement it for HotSpot JVM and the measurement of the performance of `volatile`-by-default on both X86 and ARM. Chapter 4 describes the optimizations including a speculative compilation optimization I propose and develop to reduce the cost of `volatile`-by-default on ARM and its evaluations. Chapter 5 talks about my work to provide an SC enforcing Julia compiler and interesting finds from performance evaluations. Chapter 6 discusses about related work. Chapter 7 concludes.

CHAPTER 2

Background

In this chapter, I will give an overview on memory models and its relation with compiler/hardware optimizations, discuss in details about the memory models of Java and Julia, and provide some information on the HotSpot JVM and the Julia compiler.

2.1 Memory Models and Compiler/Hardware Optimizations

A *memory consistency model* (or simply *memory model*) of a programming language defines the possible values that a shared-memory read may return in that language. Different kinds of memory models allow different kinds of reorderings of memory access. Stronger memory models limit more memory reorderings, and thus provide stronger guarantees of the program execution to the programs. Weaker (or relaxed) memory models allow more memory access reorderings to admit more optimizations, but in turn expose the programmers to a more complex and unintuitive semantics.

One of the most intuitive strong memory models is *sequential consistency* (SC) [Lam79]. It guarantees that all memory reads and writes agree on a global order, and the global order is consistent with the per-thread program order. With SC, programmers can always view the execution of their program as a sequential interleaving of the instructions from each thread.

Compiler optimizations that may reorder memory accesses such as *common subexpression elimination (CSE)*, *code motion*, and *dead store elimination* may violate SC. Hardware optimizations like *write buffers* may also produce program executions that does not follow SC. In order to achieve SC, both the hardware and the compiler will need to disable such optimizations.

2.1.1 The Java Memory Model

The Java memory model was defined more than a decade ago [MPA05] 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" [MPA05]. 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 [AH90], whereby SC is only guaranteed for *data-race-free* programs. A *memory race* occurs when two threads concurrently access the same memory location and at least one of those accesses is a write. A program is considered to be data-race-free if all instance variables that participate in a memory race are declared `volatile`.[1] The JMM guarantees SC semantics for `volatile` variables, which requires implementations to disable many compiler optimizations and to emit fence instructions that prevent the hardware from violating SC through out-of-order execution.

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

program safety The JMM strives to ensure safety for all programs, even ones with data races. The JMM's notion of program safety is centered around the idea of preventing "out-of-thin-air reads" [MPA05]. In the presence of data races, some compiler optimizations can in principle lead to a situation whereby multiple program transformations justify one another in a cycle. Such transformations can introduce values in the program that would never otherwise occur, which creates a potentially serious security concern. The

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

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 [MPA05].[2]

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" [MPA05]. 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 `double`s and `long`s 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."[3]

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

[2] The JMM's causality is known to disallow some optimizations that it was intended to allow, notably common subexpression elimination [CKS07, SA08]. 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.

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

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.[4]

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 [SH97] and Dekker's mutual exclusion algorithm.

2.1.2 Missing-annotation bugs

The misbehaviors above are instances of what I 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 my 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`. I report the first twenty 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.[5] These errors occur in popular systems such as the Cassandra database, the HDFS distributed file system,

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

[5] 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 twenty issues listed here are returned by a search in the issues tracker as of 05/12/2020.

and the Spark system for big-data processing[6] and can thereby impact the applications that employ these systems.

Missing-annotation bugs contrast with data races that violate program invariants due to the programmer's failure to implement the necessary synchronization protocol. Examples of the latter errors include atomicity violations that arise from failing to hold a lock or holding the wrong lock, and ordering violations that arise when threads fail to signal one another properly. Yet as Boehm et al. [Boe11] points out, missing-annotation bugs are far from "benign" but rather can cause surprising and harmful behaviors. For instance, exposing a partially constructed object, as shown above, can have serious security implications [TSM17].

2.1.3 The Julia Memory Model

The Julia programming language is a new and emerging dynamic language aims to provide the flexibility of a dynamic language, while retaining good performance comparable to traditional static languages. It is open-sourced and is under active development.

Recently, Julia announced the addition of composable multi-threaded parallelism in the language [BNP]. It does not, however, provides a memory model, meaning that its memory model is implicit in the optimizations that the compiler and hardware perform. As a result, the actual behavior of its memory model is similar to that of the LLVM, which is inspired by C++0x memory model and is weaker than SC.

As a result, many surprising behaviors and violations of critical program invariants described above that can happen in Java also exist in Julia. For example, Figure 2.1 shows an implementation of the Dekker's mutual exclusion algorithm in Julia that is not guaranteed to work as expected.

Assume we have two threads both want to enter a critical section. Before they enter the critical section, they will call `setAndRead()` to signal their intent to enter the critical section by setting the corresponding element in `lockArr`, and check if the other thread has done so by checking the other element in `lockArr`. Under SC, the algorithm can work correctly, i.e.,

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

```julia
import Base.Threads.@spawn
import Printf.@printf

lockArr = Int32[0; 0]
waitArr = Int32[-1; -1]

function setAndRead(threadNo)
    if threadNo == 1
        lockArr[1] = 1
        waitArr[1] = lockArr[2]
    else
        lockArr[2] = 1
        waitArr[2] = lockArr[1]
    end
end

function test()
    global lockArr = [0;0]
    global waitArr = [-1;-1]
    r1 = @spawn setAndRead(1)
    r2 = @spawn setAndRead(2)
    wait(r1)
    wait(r2)

    if waitArr[1] == 0 && waitArr[2] == 0
        # Both threads enter critical section.
        println("SC Violation!")
        return false
    end
end
```

Figure 2.1: An implementation of Dekker's mutual exclusion algorithm in Julia which is not guaranteed to work correctly

we will never see both elements in `waitArr` to be 0. However, it is possible with the current Julia runtime that `test()` returns 0, which means it would allow both threads to enter the critical section.

2.2 The HotSpot Java Virtual Machine

The Oracle's HotSpot JVM is an implementation of the Java Virtual Machine Specification [Jav17]. It is widely used and part of the OpenJDK — the official reference implementation of Java SE. [7]

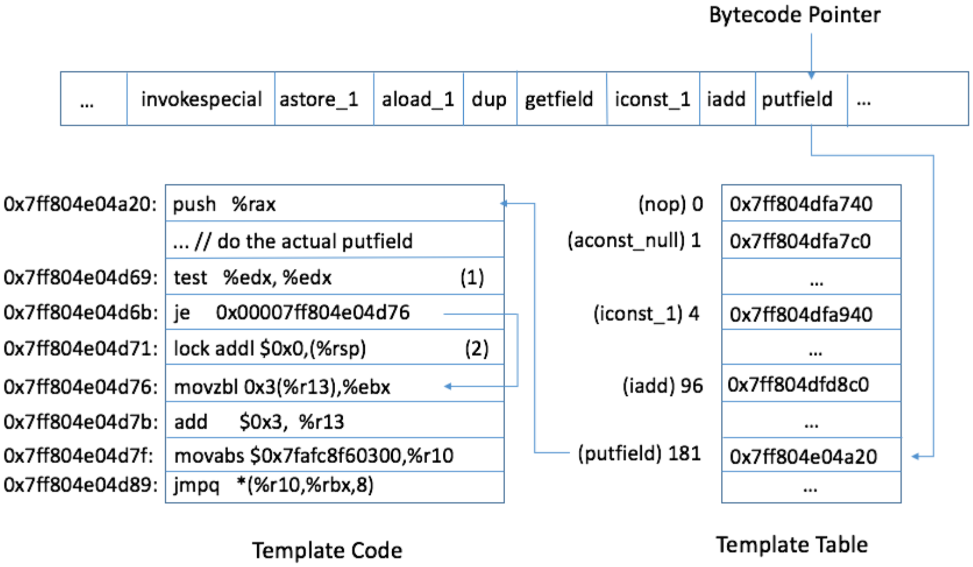
To execute Java bytecode instructions, HotSpot employs just-in-time (JIT) compilation. In this style, bytecodes are first executed in interpreter mode, with minimal optimizations. During execution, HotSpot identifies parts of the code that are frequently executed ("hot spots") and compiles them to optimized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

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

2.2.1 The HotSpot JVM Interpreter

The HotSpot JVM uses a *template-based interpreter* for performance reasons. In this style a `TemplateTable` maps each bytecode instruction to a *template*, which is a set of assembly instructions (and hence platform-specific). The `TemplateTable` is used at JVM startup time to create an interpreter in memory, whereby each bytecode is simply an index into the `TemplateTable`.

[7] My work is based on OpenJDK 8u. All specific descriptions of any technical details in this thesis is based on this version.



```
0x7ff804e04a20: push   %rax
                ... // do the actual putfield
0x7ff804e04d69: test   %edx, %edx                (1)
0x7ff804e04d6b: je     0x00007ff804e04d76
0x7ff804e04d71: lock addl $0x0,(%rsp)            (2)
0x7ff804e04d76: movzbl 0x3(%r13),%ebx
0x7ff804e04d7b: add    $0x3,  %r13
0x7ff804e04d7f: movabs $0x7fafc8f60300,%r10
0x7ff804e04d89: jmpq   *(%r10,%rbx,8)
```

Figure 2.2: Interpretation example of bytecode `putfield`

Figure 2.2 illustrates how the template-based interpreter works. The bytecode pointer (BCP) is currently pointing at the bytecode `putfield` (181). The interpreter uses this bytecode as an index in the `TemplateTable` (right side of the figure) to find the address of the corresponding template. The interpreter then jumps to this address to begin executing the template (left side of the figure). After writing to the field, the last four lines of the template show how the interpreter reads the next BCP index, increments the BCP, and jumps to the next code section. In this example, we add 3 to BCP (`%r13`) to point to the next bytecode, because the length of the `putfield` instruction is 3 bytes: a 1-byte opcode and a 2-byte index representing the field.

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, I 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 has its runtime with JIT compilation. After the code has been loaded, the Julia runtime will use simple heuristics to decide if the code needs to be interpreted by its interpreter or be compiled by its compiler.

The Julia compiler is built using the LLVM toolchain [LA04]. LLVM is a compiler infrastructure that can provide compiler writers easy access to a collection of industrial strength compiler techniques. The Julia compiler can function as an LLVM front-end: if the Julia code needs to be compiled, the compiler will first generate a Julia level IR and perform some simple optimizations. After that, the Julia compiler will transform the Julia level IR into an LLVM IR.

The LLVM execution engine will execute a sequence of different LLVM *passes* to analyze, optimize, and generate the native instructions to execute. The subset of passes and the ordering of how they are executed are specified by the Julia compiler. Julia also implements their own LLVM passes and include them in the pass pipeline.

CHAPTER 3

volatile-by-default JVMs for Server Applications

3.1 Introduction

A *memory consistency model* (or simply *memory model*) defines the possible values that a shared-memory read may return and thus plays a central role in the semantics of a multithreaded programming language. Choosing a memory model for a language involves an inherent performance-programmability trade-off. *sequential consistency* (SC) [Lam79] provides an intuitive programming model by ensuring that a program's instructions (appear to) execute in a global total order consistent with the per-thread program order. But achieving SC requires the compiler to disable some common optimizations, and on current hardware platforms it further requires the compiler to restrict hardware optimizations by emitting expensive fence instructions on shared-memory accesses.

To avoid this cost, the Java language [MPA05] has adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The Java Memory Model (JMM) is based on the DRF0 style [AH90], whereby SC is only guaranteed for *data-race-free* programs. A program is considered to be data-race-free if all instance variables that participate in a memory race are declared `volatile`.[1] The JMM guarantees SC semantics for `volatile` variables, which requires implementations to disable many compiler optimizations and to emit fence instructions that prevent the hardware from violating SC through out-of-order execution.

The downside, however, is that programmers must carefully declare certain variables as

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

`volatile` in order to enforce the per-thread program orderings that are required for proper behavior. If a programmer does not annotate the necessary variables as `volatile`, programs are exposed to the relaxed semantics of the JMM, which is complex, unintuitive, and can violate critical program invariants. For example, under the JMM an object can be accessed before its constructor has completed, accesses to `long`s and `double`s are not guaranteed to be atomic, and some programming idioms, such as double-checked locking [SH97, BBB17], are not guaranteed to behave correctly. This possibility is not just theoretical: errors due to the failure to annotate certain variables as `volatile`, which I term *missing-annotation bugs*, can be found across a range of widely used Java applications.

This *performance-by-default* approach to memory models is acceptable for and consistent with the design philosophy of languages like C and C++ [BA08]. However, I argue that such an approach conflicts with the design philosophy of "safe" languages like Java. Indeed, when it comes to type and memory safety, Java instead employs a *safe-by-default* and *performance-by-choice* approach: type and memory safety are provided by default, but there is a library of type-unsafe operations meant for use by expert programmers in performance-critical code [MPM15].

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

volatile-by-default JVMs for X86 and ARM At the outset, it is unclear if the volatile-by-default semantics is practical for Java, given the cost of memory fences on today's hardware platforms. One should deem the volatile-by-default approach as unacceptable if the only way to make programs reasonably efficient is to declare large portions as relaxed. In fact, current conventional wisdom about the cost of these fences and the associated cost of providing SC to the programmer strongly points against the volatile-by-default approach. For instance, Kaiser et al. (2017) say that SC is "woefully unrealistic" due to "the significant performance costs ... on modern multi-core architectures" [KDD17] and Demange et al. (2013) say that "SC would likely cripple performance of Java programs on all modern microprocessors." [DLZ13] I first demonstrate, through comprehensive empirical evaluation, that the volatile-by-default semantics is in fact acceptable for a predominant use case for Java today — server-side Java applications running on Intel X86 architectures. I then show that on another widely used hardware platform today, ARM, where both reads and writes can be reordered by hardware [FGP16], volatile-by-default semantics also has a much lower overhead than people have expected.

The straightforward way to implement the volatile-by-default semantics is through a source-to-source translation that adds the appropriate volatile annotations. The advantage of this approach is that it is *independent* of the JVM, allowing us to evaluate the cost of volatile-by-default semantics on various JVMs and hardware architectures. Unfortunately, neither Java nor the Java bytecode language provides a mechanism to declare array elements as volatile. Thus, such an approach fails to provide the desired semantics. I 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 I settled on changing an existing JVM implementation, I considered doing so in a research virtual machine [AAB05]. But it was not clear how the empirical observations from such a JVM would translate to a production JVM implementation.

Therefore I implemented the volatile-by-default semantics by directly modifying Oracle's HotSpot JVM which is part of the OpenJDK version 8u [Ope17]. I call my modified version for X86 and ARM VBD-HotSpot and VBDA-HotSpot respectively. To the best of my knowledge, they are the first implementations of SC for a production Java virtual machine (JVM) that includes the state-of-the-art implementation technology, such as dynamic class loading and just-in-time (JIT) compilation. This in turn enables me to provide the first credible experimental comparison between SC and the JMM. This is also the first comprehensive study of the cost of providing SC for any language on ARM.

I implemented VBD-HotSpot and VBDA-HotSpot by reusing mechanisms already in place for handling `volatile` variables in the interpreter and compiler. This provides me two advantages. First and foremost, the correctness of VBD-HotSpot mostly follows from the correctness of the HotSpot JVM's implementation of `volatile` semantics. Second, I automatically obtain the benefits of optimizations that HotSpot already employs to reduce the overhead of `volatile` accesses. VBD-HotSpot is open-source and available for download at `https://github.com/SC-HotSpot/VBD-HotSpot`. VBDA-HotSpot is also open-sourced and can be found at `https://github.com/Lun-Liu/schotspot-aarch64`.

Results For the DaCapo benchmarks [BGH06] on a modern X86 server, with no `relaxed` annotations, the overhead of VBD-HotSpot versus the unmodified HotSpot JVM is 28% on average, with a maximum of 81%. Given that VBD-HotSpot potentially inserts a fence on *every* heap store, I believe that this overhead is less than commonly assumed. My experiments show that the benchmarks that incur the highest overheads are either single-threaded or mostly-single-threaded. Excluding these benchmarks reduces the average overhead to 21%.

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

Another common assumption is that the cost of SC increases with the number of processor cores and sockets. For instance, a fence on x86 stalls the processor until it has received read invalidations from all processors that have a shared copy of addresses in its pending store buffer. One expects these stalls to be longer when there are more threads in the program, and when it is running across a larger number cores and sockets. However, my evaluation shows that this assumption is not true, at least in my experimental setting. The overhead of SC *improves* with increased concurrency and with the number of sockets; apparently the increased cost of fences is compensated by the increased cost of regular memory accesses.

3.2 `volatile`-by-default Semantics for Java

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

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

Precisely defining the SC semantics requires one to specify the granularity of thread interleaving, which has been identified as a weakness of the SC memory model [AB10]. 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) `double`s and `long`s, 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. I argue that the `volatile`-by-default semantics is a natural *baseline* guarantee that a "safe" language should provide for all programs. The `volatile`-by-default memory model clearly satisfies the JMM's desired programmability and safety goals. In terms of programmability, `volatile`-by-default is strictly stronger than the JMM, so all program guarantees provided by the JMM are also provided by `volatile`-by-default. In terms of safety, the `volatile`-by-default semantics prevents all cyclic dependencies and hence rules out the particular class of such cycles that can cause out-of-thin-air reads. Moreover, `volatile`-by-default eliminates all missing-annotation bugs.

Further, the `volatile`-by-default semantics provides a more general notion of safety by protecting several fundamental program abstractions [MMM15]. First, all primitives are accessed atomically. Second, sequential reasoning is valid for all programs. This ensures, for example, that an object cannot be accessed until it is fully constructed (unless the program explicitly leaks `this` during construction), and more generally that program invariants that rely on program order are guaranteed regardless of whether the program has data races.

Finally, I 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.

3.3 `volatile`-by-default JVMs for X86 and ARM

The straightforward way to implement the `volatile`-by-default semantics is through a source-to-source translation that adds the appropriate `volatile` annotations. The advantage of this approach is that it is *independent* of the JVM, allowing us to evaluate the cost of `volatile`-by-default semantics on various JVMs and hardware architectures. Unfortunately, neither Java nor the Java bytecode language provides a mechanism to declare array elements as `volatile`. Thus, such an approach fails to provide the desired semantics. I 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 I settled on changing an existing JVM implementation, I considered doing so in a research virtual machine [AAB05]. But it was not clear how the empirical observations from such a JVM would translate to a production JVM implementation.

Therefore, I opted to instead implement the `volatile`-by-default semantics through a direct modification to the Java virtual machine, which executes Java bytecode instructions. I chose to modify Oracle's HotSpot JVM, which is widely used and part of the OpenJDK — the official reference implementation of Java SE. In particular I modified the version of HotSpot that is part of the OpenJDK 8u for both X86 and ARM. The modified version for X86, called VBD-HotSpot, adds a flag `-XX:+VBD` that allows users to obtain `volatile`-by-default semantics. The ARM version, VBDA-HotSpot, has a similar flag `-XX:+SC`.

As mentioned in Section 2.2, to execute Java bytecode instructions, HotSpot employs just-in-time (JIT) compilation. In this style, bytecodes are first executed in interpreter mode, with minimal optimizations. During execution, HotSpot identifies parts of the code that are frequently executed ("hot spots") and compiles them to optimized native code for better performance.

The HotSpot JVM has one interpreter and two just-in-time compilers. I have implemented the `volatile`-by-default semantics for Java server applications, which are a dominant use case in practice. Accordingly I have modified the HotSpot interpreter as well as the HotSpot server compiler. I have implemented `volatile`-by-default semantics for both X86 and ARM, the two most common hardware architectures nowadays.

I implemented VBD-HotSpot and VBDA-HotSpot by reusing mechanisms already in place for handling `volatile` variables in the interpreter and compiler.

3.3.1 `volatile`-by-default Interpreter

Since the interpreter is platform-specific, different fence instructions are used for VBD-HotSpot and VBDA-HotSpot. I will first talk about how I make the interpreter `volatile`-by-default on X86, and then present my modifications to the interpreter on ARM.

Figure 2.2 shows how the HotSpot JVM handles accesses to `volatile` variables on X86. The SC semantics for `volatile` accesses is achieved by inserting the appropriate platform-specific fences before/after such accesses. In the case of X86, which has the relatively strong *total store order* (TSO) semantics [OSS09], 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 [JSR18]. In the figure, %edx is already loaded with the field attribute for `volatile`. Instruction (1) tests if the field is declared `volatile`. If so, the `lock` instruction (2) will be executed, which acts as a StoreLoad barrier on X86; otherwise the `lock` instruction is skipped.

To implement my `volatile`-by-default semantics for X86, I therefore modified the template for `putfield` to unconditionally execute the `lock` instruction. This is done by removing instruction (1) and the following jump instruction `je`. I also added the `lock` instruction to the templates for the various bytecode instructions that perform array writes (e.g., `aastore` for storing objects into arrays, `bastore` for storing booleans into arrays, etc.).

I 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, I elide the check of the `volatile` attributes and always adds 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.

I 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 I had to modify the templates for all of the `fast_*putfield` bytecodes in order to unconditionally execute the `lock` instruction, as shown earlier for `putfield`.

Finally, the interpreter treats a small number of common and/or special methods, for example math routines from `java.lang.Math`, as *intrinsic*: the interpreter has custom assembly code for them. However, I examined the X86 implementations of these intrinsics and found that none of them contain writes to shared memory, so they already preserve SC.

Similar to X86, for ARM, I manually inspected the template instructions in the ARM interpreter for the bytecodes that read from or write to memory, such as `getfield` and `putfield`. The template code for each bytecode checks the `volatile` attribute of the given field and adds the necessary fences if the attribute is set. In VBDA-HotSpot, I 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. I 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, I examined and modified bytecode-rewriting optimizations and intrinsics in the interpreter in VBDA-HotSpot the same way I treated VBD-HotSpot.

To implement the semantics of `volatile` on ARM, the interpreter must insert a load-load and load-store barrier after a `volatile` load, providing *acquire* semantics for the load; a store-store barrier and a load-store barrier before a `volatile` write, providing *release* semantics for the write; and a store-load barrier after a `volatile` write. The interpreter uses ARM's `dmb` (data memory barrier) instruction for this purpose. In particular, it needs a `dmb ishld` instruction to enforce acquire semantics after a load, `dmb ish` to enforce release semantics before a store, and `dmb ish` to enforce store-load dependencies after a store.

However, the baseline HotSpot JVM has a bug of inserting an overly weak barrier before `volatile` writes in the interpreter. Specifically it inserts a `dmb ishst` instruction, which performs a store-store barrier but not also a load-store barrier; obtaining both barriers instead requires a `dmb ish` instruction.[2] I have fixed this bug and use the fixed version of the baseline HotSpot JVM in all of our experiments. Interestingly, the use of `dmb ishst` before writes suffices for VBDA-HotSpot, because the preceding memory operation must end in a `dmb ish` (for stores) or a `dmb ishld` (for loads), both of which act as load-store barriers.

3.3.2 `volatile`-by-default Compiler

When the JVM identifies a "hot spot" in the code, it compiles that portion to native code. As mentioned earlier, I 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

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

in the code generation phase. Optimizations are performed during each of these last two phases as well.

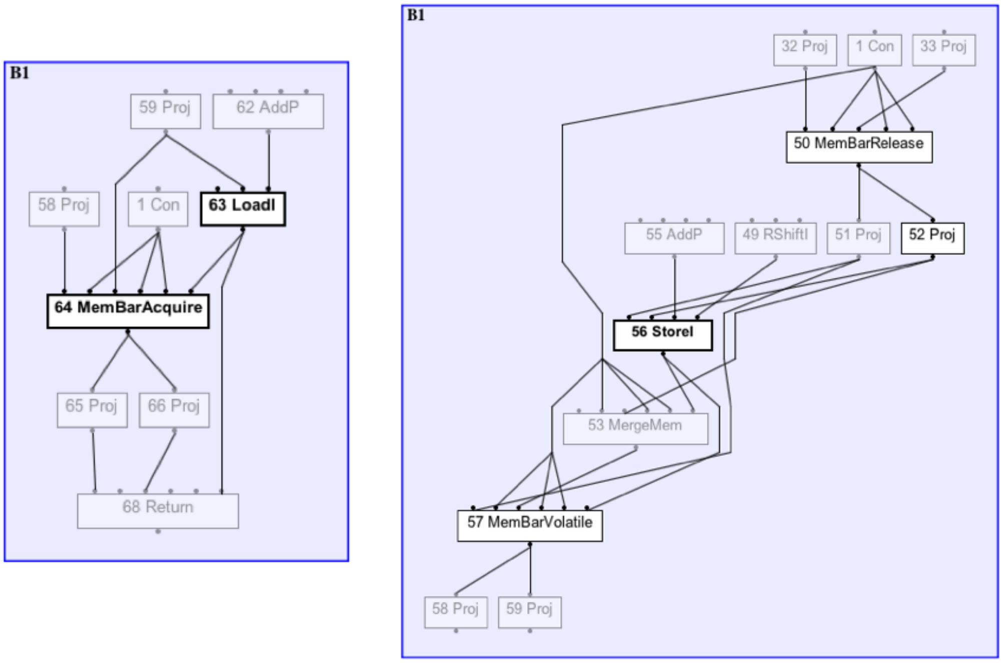


Figure 3.1: Ideal graph sections for `volatile` loads (left) and stores (right).

At the Ideal graph level, the semantics of `volatile` is implemented by three kinds of memory-barrier nodes, each of which represents a specific combination of the four basic memory barriers: *LoadLoad*, *LoadStore*, *StoreLoad*, and *StoreStore*. Figure 3.1 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.[3]

The memory-barrier nodes in the Ideal graph are translated to their counterparts in the

[3] On the POWER processor [MMS12], which is not multi-copy atomic, a `MemBarVolatile` also precedes each `volatile` load, but this is not necessary for X86.

lower-level IR. When generating machine code, they are finally translated into the appropriate assembly instructions. On X86 both the `MemBarAcquire` and `MemBarRelease` nodes become no-ops, since TSO already enforces those instruction orders. On ARM, `MemBarAcquire` becomes a `dmb ish ld`, and both `MemBarRelease` and `MemBarVolatile` become `dmb ish`. However, it is critical to keep these memory-barrier nodes in the code until the point of code generation, in order to prevent the compiler from performing optimizations that violate their semantics.

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

An additional complication is that the server compiler treats many methods as intrinsic, providing a custom Ideal graph for each one. I 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, I 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. I 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, I simply set the flag `-XX:-OptimizeStringConcat`, which causes the methods to be compiled normally instead of using the intrinsic implementations.

Modifying the compiler at this early stage ensures that I need not worry about the potential for any downstream compiler optimizations to violate SC, since those optimizations already respect the semantics of memory-barrier nodes. This holds true even for the local optimizations that are performed during Ideal-graph construction. For example, the local optimizations on a `Store` node, such as redundant store elimination, already take into account the presence of any preceding memory-barrier nodes, which is necessary to avoid violating the semantics of `volatile` stores. The ideal graph is also platform independent, I then rely on the X86 and aarch64 backend to compile the memory-barrier nodes appropriately.

3.3.3 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.

I 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 I 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. I updated this code to properly remove `MemBarAcquire` and `MemBarVolatile` nodes for all non-escaping objects.[4]

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

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

3.3.4 Correctness

My 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. I also validated VBD-HotSpot's correctness in several ways. First, I 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, I created a suite of several litmus tests that sometimes exhibit non-SC behavior under the unmodified HotSpot JVM, such as a version of Dekker's mutual exclusion algorithm. I have run these litmus tests many times on the VBD-HotSpot and VBDA-HotSpot compiler and they have never exhibited a non-SC behavior, which helps lend confidence in my implementation.

3.3.5 `volatile`-by-default for Java and Scala

Finally, I note that VBD-HotSpot ensures `volatile`-by-default semantics for Java bytecode, but that does not immediately provide a guarantee in terms of the original Java source program. However, I have manually examined the widely used `javac` compiler that is part of the OpenJDK, which compiles Java source to bytecode, and ensured that it performs no optimizations that have the effect of reordering memory accesses. Hence compiling a Java program with `javac` and executing the resulting bytecode with VBD-HotSpot provides `volatile`-by-default semantics for the original program. I also examined the `scalac` compiler that compiles Scala source to Java bytecode[5] and found no optimizations that reorder memory accesses, so the same guarantees hold for Scala programs running on VBD-HotSpot.

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

3.4 Performance Evaluation of **`volatile`**-by-default

3.4.1 Performance Evaluation of VBD-HotSpot

In this section I describe my experiments that provide insight into the performance cost of SC for JVM-based server applications on X86, which are a dominant use case today. I compared the performance of VBD-HotSpot to that of the original HotSpot JVM on several benchmark suites.

3.4.1.1 DaCapo Benchmarks

The DaCapo benchmarks suite is a set of open-source Java applications that is widely used to evaluate Java performance and represents a range of application domains [BGH06]. I used the DaCapo 9.12 distribution. I excluded five of the benchmarks: *batik* and *eclipse* are not compatible with Java 8; *tradebeans* and *tradesoap* fail periodically, apparently due to an incompatibility with the `-XX:-TieredCompilation` flag[6], which VBD-HotSpot employs (see below); and *lusearch* has a known concurrency error that causes it to crash periodically.[7]

I ran the DaCapo benchmarks on my server machine and used the default workload and thread number for each benchmark. I used an existing methodology for Java performance evaluation [GBE07]. For each JVM invocation, I ran each benchmark for 20 iterations, with the first 15 being the warm-up iterations, and I calculated the average running time of the last five iterations. I ran a total of 10 JVM invocations for each test and calculated the average of the 10 averages.

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

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

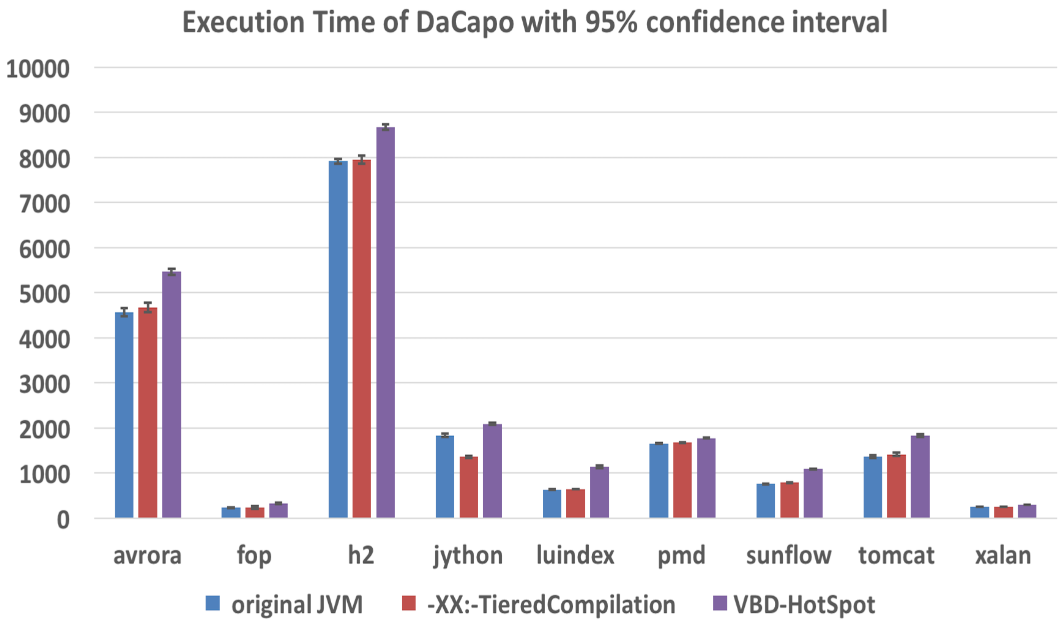


Figure 3.2: Execution time in milliseconds of VBD-HotSpot on the DaCapo benchmarks. “original JVM” means running the baseline HotSpot JVM without additional flags; “`-XX:-TieredCompilation`” means running the baseline HotSpot JVM with `-XX:-TieredCompilation`; “VBD-HotSpot” shows results of running VBD-HotSpot.

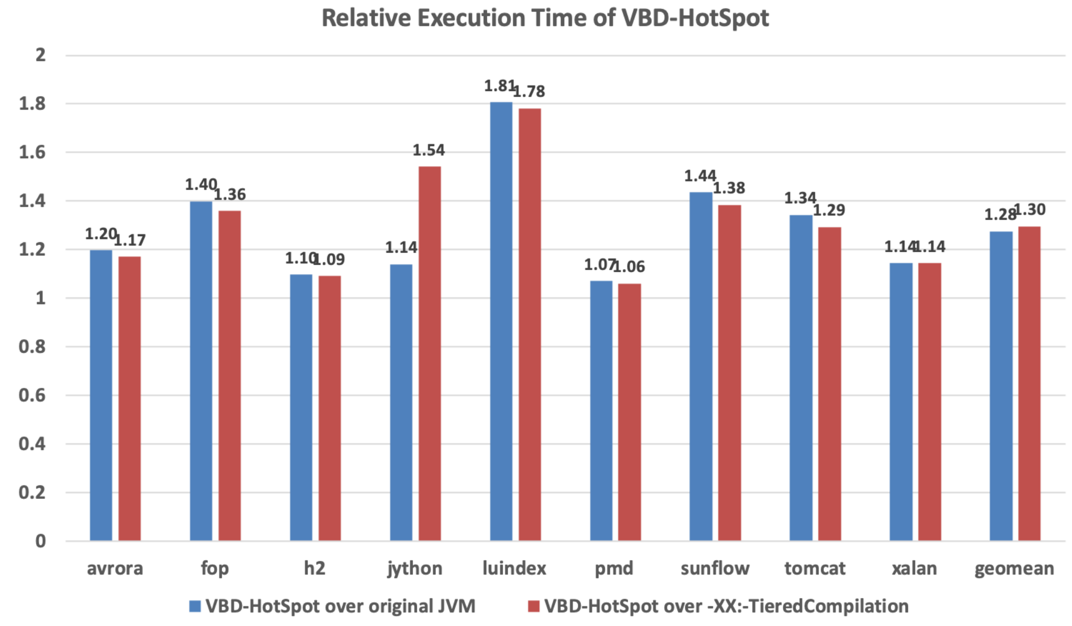


Figure 3.3: Relative execution time of VBD-HotSpot on the DaCapo benchmarks

Figures 3.2 and 3.3 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 I only modified the server compiler, VBD-HotSpot employs the `-XX:-TieredCompilation` flag to turn off tiered compilation and employ only the server compiler. Therefore I also present the results for running the original HotSpot JVM with this flag.

The geometric mean of all relative execution times represents a slowdown of 28% versus the original JVM, and the maximum slowdown across all benchmarks is 81%. The results indicate that 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 my experiments I 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.[8] 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%.

I 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 in order to ensure desired program invariants and prevent data races. Hence the overhead of such synchronization might mask the cost of additional fences and also allow some of VBD-HotSpot's inserted fences to be safely removed by HotSpot's optimizations. On

[8] `http://dacapobench.org/threads.html`

the other hand, for single-threaded programs every fence I add incurs additional overhead.

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

3.4.1.2 Spark Benchmarks

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, I evaluated VBD-HotSpot on two benchmark suites for Apache Spark [ZXW16], a widely used framework for data processing. Specifically, I employ two sets of Spark benchmarks provided by Databricks as part of the `spark-perf` repository[9]: `spark-tests` includes several big-data analytics applications, and `mllib-tests` employs Spark's MLlib library [MBY15] 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.

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

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

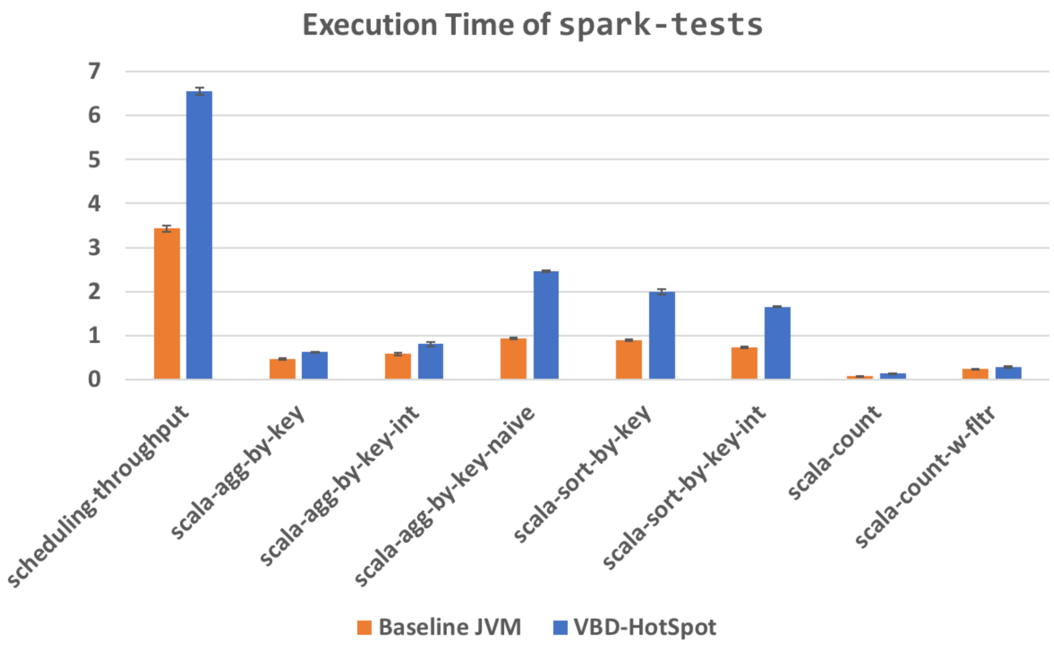


Figure 3.4: Median execution time in seconds for `spark-tests`

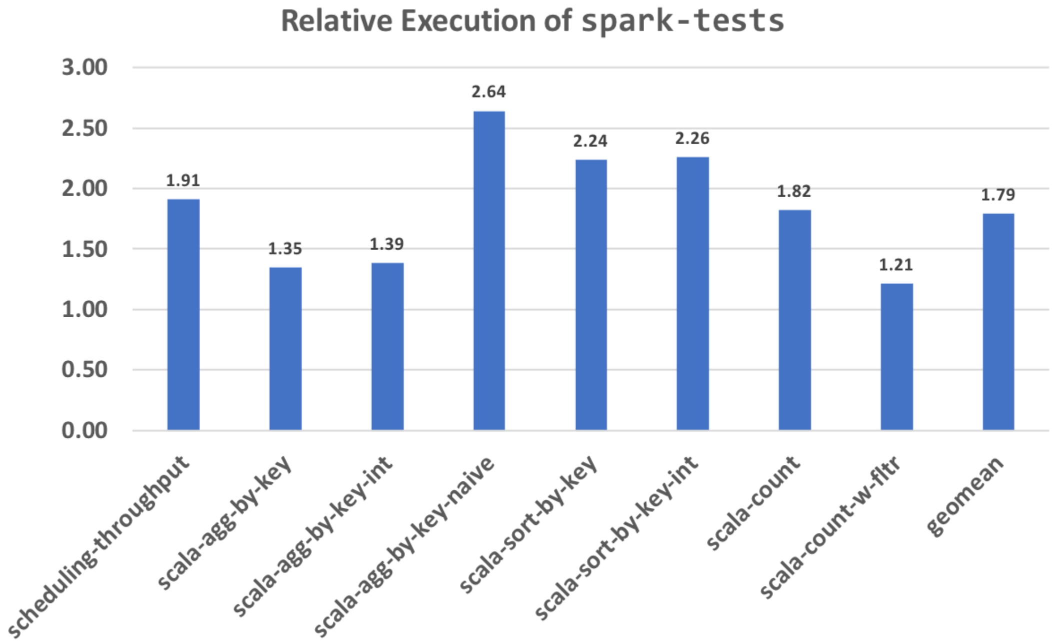


Figure 3.5: Relative execution time of VBD-HotSpot over the baseline JVM for `spark-tests`

Figure 3.4 shows the median execution times for the eight `spark-tests` benchmarks when run on the original HotSpot JVM as well as on VBD-HotSpot, with 95% confidence intervals.

Figure 3.5 shows the same results but as a relative execution time of VBD-HotSpot over the baseline HotSpot JVM. The geometric mean of the overhead of VBD-HotSpot is 79%, which is significantly higher than the average overhead of VBD-HotSpot on the Da Capo benchmarks. I surmise that the large overhead for Spark benchmarks is due Spark's use of the *resilient distributed dataset* (RDD) abstraction, which is an in-memory, immutable data structure. Each Spark operation potentially incurs many memory operations in order to read its input RDDs and write its output RDDs.

Figures 3.6 through 3.9 show the results for the `mllib` benchmarks.[10]

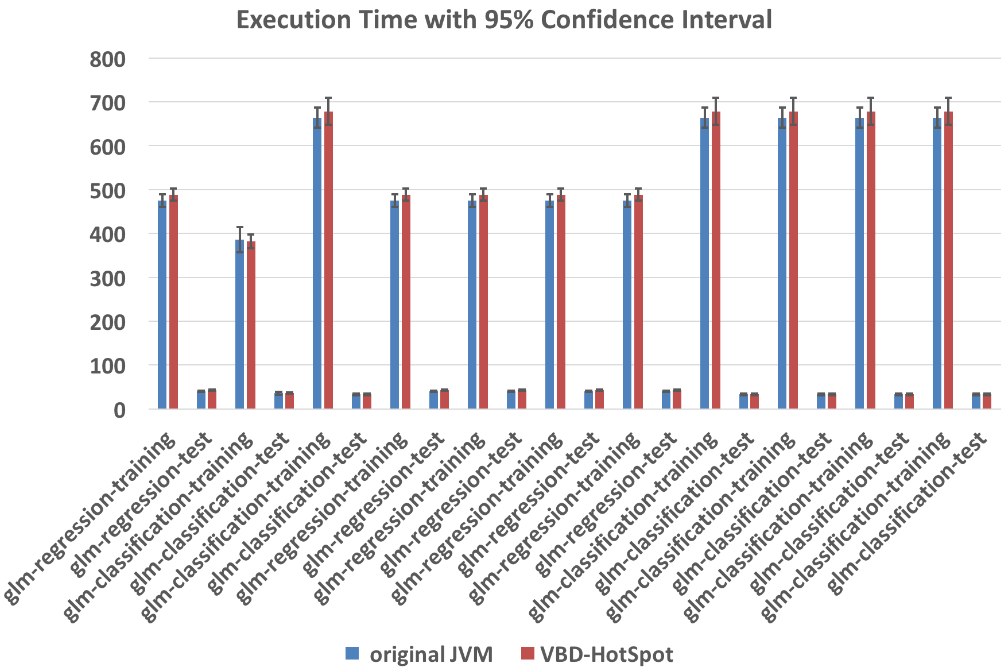


Figure 3.6: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 1 of 4)

[10] Note that several benchmarks use the same application code but with different arguments. Also note that the y-axis of Figure 3.6 has a different scale than that of the other figures, due to the longer execution times of its benchmarks.

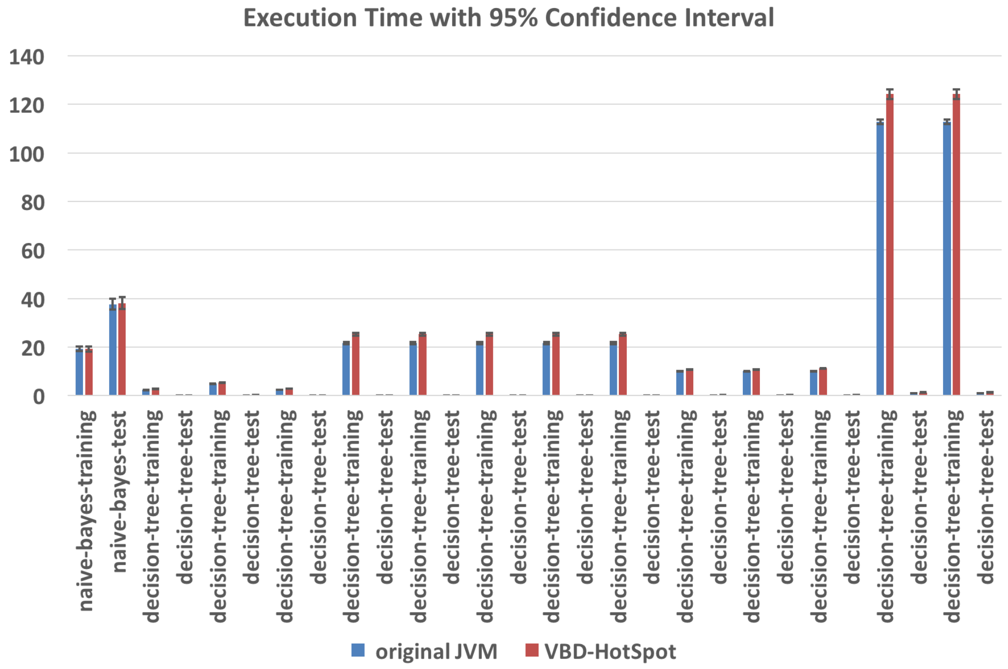


Figure 3.7: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 2 of 4)

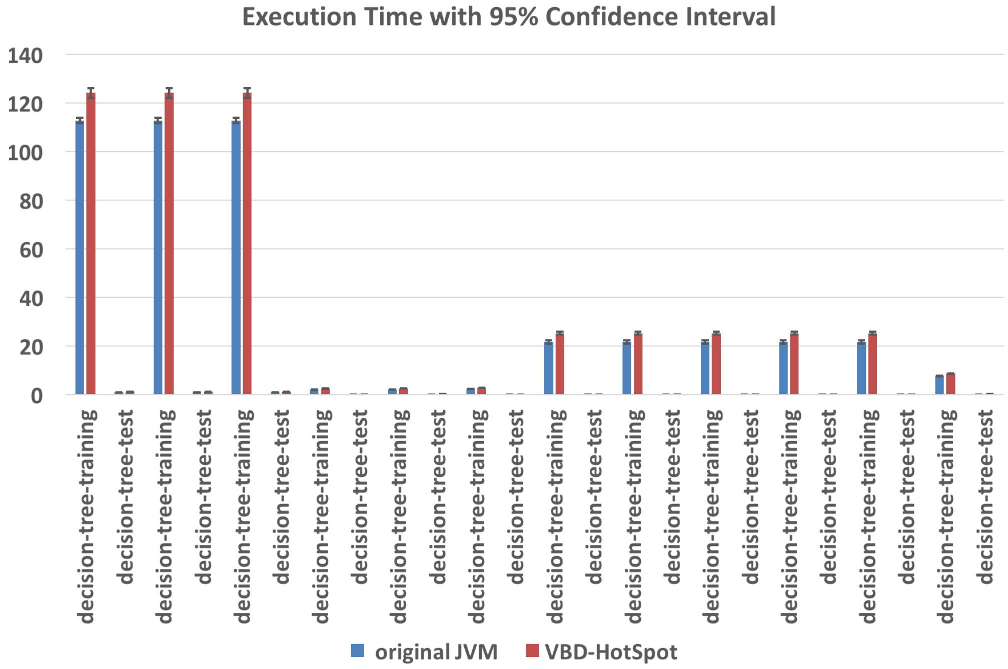


Figure 3.8: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 3 of 4)

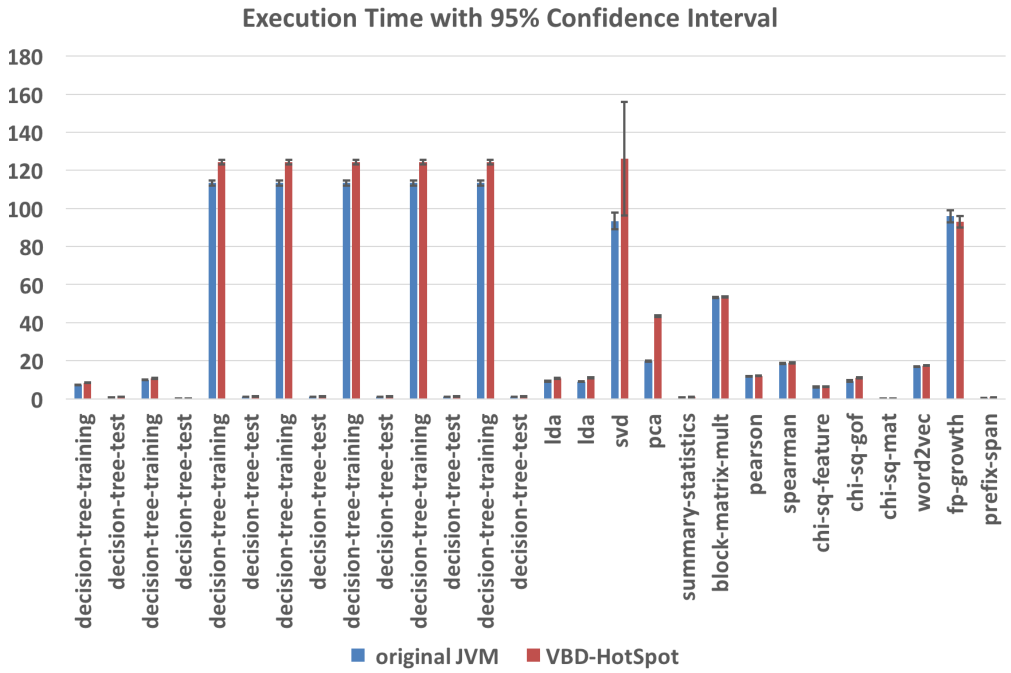


Figure 3.9: Average of 10 median execution times in seconds with 95% confidence intervals for `mllib-tests` (part 4 of 4)

Figure 3.10 shows these results in a histogram. I 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.

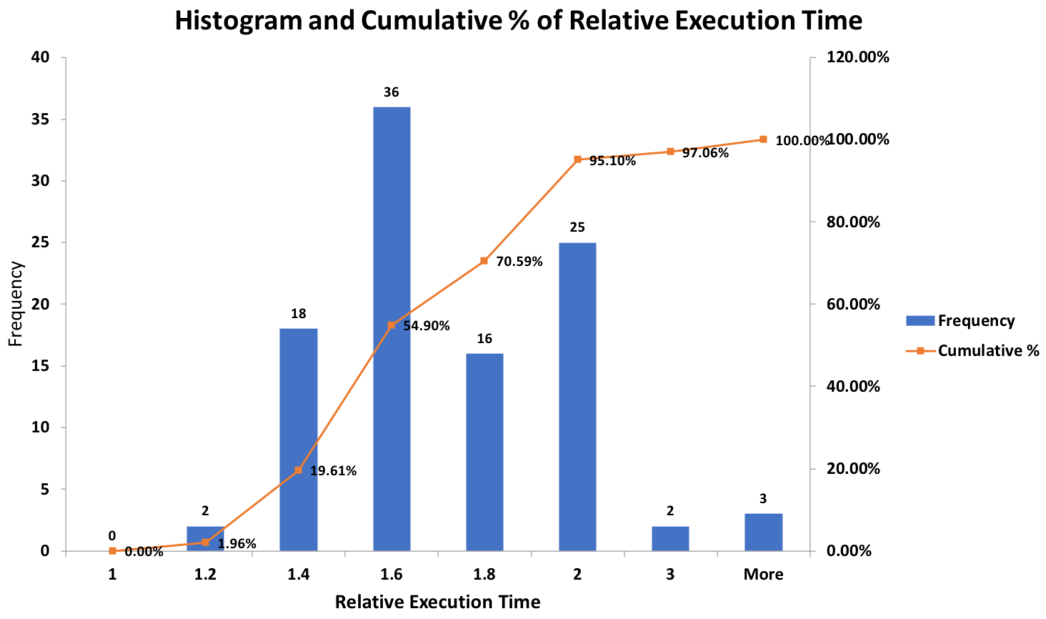


Figure 3.10: Histogram and cumulative % of relative execution time for `mllib-tests`

3.4.1.3 Scalability Experiments

I performed two experiments to understand how the cost of the `volatile`-by-default semantics changes with the number of threads/cores available. My 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. I were interested to understand how the cost of VBD-HotSpot would change with increased concurrency in general, as well as the cost difference on cores within one socket versus cores across multiple sockets.

For these experiments I 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 3.4.1.1. It also does not apply to *avrora* and *pmd* — though these benchmarks use multithreading internally, they always use a single driver thread. Therefore my experiments only employed the remaining four DaCapo benchmarks.

First I tested the overhead of the four benchmarks with 1, 3, 6, 9, 12, and 24 driver

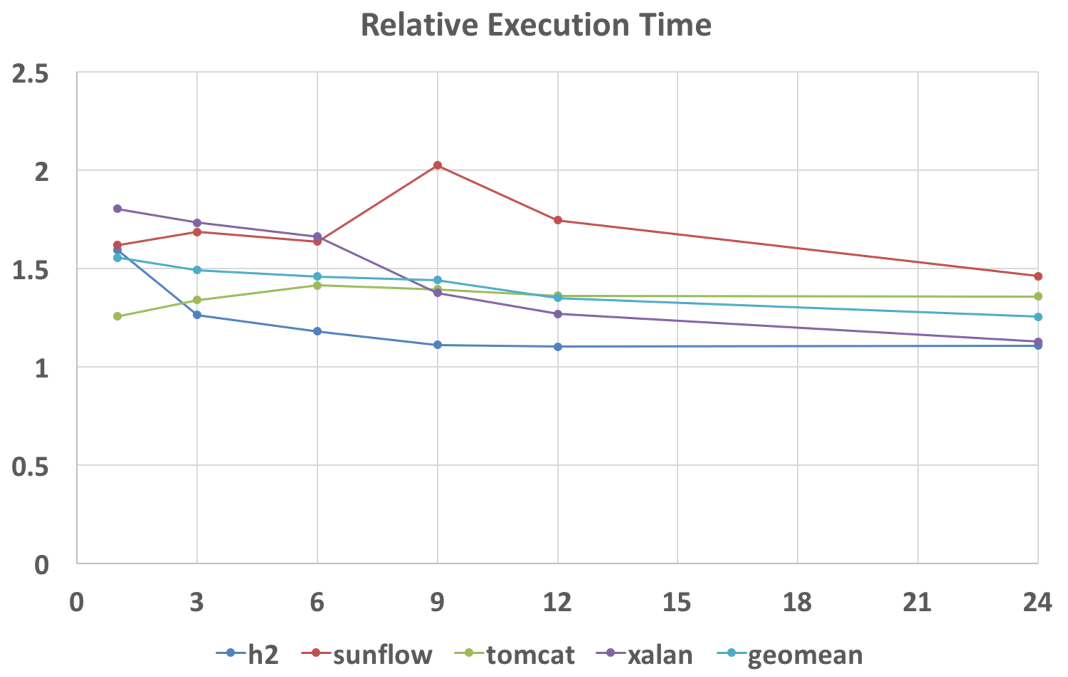


Figure 3.11: Relative cost of VBD-HotSpot with different numbers of threads/cores

threads. For the experiment with N driver threads I 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 my experiment are shown in Figure 3.11. The y-axis shows the relative execution time of running on VBD-HotSpot versus the baseline HotSpot JVM on each benchmark, and the x-axis provides this result for differing numbers of driver threads. Figure 3.12 provides the results in a different way, showing the absolute execution times in milliseconds with different numbers of driver threads. As the number of driver threads/cores increases from 1 to 12, there is a trend of improved performance for VBD-HotSpot relative to the original HotSpot JVM. The relative execution time then is flat or decreases modestly at 24 driver threads. These results belie a common assumption that SC performance suffers with increased concurrency. They also accord with an experiment by a previous work [Boe12] showing that a lock-based version of a particular parallel algorithm scales better than a version with no synchronization.

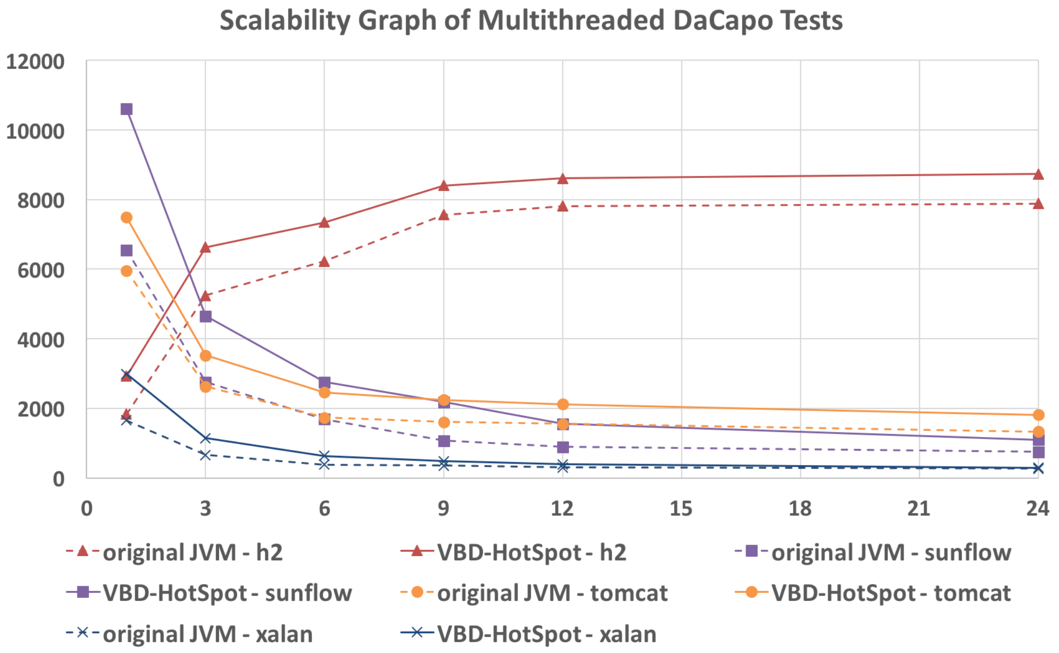


Figure 3.12: Scalability graph with different numbers of threads/cores

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

Figure 3.13 shows the relative execution times for each configuration on VBD-HotSpot versus that same configuration executed on the original HotSpot JVM. The overhead of VBD-HotSpot in the multiple-sockets configuration is uniformly lower than that in the single-socket configuration, sometimes significantly so. This is an interesting result because in my experience SC is assumed to be more expensive cross-socket, due to the increased overhead of fences. However, it appears that the cross-socket configuration slows the rest of the execution down to a greater degree, thereby offsetting the additional cost of fences.

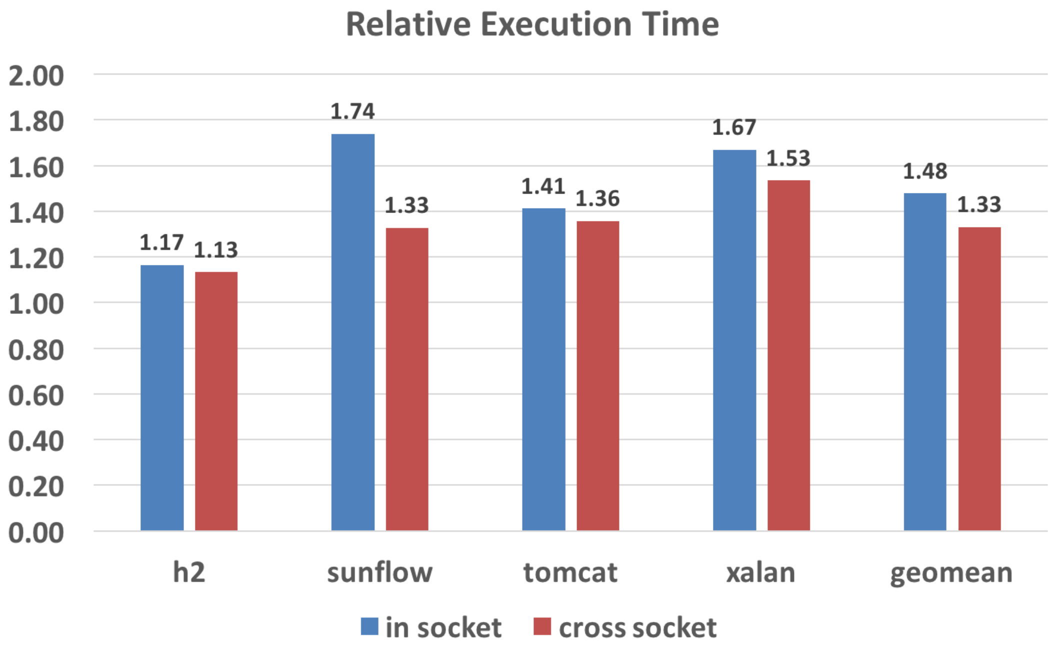


Figure 3.13: Cost of VBD-HotSpot within socket and cross-socket

3.4.1.4 Relaxed Execution

I also performed experiments to gauge the potential for judicious usage of `relaxed` annotations to improve the performance of VBD-HotSpot. I profiled four of the five Da Capo benchmarks that incur the most overhead for VBD-HotSpot[11] to determine the methods in which each benchmark spends the most execution time. Figure 3.14 shows how the overheads of these benchmarks are reduced when the top k methods in terms of execution time are annotated as `relaxed`, for k ranging from 0 to 20. Declaring a method to be `relaxed` causes the method to be compiled exactly as in the original HotSpot JVM, so memory-barrier nodes are only inserted for accesses to variables that are explicitly declared `volatile`. Note that the interpreter still executes these methods with `volatile`-by-default semantics, and any methods called by these methods are both interpreted and compiled with `volatile`-by-default semantics.

The figure shows that annotating the top 20 or fewer methods as `relaxed` provides a

[11] I were not able to perform this experiment for *tomcat* as my profiler crashes when running this benchmark.

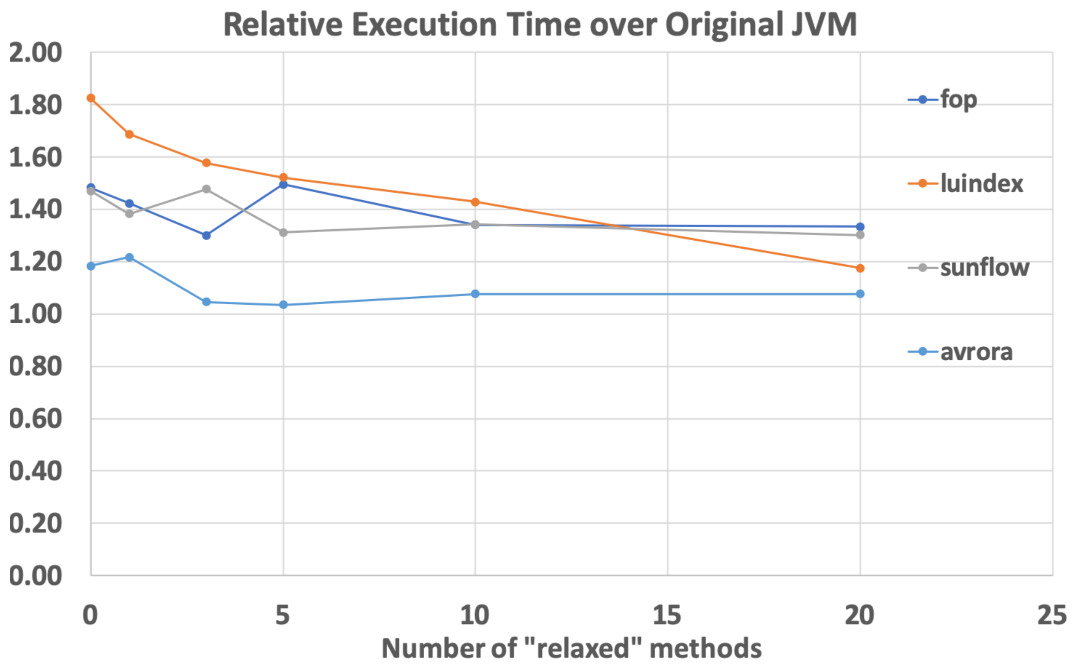


Figure 3.14: Cost of VBD-HotSpot with `relaxed` methods

large reduction in the overhead of VBD-HotSpot. One benchmark have 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.

With 20 `relaxed` annotations each on the five benchmarks in Figure 3.14, the geometric mean of VBD-HotSpot's overhead reduces to 18% for the entire Da Capo suite (with a max overhead of 34% for *tomcat*). These results show the `volatile`-by-default semantics can be a practical choice on modern server hardware, with a judicious choice of `relaxed` annotations.

3.4.1.5 Consumer PCs

Finally, I also ran my benchmarks on several consumer PC machines, in addition to my server machine. PC1 is a 6-core machine with an Intel Core i7-3930K CPU (3.20GHz), which was released in the fourth quarter of 2011. PC2 is a 4-core machine with an Intel Core i7-4790 CPU (3.20GHz), which was released in the second quarter of 2014. PC3 is a 4-core machine with an Intel Core i7-6700 CPU (3.40GHz), which was released in the third quarter of 2015.

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

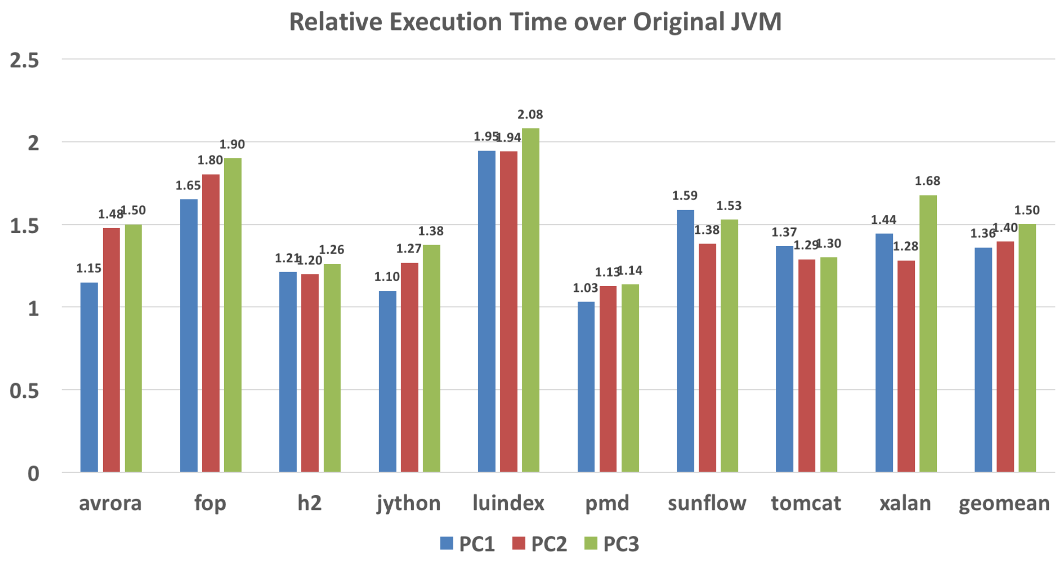


Figure 3.15: Relative execution time of the DaCapo benchmarks

I ran the DaCapo benchmarks on these machines using the same setup as in Section 3.4.1.1. Figure 3.15 shows the relative execution time for VBD-HotSpot of the benchmarks on the three machines, normalized to the execution time when run on the baseline HotSpot JVM. The geometric mean of the overhead due to the `volatile`-by-default semantics is respectively 36%, 40%, and 50% on machines PC1, PC2, and PC3, which is somewhat higher than the overhead of VBD-HotSpot on my server machine (Section 3.4.1.1).

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

3.4.2 Performance Evaluation of VBDA-HotSpot

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

3.4.2.1 DaCapo Benchmarks

The DaCapo benchmark suites are a widely used set of Java applications to evaluate Java performance [BGH06]. I use the latest maintenance release (9.12-MR1) of the DaCapo benchmarks from January 2018. Among all tests, I remove `batik` which fails on the baseline aarch64 port of OpenJDK 8u (even without any of my modifications), `tradesoap` which fails periodically, apparently due to an incompatibility with the `-XX:-TieredCompilation` flag that VBDA-HotSpot requires (as discussed below)[12], and `tomcat` due to a problem unrelated to DaCapo[13]. I 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.

I used the default workload and thread numbers for each benchmark. I employed an existing methodology for Java performance evaluation [GBE07]. In each JVM invocation I ran a benchmark for 15 warm-up iterations and then calculated the average running time of the next five iterations. I ran five invocations of each benchmark using this process and calculated the average of these per-invocation averages. Finally I calculated the relative execution time of each benchmark using the average of the averages and then calculated the geometric mean of the relative execution times over all benchmarks.

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

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

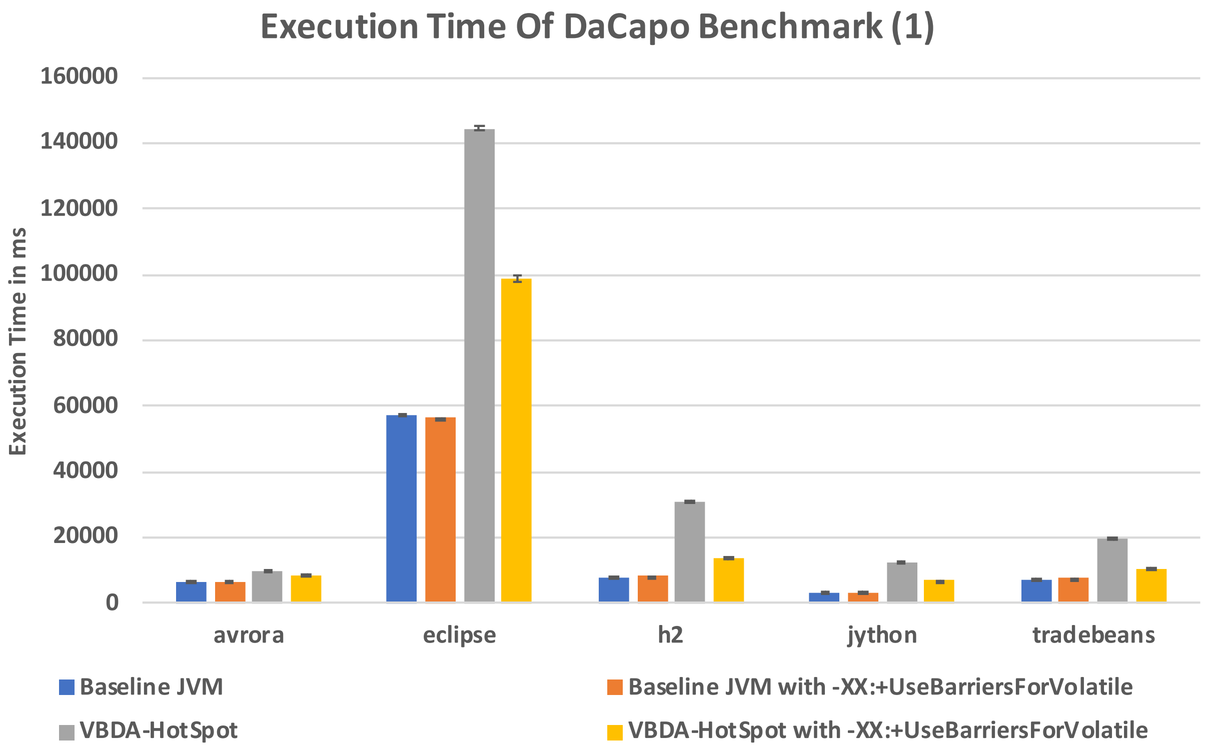


Figure 3.16: Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

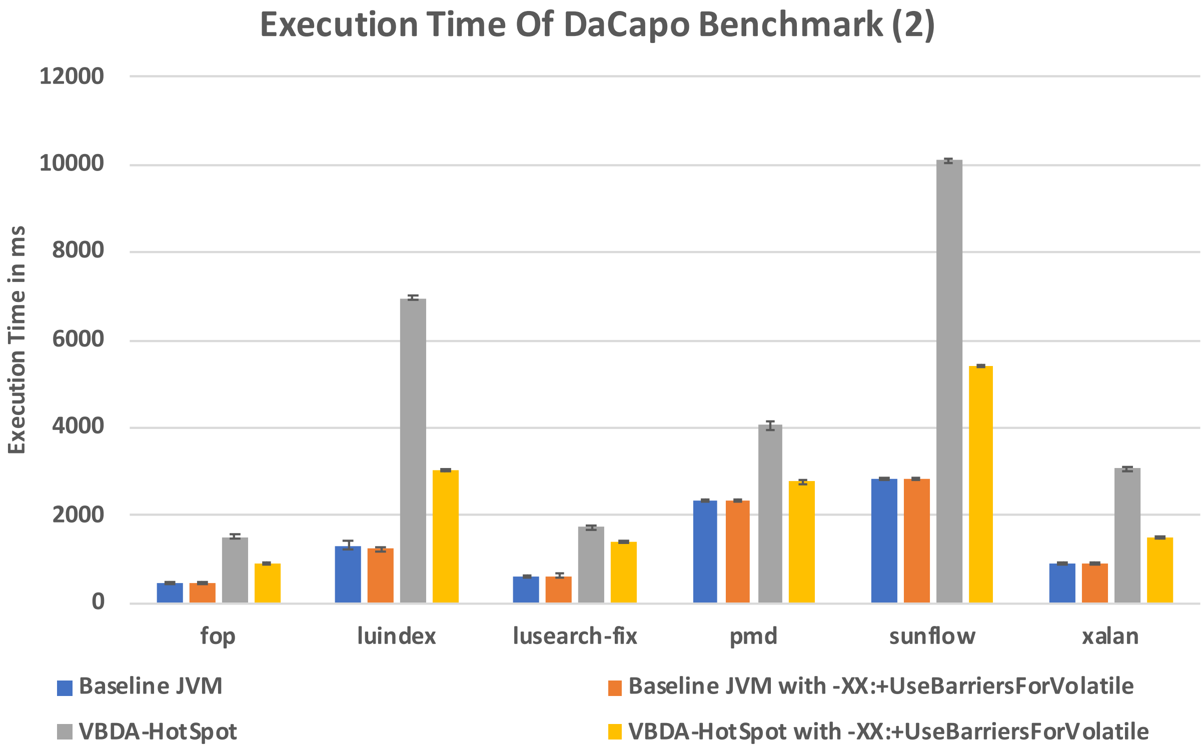


Figure 3.17: Absolute execution time of VBDA-HotSpot and baseline JVM for DaCapo benchmarks on machine A.

Figures 3.16 and 3.17 show the execution time in ms for the baseline JVM and VBDA-HotSpot on machine A. The error bars show 95% confidence intervals. I use the flag `-XX:-TieredCompilation` in all versions in order to turn off *tiered compilation*, which employs multiple compilers, since I have only modified the server compiler to respect the `volatile`-by-default semantics. I have verified that for the baseline HotSpot JVM, there is very little performance difference with and without tiered compilation on the DaCapo benchmarks.

The figures show that for the baseline JVM, the performance with or without the `-XX:+UseBarriersForVolatile` flag is almost the same (1% difference). However, VBDA-HotSpot is much faster with the flag than without it, even though the new one-way fences are intended to improve the performance of `volatile` accesses. On further investigation, I identified two causes for this counter-intuitive behavior. First, I 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 hardware that I use is still not exploiting the release-acquire semantics of the one-way fence instructions in their implementation.

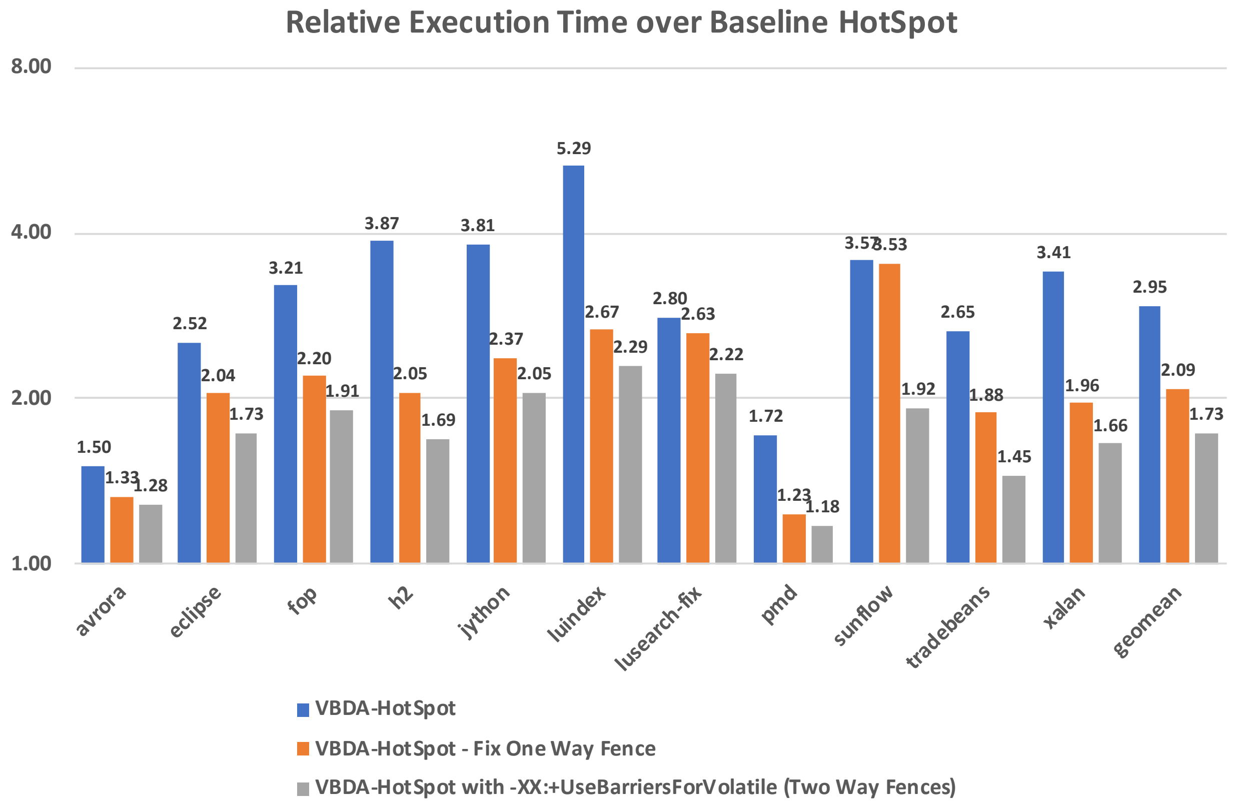


Figure 3.18: Relative execution time of VBDA-HotSpot, VBDA-HotSpot with bug fix for one way fences, VBDA-HotSpot with two way fences on machine A, y-axis in logarithmic scale.

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.[14] Fixing those heuristics as suggested in the bug report makes a dramatic difference, as shown in Figure 3.18, 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

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

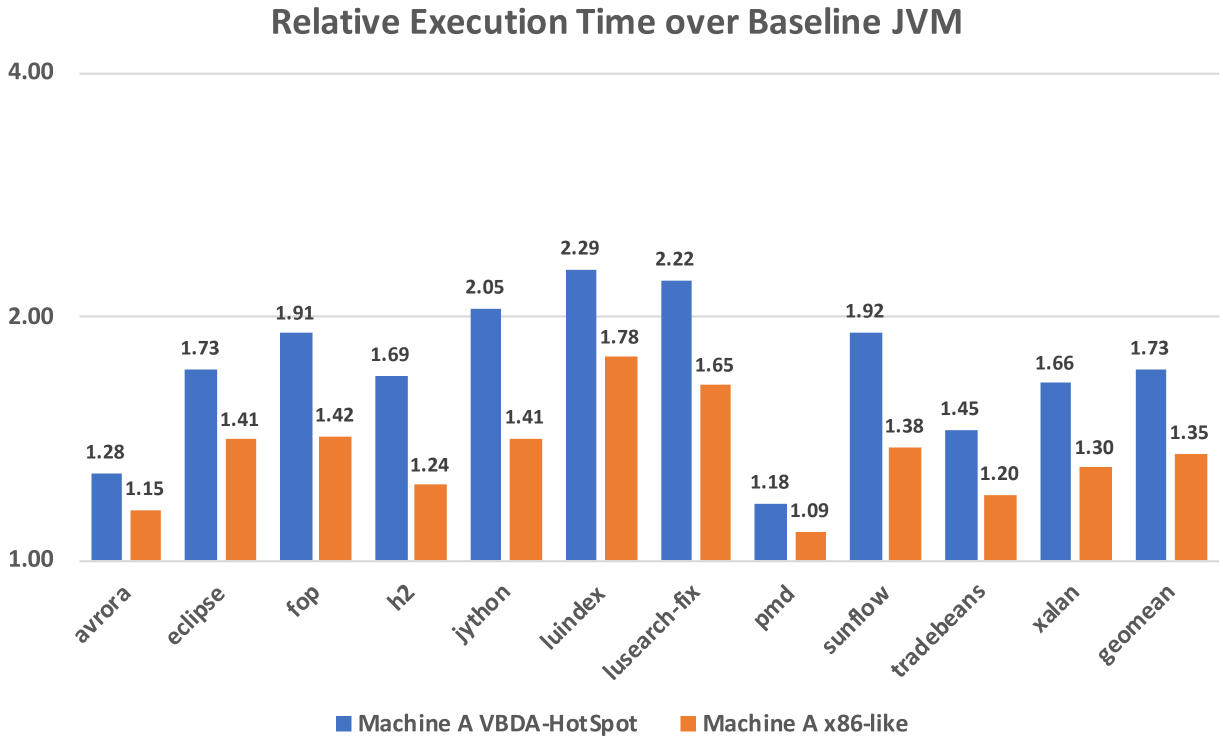


Figure 3.19: Relative execution time of VBDA-HotSpot and X86-like VBDA-HotSpot over baseline JVM for DaCapo on machine A, y-axis in logarithmic scale.

an average overhead of 73%. Therefore, in the rest of the paper I report numbers with the `-XX:+UseBarriersForVolatile` flag for VBDA-HotSpot.

The first series in Figures 3.19 and 3.20 respectively shows the relative execution time of VBDA-HotSpot over the baseline HotSpot JVM on machine A and machine B (for machine A these are the same numbers as shown in the last series in Figure 3.18). 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, I also implemented an “X86-like” version of VBDA-HotSpot that inserts the store-load barriers after each store but removes all other barriers in the interpreter, in the intrinsics implementations, and in the ideal graph for the compiler. The second series in Figures 3.19 and 3.20 respectively shows the relative

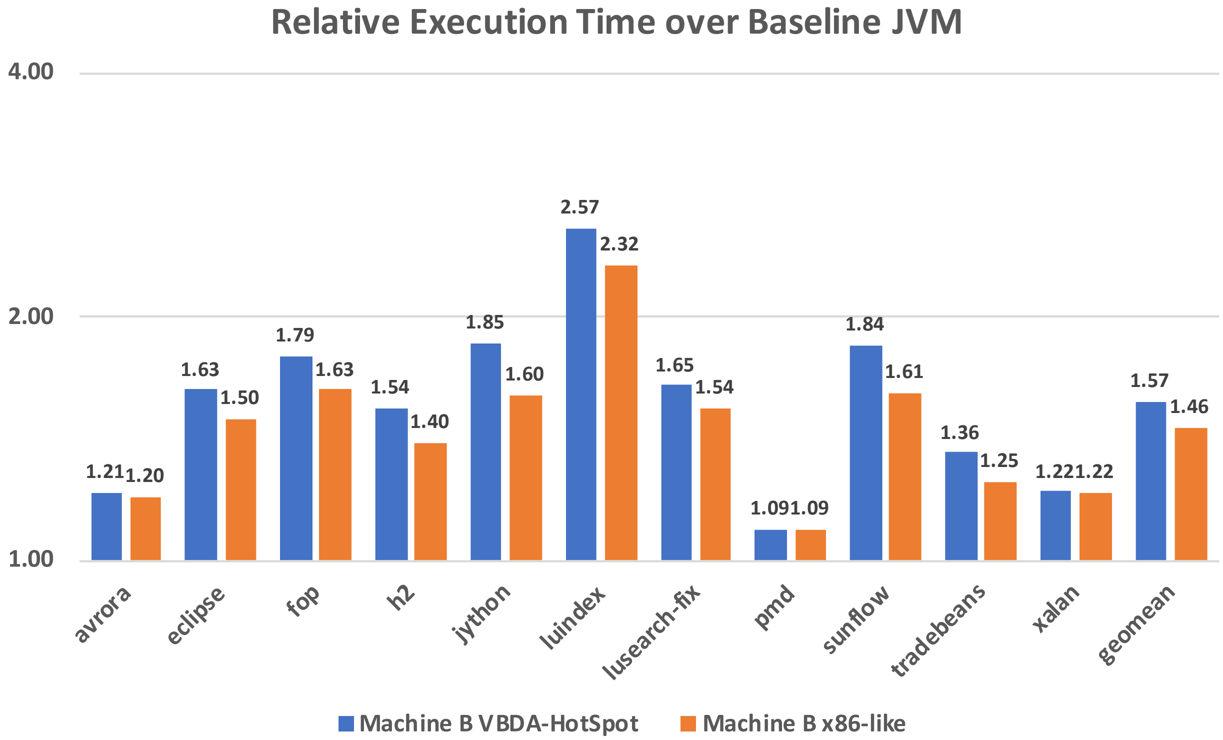


Figure 3.20: Relative execution time of VBDA-HotSpot and X86-like VBDA-HotSpot over baseline JVM for DaCapo on machine B, y-axis in logarithmic scale.

execution time of this X86-like VBDA-HotSpot over the baseline JVM on machine A and machine B.

The X86-like VBDA-HotSpot results in average and maximum overheads of 35% and 78% for DaCapo on machine A, and average and maximum overheads of 46% and 132% for machine B. In other words, the additional fences on reads required by VBDA-HotSpot only double the overhead versus the X86-like version, despite the fact that reads dominate writes in typical programs. The X86-like implementation must insert a full fence, `dmb ish`, after a `volatile` write to implement the store-load barrier, so it seems that additional fences do not incrementally add much overhead.

3.4.2.2 Scalability Experiments

I also performed experiments on both machine A and machine B to understand how the overhead of VBDA-HotSpot changes with the number of threads/cores available, as an indication of how the cost of `volatile`-by-default may change as processors introduce more cores. Since committing a memory operation involves inter-core communication (e.g., coherence messages) and fences require a core to stall until all prior operations have committed, one may expect the overhead of the `volatile`-by-default approach to increase with the number of cores. I previously showed that in fact the relative overhead of `volatile`-by-default decreases or stays the same as the number of cores increases on X86 [LMM17], apparently because of the additional cost of regular loads and stores, but I were interested to investigate whether the same would hold true on the weaker ARM platform.

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

Experiments on both machines show a similar trend as in the prior X86 work: as the number of driver threads/cores increases, the relative overhead of VBDA-HotSpot stays the same or decreases modestly. The results of my experiment on machine B are shown in Figure 3.21; the results for machine A are similar. The figure shows how the relative execution time of VBDA-HotSpot changes with different numbers of external threads. Interestingly, VBDA-HotSpot is faster than the baseline HotSpot JVM for the `xalan` benchmark at 36 and 48 cores. Plotting the absolute execution times, as shown in Figure 3.22, I see that the baseline JVM stops scaling after 24 threads while VBDA-HotSpot stops at 48 threads.

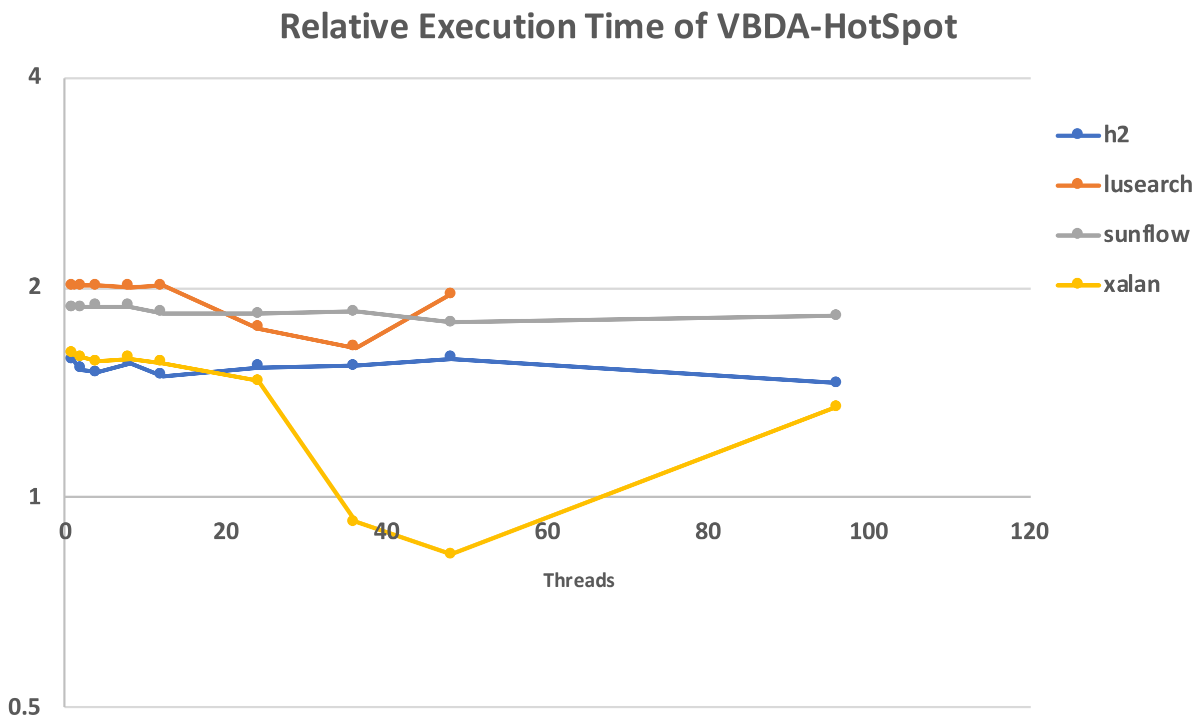


Figure 3.21: Relative cost of VBDA-HotSpot with different numbers of threads/cores on machine B. `lusearch` does not support running with 96 threads, y-axis in logarithmic scale.

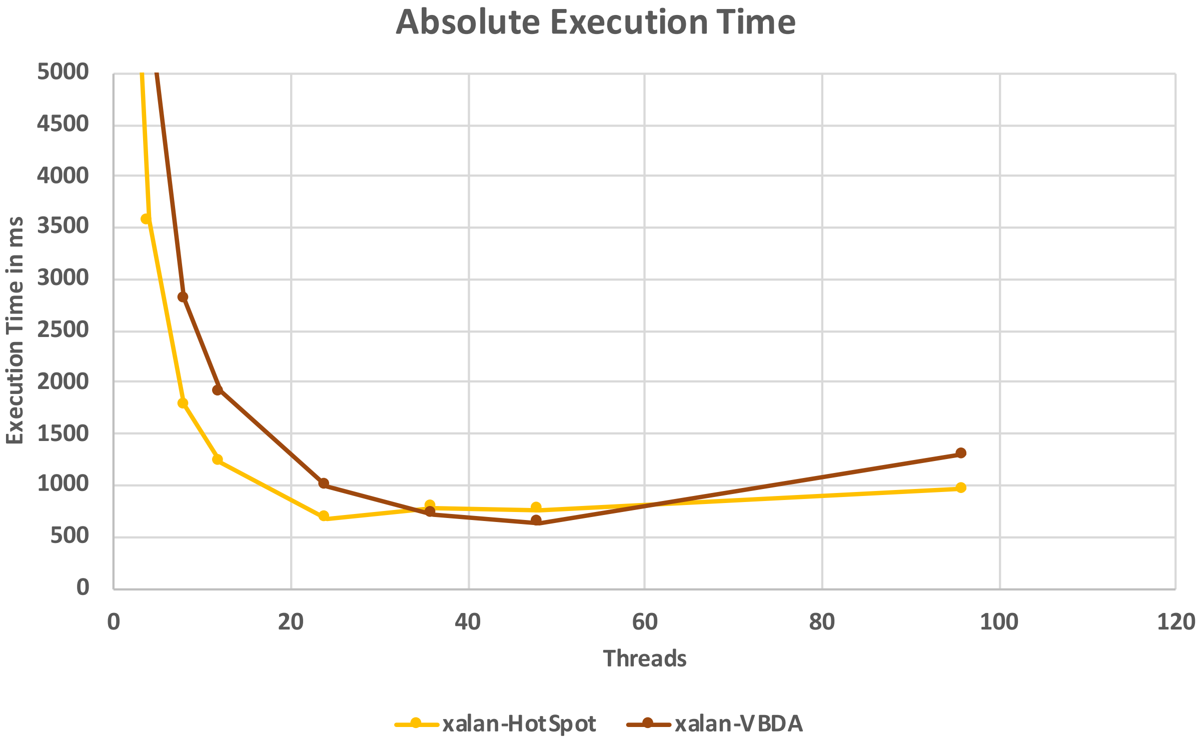


Figure 3.22: Execution time of VBDA-HotSpot and baseline JVM running `xalan` with different numbers of threads/cores on machine B.

Cross-socket memory accesses tend to be more expensive than within-socket accesses. To understand how the additional fences of VBDA-HotSpot affect this trend, I performed an experiment to measure the relative performance difference for VBDA-HotSpot when running on cores within the same socket versus on cores across sockets. I ran the DaCapo benchmarks with 2 driver threads in two different configurations: one using cores 0 and 1, which are on the same socket, and one using cores 0 and 2, which are on different sockets. Figure 3.23 shows the relative execution times for each configuration on VBDA-HotSpot versus that same configuration executed on the baseline JVM. From the results I can see that the relative overhead of VBDA-HotSpot in the multiple-sockets configuration is the same or slightly lower than that in the single-socket configuration. These results are again consistent with the earlier experiments on X86 [LMM17].

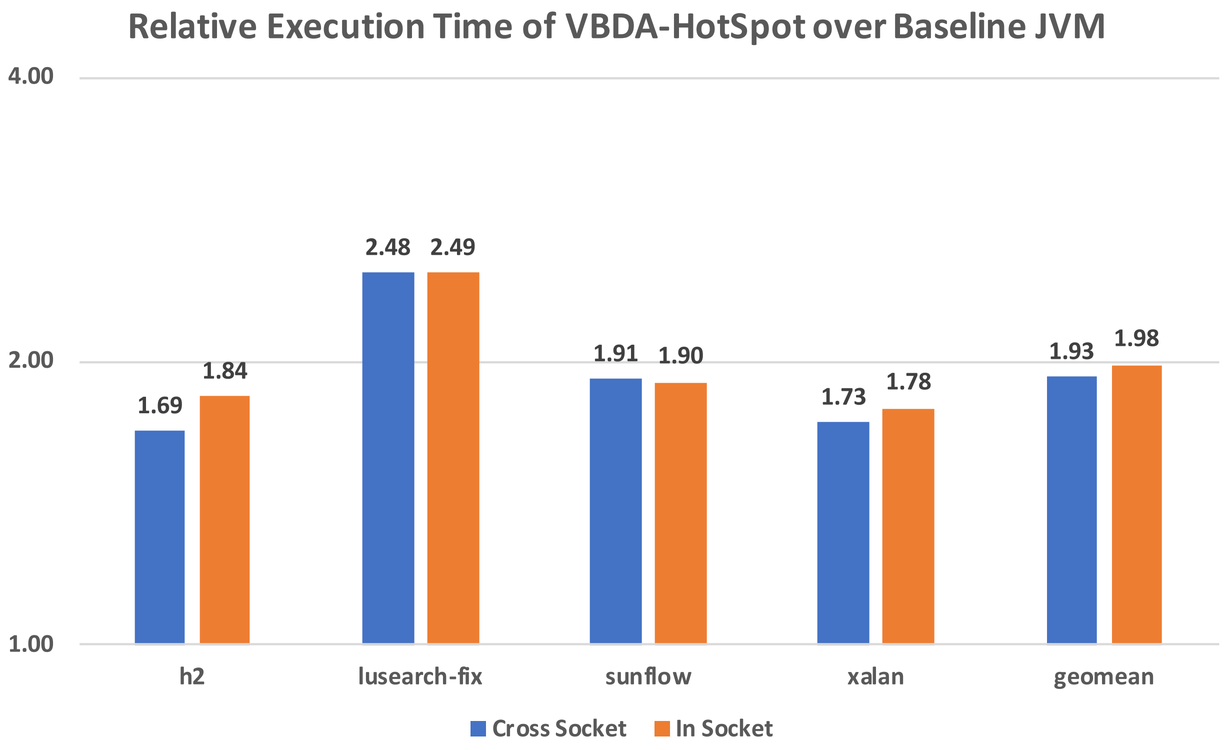


Figure 3.23: Cost of VBDA-HotSpot cross-socket and within socket on machine A, y-axis in logarithmic scale.

3.4.2.3 Spark Benchmarks

Finally, I tested VBDA-HotSpot's performance on big data and machine learning benchmarks for Apache Spark [ZXW16]. As in prior work [LMM17] I use the `spark-tests` and `mllib-tests` benchmarks from the `spark-perf` repository[15] provided by Databricks.

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

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

of the median execution times of each test.

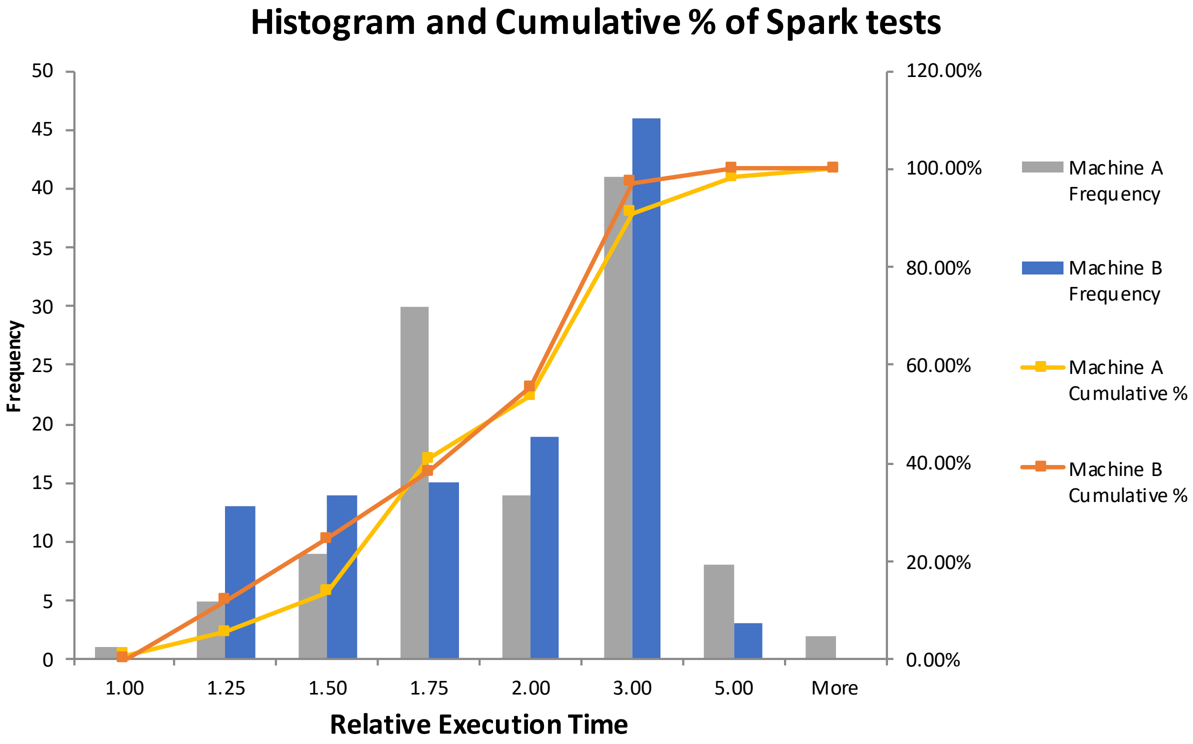


Figure 3.24: Histogram and cumulative % of relative execution time of VBDA-HotSpot for Spark benchmarks.

Figure 3.24 summarizes the results in a histogram. For example, the second gray bar from the left indicates that on Machine A there are 5 benchmarks that incur a relative execution time between 1.00 (exclusive) and 1.25 (inclusive). I omitted four benchmarks that failed on the original JVM (*als*, *kmeans*, *gmm*, and *pic*). The geometric mean of VBDA-HotSpot's relative execution time on Machine A and Machine B is 2.03 and 1.85 respectively, representing a 103% and 85% overhead over the baseline HotSpot JVM. These results are consistent with those found for VBD-HotSpot in the prior work[LMM17].[16] I suspect that the high overhead for Spark tests versus the DaCapo benchmarks is due to the many array reads and writes that are necessary to implement Spark's key data structure, the resilient distributed dataset (RDD) [ZXW16]. Indeed, on a version of VBDA-HotSpot that (unsoundly) omits fences

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

for array-element accesses, the average overhead of the Spark benchmarks on Machine A is reduced from 103% to 46%.

CHAPTER 4

A Speculative Implementation of `volatile`-by-default

As shown in the previous section, the baseline implementation of the `volatile`-by-default semantics for Java incurs considerable overhead on a weak platform like ARM. While adding `relaxed` annotations is one way to reduce the overhead of VBDA-HotSpot, the results above argue for the need to explore optimization techniques for language-level SC on stock weak-memory-model hardware.

I 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, any speculatively compiled code for the object is removed, and future JITed code for the object will include the necessary fences in order to ensure SC. I call the version of HotSpot that uses this approach to ensuring the `volatile`-by-default semantics S-VBD.

4.1 Design Overview

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

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

Instead, I 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, I compare the recorded ID to the ID of the current thread, in order to decide whether the object can still be treated as *safe* or not. Once an object becomes *unsafe* it remains so for the rest of its lifetime, so no more checking is required.

Even though checking thread locality is much less expensive than checking for data races, the runtime overhead would be prohibitive if I do this check on every field access. However, a key property of the way I 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, I can safely replace multiple per-access checks to an object with a single check at the beginning of the method. Note that this is sound even if an object becomes non-thread-local in the middle of a method — the second thread that accesses the object will force a decompilation of this method. I 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 I 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 my implementation I instead switch on a per-class basis. That is, as soon as any instance of class `C` becomes *unsafe*, I 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, I must decide how to switch from *safe* to *unsafe* mode in a correct and low-complexity way. I 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). I show how to leverage this capability for my purpose. Specifically I use HotSpot's *dependency tracking* mechanism to record the speculatively compiled *fast* methods that may access fields of objects of a given class `C`. The first time that some instance of `C` is found to be *unsafe*, S-VBD invokes HotSpot's deoptimization facility to safely pause all threads and remove the compiled versions of all methods that depend on `C` before resuming execution. If JIT compilation is later triggered on any of these methods, the *slow* versions will be used.

Finally, I have described my 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 I currently treat these accesses exactly as in VBDA-HotSpot. Specifically, I unconditionally insert the appropriate memory barriers for these accesses to ensure the `volatile` semantics. I also unconditionally insert fences for intrinsics as in VBDA-HotSpot.

4.2 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 my implementation in detail.

To simplify the presentation, I first describe my 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.

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

Figure 4.1: Just-in-time compilation of a method.

As mentioned above, I have implemented an intraprocedural analysis at class-load time that performs this optimization. I then describe the more general case where per-field-access checks are required in the next subsection.

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

Figure 4.1 shows what happens when a method gets "hot" enough and is chosen to be compiled. I 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 I check the class's mode again. If the class's mode has changed, it means that some object of the class has been found to be *unsafe* on another thread in the meanwhile, so I abort the compilation.

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

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

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

Figure 4.2: The *slow* and *fast* versions of a method.

Otherwise I register the compiled method for subsequent execution. (If there are inlined methods I 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 4.2 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.

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

Figure 4.3: The interpreted version of a method.

The `switch_to_slow` pseudocode in the figure illustrates the latter process. I first change the mode of the given class `C` to *slow*. The `deoptimize_to_slow` function (definition not shown) then leverages the HotSpot JVM's existing mechanism for *deoptimization* to invalidate all compiled methods that *depend upon* `C`, which includes the methods of `C` and its superclasses, as well as any methods in which one of these methods is inlined. This function also changes the mode of all of `C`'s superclasses to *slow*. The `deoptimize_to_slow` function is implemented as a "VM operation" in the HotSpot JVM, which causes all other threads to be stopped before its execution so that it can safely invalidate compiled methods. Also, I make the `switch_to_slow` function shown in Figure 4.2 atomic to prevent multiple threads from deoptimizing the same methods and to prevent the `compile` function in Figure 4.1 from concurrently registering any *fast* methods for class `C`.

Finally, I 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, I 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 4.3.

4.3 Implementing Per-Access Checks

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

As mentioned earlier, at class-load time an intraprocedural analysis identifies field accesses whose receiver object is definitely `this`, so I can avoid checks on these accesses. The analysis also rewrites all other `getfield` bytecodes in the method to a new `check_getfield` bytecode that I have defined in S-VBD, and similarly for all other `putfield`s in the method. Later, whenever a `check_getfield` bytecode is encountered during interpretation or compilation, I simply treat it as if it were an inlined call to a *getter* method on the receiver object. That is, I 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 I 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 I record a dependency of the method `C.m` on class `D` whenever I encounter such a field access, extending the dependency-tracking mechanism that HotSpot uses for deoptimization as described earlier.

4.4 Optimizing Fence Insertion

In addition to speculative compilation, I implemented an orthogonal optimization that reduces the number of fences required to enforce the `volatile`-by-default semantics for ARM. The first two rows of Table 4.1 show the memory barriers required before and/or after a `volatile` memory access in Java, as described in the JMM Cookbook [JSR18], and the corresponding

	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`

Table 4.1: The implementation for `volatile` accesses on ARM in HotSpot (first two rows) and an optimized implementation for memory accesses on ARM in S-VBD (last two rows).

ARM instructions used to achieve those barriers in HotSpot. For example, a `volatile` load requires a LoadLoad and LoadStore barrier after it, which is implemented by a `dmb ish ld` instruction in ARM.

The baseline implementation of VBDA-HotSpot, which uses the approach of VBD-HotSpot, simply inherits this implementation strategy for `volatile`s from HotSpot. However, I 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 4.1 show my 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 I have eliminated the need for any memory barriers before a VBD store. Further, while I have added a StoreStore barrier after a VBD store, the corresponding implementation of the required barriers in ARM remains the same, namely the use of a full fence `dmb ish`.

I implemented this optimized strategy, which I call `VBD-Opt`, in S-VBD. For the interpreter, I changed the barriers inserted as described above. For the server compiler, for simplicity of implementation I 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, I eliminate the `dmb ish` instruction before each store.

4.5 Performance Evaluation

4.5.1 DaCapo Benchmarks

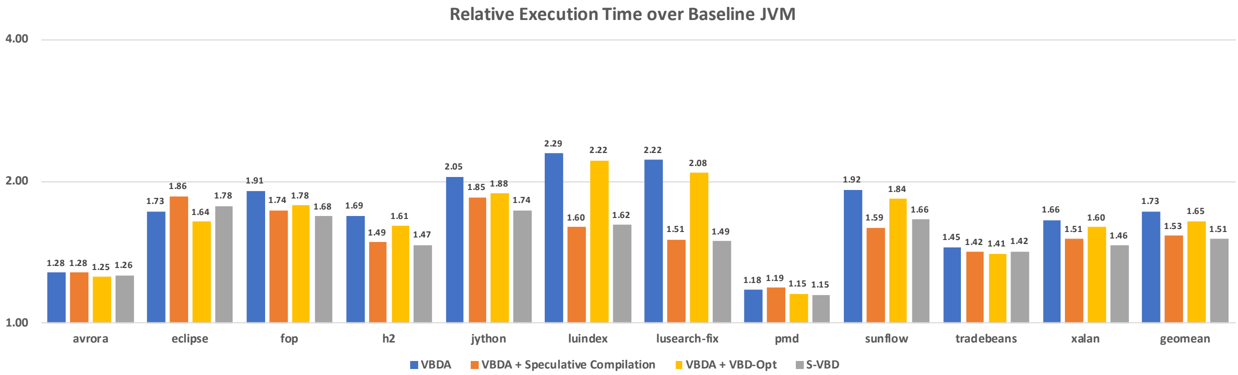


Figure 4.4: Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine A, y-axis in logarithmic scale.

I measured the peak performance of S-VBD for the DaCapo benchmarks using the same methodology as in the previous section. The fourth series in Figures 4.4 and 4.5 shows the overhead of my approach over the baseline HotSpot JVM on machine A and machine B. The geomean overhead of the S-VBD approach is respectively 51% and 37% for the two machines, which is a significant improvement over the geomean overheads of the original VBDA-HotSpot (the first series in the figures) at 73% and 57%. Also, the maximum overhead across all benchmarks respectively reduces from 129% to 78% and from 157% to 73%.

Figures 4.4 and 4.5 also isolate the effect of each of my optimizations: the second series shows the relative performance when using just speculative compilation, and the third series

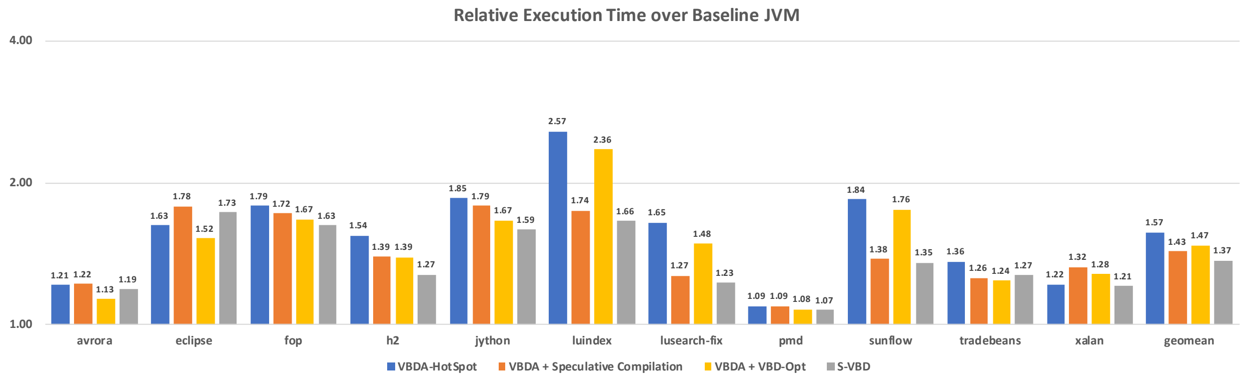


Figure 4.5: Relative execution time of VBDA + speculative compilation, VBDA + VBD-Opt, S-VBD over the baseline JVM compared to VBDA-HotSpot on machine B, y-axis in logarithmic scale.

shows the relative performance when using just the VBD-Opt fence optimization. On its own each optimization provides a considerable performance improvement, but speculative compilation clearly is the more effective optimization. As the fourth series shows, together they are even 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. I measured the startup performance of both VBDA-HotSpot and S-VBD using an existing methodology [GBE07]. I 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. I 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, I 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 4.6. 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-

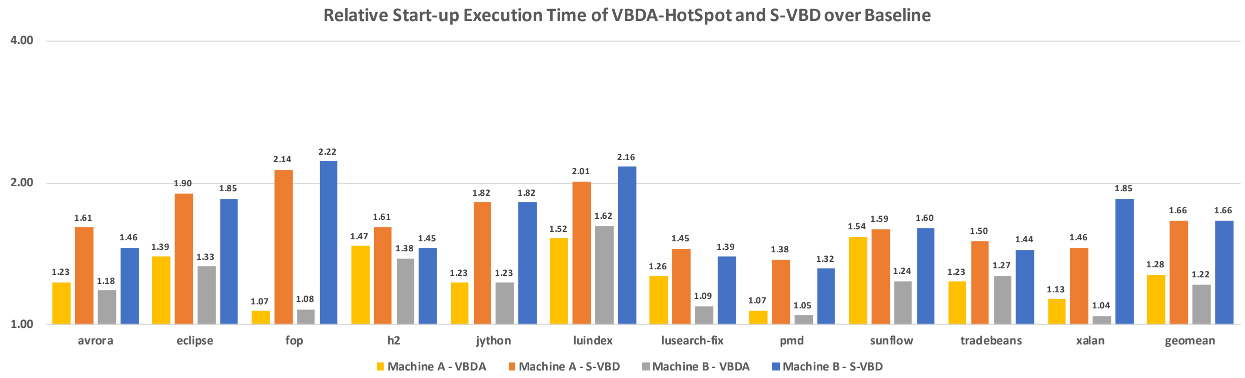


Figure 4.6: Relative startup execution time of VBDA-HotSpot and S-VBD over the baseline JVM, y-axis in logarithmic scale.

HotSpot.

4.5.2 CheckOnly Overhead

To further understand the overheads of S-VBD, I 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 4.7 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 my 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.

4.5.3 Spark Benchmarks

I also measured the peak performance of S-VBD for the `spark-perf` benchmarks using the same methodology as in the previous section. Figure 4.8 summarizes the results of `spark-tests` and `mllib-tests` in a histogram. The geometric mean of S-VBD's relative

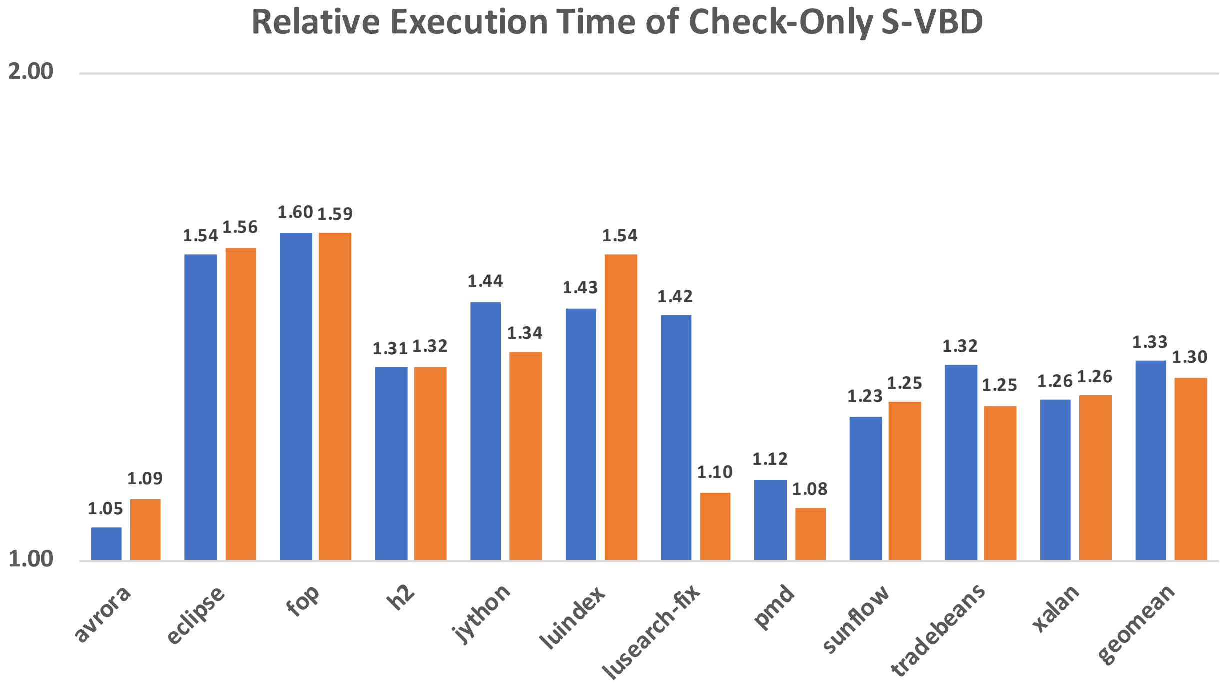


Figure 4.7: Relative execution time of check-only S-VBD over baseline JVM on machine A and machine B, y-axis in logarithmic scale.

execution time on Machine A and Machine B is 2.01 and 1.86 respectively, representing a 101% and 86% overhead over the baseline HotSpot JVM. Comparing these results to the ones for VBDA-HotSpot from Figure 3.24 I see that my speculative compilation strategy provides little speedup for these benchmarks. I 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.

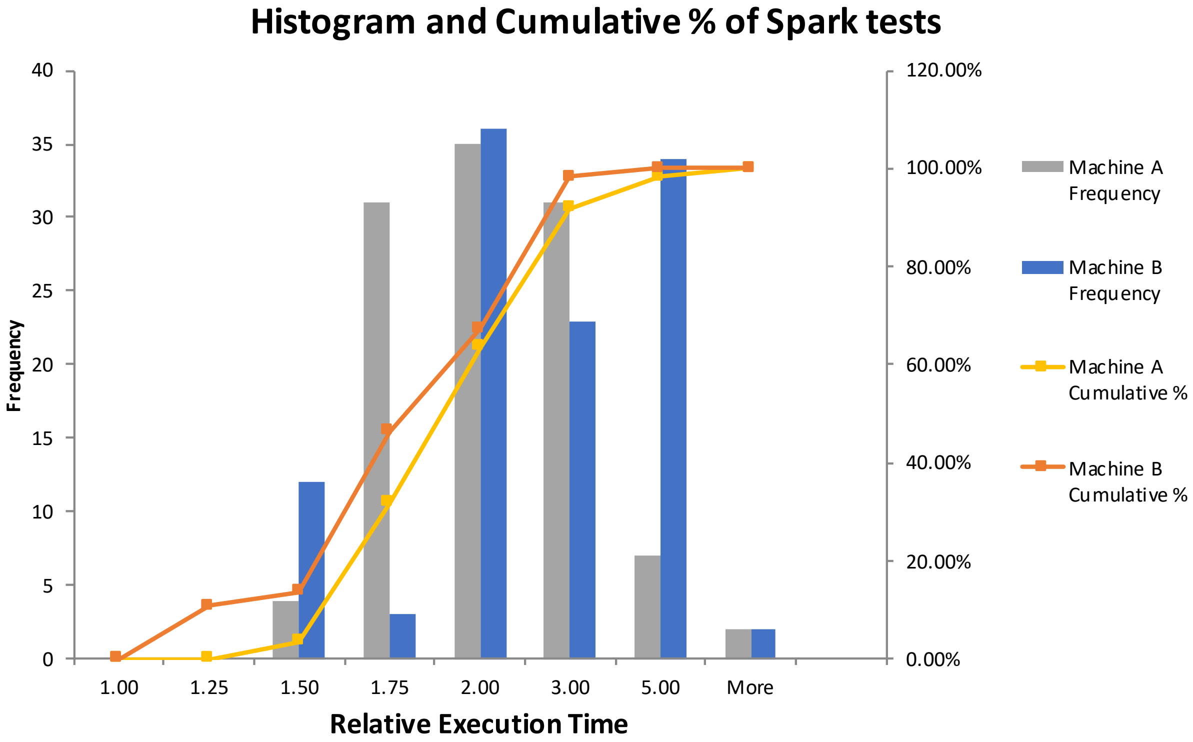


Figure 4.8: Histogram and cumulative % of relative execution time of VBDA-HotSpot for Spark benchmarks.

CHAPTER 5

A Sequential Consistency Enforcing Julia Compiler

Another language I have looked into is the Julia language. Julia aims to be a fast, dynamic, and easy to use programming language. Thanks to its high-level syntax, script-like characteristic (because it's dynamically typed), and reasonably fast performance, Julia has gained much popularity in fields like scientific computation.

The Julia language does not have a memory model for its recently added multi-threading support [BNP]. Julia uses LLVM as a backend for the compiler and does not put additional work into enforcing a stronger memory model. Based on the fact that the LLVM memory model is inspired by the C++0x memory model [LLVb], it is reasonable to conclude that the Julia's memory model is weaker than SC. In fact, simple litmus tests like the Dekker's algorithm can show the SC-violating behaviors in Julia.

However, like Java, I believe Julia's design philosophy suggests it a good candidate to have SC: Julia is designed to be an accessible and easy-to-use language for programmers from any background. It provides many safety guarantees by default, while also allows more performance-focus users to use various macros to "opt-out" of the safety checks. For example, Julia uses bounds checking to ensure memory safety when access arrays, but `@inbounds` macro can be used to elide such checks in a tight loop.

Part of the reason behind Julia's decision (or non-decision) of its memory model is the lack of measurement for the cost of it having SC. While previous work suggest that the cost of SC for Java can be lower than people have expected, it still remains to be seen whether the differences in the programs workloads (array-heavy for Julia) or the fact that Julia is implemented using LLVM will change the cost of SC on it.

To answer those questions, I have explored ways to provide SC for Julia. I have implemented an SC enforcing Julia compiler that guarantees SC and measured the cost of SC for Julia. I have further analyzed different factors that might affect the cost of SC and explored possibilities of using annotations to reduce the cost of SC.

5.1 Design Overview

To implement an SC enforcing Julia compiler, the general idea is to make sure all loads and stores to shared memory in the program have SC semantic. I will give an overview of the design below and talk about the details of implementation in the next section.

To implement the above idea, I need to transform all loads and stores in the LLVM IR to have SC semantics. I must first decide when to transform the loads and stores in the IR. There are two choices: 1) I can perform such transformation when the Julia compiler is generating the LLVM IR from its lowered form IR; or 2) I can implement such transformation as an LLVM pass, where I will iterate through all loads and stores in every function and rewrite them.

The benefit of the first approach is that I may be able to take advantage of some semantics in the Julia AST for optimization. However, this approach is more intrusive to the existing code base. Implementing the transformation at such an early stage also means I won't have accesses to the many existing LLVM analysis that will be available later, many of which will be very helpful for more complicated optimizations. The second approach requires much less change to the Julia compiler itself, which as suggested by the Julia developers is much preferred. Besides, Julia can still preserve some of its semantics information as metadata in the LLVM IR, and I can still take advantage of Julia-level semantic information even if I only modify LLVM. Therefore I decide to implement my changes as an LLVM pass.

I have implemented a transformation LLVM pass which rewrites loads and stores in the program to have SC semantics using combinations of LLVM's atomic instructions and fence instructions. However, simply rewriting every load and store in the LLVM IR to have SC semantics will lead to huge slowdown. Therefore, I have also implemented two optimizations to reduce the number of load and store instructions I need to rewrite, which I will describe more in detail in 5.2.4.

To follow the "safe-by-default, performance by choice" principle of safe languages, I want to provide experienced programmers "escape hatches" to annotate certain parts of the code that is more performance intensive. I implemented a `@drf` annotation that allows certain loops/functions/modules to "escape" from the SC semantics and get the original behaviors of the Julia language.

The implementation of my SC-by-Default Julia compiler is open sourced and can be viewed on GitHub: https://github.com/Lun-Liu/SC-by-Default-Julia.

5.2 Implementation

My implementation is based on the v1.4.1 release of the Julia runtime on GitHub. Additionally, I have changed the LLVM version used to 9.0.1 because of a patch needed to handle atomic operations for non-zero address spaces correctly.

In the rest of this section, I will describe in detail how the above designs are implemented. I will first overview the LLVM pass that rewrites loads and stores to have SC semantics. Then I will go into details about the LLVM instructions I use to rewrite loads and stores to make them SC. I will also cover the caveats of the current implementation. Then I will go over a few optimizations I implement to reduce the performance cost as well as the `@drf` annotation available for programmers as the "escape hatch" for performance.

5.2.1 `AddSC` LLVM Pass

I implement my changes as an LLVM Pass, and I call it "`AddSC`". This pass is the first optimization pass in the passes pipeline specified by the Julia JIT compiler. It needs to be the first pass so the following passes will respect the SC semantics when trying to optimize the IR.

`AddSC` inherits from *FunctionPass* in LLVM. LLVM pass manager will iterate through every `Function &F` and call `AddSC`'s `runOnFunction` method with `F`. In `runOnFunction`, I iterate through every instruction in every basic block and rewrite the load and store instructions that need to be made SC using atomic instructions/fence instructions. I will cover the details on what specific instructions I used in the next section. Some of the load/store instructions can be skipped because of my optimizations or the `@drf` annotation as explained in later sections too.

Every LLVM pass also needs to specify which analysis it depends on and if it changes the IR, which analysis it can preserve. `AddSC` *requires* `LoopInfoWrapperPass` to find loops with annotations. Since it does not change the loop or the CFG in any way, it can also preserve the `LoopInfoWrapperPass` and the CFG.

5.2.2 Rewrite Loads and Stores using Atomic and Fence Instructions

In LLVM, loads and stores can have different levels of *atomic ordering* [LLVa], which provide different atomicity and ordering guarantees. By default, LLVM loads and stores are not atomic: if there is a race on a memory location, memory accesses will have undefined behaviors. Many load and store instructions that the Julia compiler currently generates are regular loads and stores and are not atomic.

One can also declare loads and stores to have a higher level of atomicity. Among all atomicities, *Acquire* makes a load to have lock-acquire semantics, *Release* makes a store to have lock-release semantics, *SequentiallyConsistent* provides *Acquire* semantics for loads and *Release semantics* for stores. It also guarantees a total ordering among all *SequentiallyConsistent* operations.

I implement the SC enforcing Julia compiler by rewriting every load and store in LLVM IR to be *SequentiallyConsistent* - simply making all loads *Acquire* and all store *Release* would not be sufficient since a global ordering is required for SC. However, there are also limitations to which loads and stores can be *atomic operation*s in the IR [LLVb]. For example, the pointee of atomic loads and stores must be either an integer, a pointer, or a floating-point, while regular loads and stores are allowed to point to aggregate types such as *ArrayType* or *StructType*. *CodeGen* process will fail if those loads and stores have high atomicity levels and are marked as atomic operations.

For those loads and stores, I still mark them as SequentiallyConsistent during `AddSC` pass, but I rewrite them again before codegen to instructions that are supported. Specifically, I implemented another LLVM pass called `SCExpand`. `SCExpand` is the last optimization pass in the pipeline before codegen. During `SCExpand`, every such load or store's atomicity level will be set back to the same as the regular loads and stores, to comply with LLVM IR's requirements. I then make a few additional changes to the IR to make sure that those instructions won't be reordered by hardware optimizations or compiler optimizations after this pass.

To prevent the hardware from reordering memory instructions, during `SCExpand`, I add an *Acquire* fence after a load, a *Release* fence before a store, and a *SequentiallyConsistent* `FenceInstruction` after every such store. The `FenceInstruction`s in LLVM are similar to the `MemBar` nodes in the Java Ideal Graph. The `Acquire` fence provides a barrier of lock-acquire semantics, the `Release` fence provides a barrier of lock-release semantics. The `SequentiallyConsistent` fence guarantees that a total ordering exists between all `SequentiallyConsistent` operations. Those FenceInstructions will be compiled to corresponding hardware barriers in the compiled code: on X86, `Acquire` and `Release` fence will become no-op, and `SequentiallyConsistent` fence will become a `MFENCE` instruction to prevents loads to be reordered before stores, which is the only kind of reorderings X86 will perform.

Formally, according to the specification of LLVM [LLVb], the compiler only has to respect the semantics of those fence instructions if they are used with loads and stores that have at least *Monotonic* atomicity levels, which is also higher than the atomicity level of regular loads and stores. So simply surrounding regular loads and stores with `FenceInstructions` won't always prevent the compiler from performing optimizations (e.g. during codegen) that might violate SC.

Therefore, during `SCExpand`, I also set the `volatile` attribute of each such load or store instruction to `true`. A `volatile` load or store in LLVM is not the same as a `volatile` load or store in Java, but has a similar semantics to the `volatile` keyword in C/C++. Essentially it prevents any compiler optimization, including the ones during codegen, on the memory access. While it may be an overkill, it guarantees that no SC violation because of compiler optimizations will happen on those accesses after we set the atomicity level back to regular loads and stores.

Notice that marking the loads and stores as `volatile` and adding FenceInstructions may prevent more compiler optimizations than needed for SC. An alternative solution here to handle those loads and stores would be to rewrite the Julia compiler to only generate LLVM loads and stores that is compatible with higher atomicity levels. For example, in the place where they want to load a *struct*, they should manually load each field of the struct using separate load instructions. However, this also requires a lot of modification to the Julia compiler.

5.2.3 Caveats

I believe my implementation is a good first step to understand the cost of SC for Julia on X86. However, there does exist certain caveats of my current implementation.

The first caveat is that I only modified the Julia compiler, but not the interpreter. Although most code will be compiled since Julia don't have very complicated JIT heuristics, I don't enforce additional guarantees than what is already there for the memory accesses from the interpreter.

Second, to have end-to-end SC, it is also necessary to make sure no Julia optimizations before my SC transformation pass will violate SC. I believe this is better done in collaboration with the Julia developers considering the fast-changing nature of the Julia language and its implementation.

5.2.4 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 the transformation pass can ignore them.

If the data being loaded is immutable, it is already guaranteed to have SC semantics. A store to an immutable data (i.e. initialization of immutable data) does not need SC rewriting, but it does need to have Release semantics so that so that it will be guaranteed to order before the write of the pointer to this data, and preventing anyone from observing uninitialized value.

If the data is only visible to the runtime, it won't affect the semantics of the language and don't need additional guarantees. 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 union types of small, primitive "bits" types 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 I don't need to rewrite those loads and stores.

If the data is on the stack, it is already guaranteed to have SC semantics. Although Julia does perform optimization to stack-allocate objects, it will only do so if it can prove that the data is immutable or does not escape. Theoretically, there are certain tricks to get a pointer to object on the stack. However, it is not considered "safe" operation in Julia and such operations do not play well with GC either.

Motivated by this, I implement an optimization that uses the type information for LLVM's *type-based alias analysis (tbaa)* and only rewrite loads and stores of certain types. 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 of the instruction. I went through all tbaa types that Julia uses and identified the types that are "safe" for SC and the ones that are not always "safe" following the above principles. During my transformation pass, for each candidate load or store instruction, I will check the tbaa information and do not rewrite the ones that are accessing "safe" locations. Table 5.1 lists all the tbaas used by 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	No
jtbaa_stack	stack slot	No
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	No
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	No
jtbaa_array	Runtime data structure of a Julia array (jl_array_t)	No
jtbaa_arrayptr	The pointer inside a jl_array_t	No
jtbaa_arraysize	A size in a jl_array_t	No
jtbaa_arraylen	The len in a jl_array_t	No
jtbaa_arrayflags	The flags in a jl_array_t	No
jtbaa_arrayoffset	The offset in a jl_array_t	No
jtbaa_arrayselbyte	a selector byte in a isbits Union jl_array_t	No
jtbaa_const	Memory that is immutable by the time LLVM can see it	No

Table 5.1: A list of tbaas in Julia and if they need SC rewriting.

Some of the functions in Julia is implemented as builtins in C++. While the implementations in C++ are also compiled by LLVM, loads and stores from such C++ builtins do not have Julia specific tbaa information. Instead of treating all of them as "unsafe" for SC, I manually inspected the C++ source code of such builtin functions. I rewrite loads and stores to shared memory using atomic accesses and treat other loads and stores as "safe" for SC.

The above optimization allows me not to rewrite many load and store instructions. However, the tbaa information sometimes is too conservative. To further optimize the SC enforcing compiler, I implement an additional optimization to identify loads and stores to stack locations. In Java, this is easy to tell because one can tell if an operation is accessing a local variable (e.g. `iload_x`) or a heap location (e.g.`getfield`) by looking at the bytecode. In the LLVM IR, however, the operand of loads and stores can be a heap location or a local variable on the stack, and it is not immediate clear if a random load or store instruction is operating on a heap or stack location.

Luckily, the Julia compiler uses the custom address spaces feature to identify GC-tracked pointers and their uses, and I can use this information to help me decide if pointer can point to heap locations. LLVM allows each pointer to have an integer representing the *address space*. Specifically, in the Julia compiler, *GC tracked pointers* have address spaces of 10, *derived pointers* have address space of 11, *callee rooted pointers* have address space of 12, and *pointers loaded from tracked object* have address space of 13 [Jul]. On the other hand, the `alloca` instruction in LLVM, which allocates memory on the stack frame, always allocate the object in the address space for allocas: address space 0. Therefore, as an optimization, I also checked the address space of the pointer of candidate load and store instructions and if they are in address space 0, I won't rewrite them.

5.2.5 `@drf` annotations

Sometimes experienced programmers may want to allow certain performance intensive parts of their programs to have a weaker semantic but better performance. For example, considering the `fill!(A, x)` function in Julia. It fills array `A` with the value `x`. The body of this function is very straight-forward - a loop that writes every element in the array. Without SC, there is no atomicity or ordering constraints, the whole loop will be optimized into a `memset` call. With SC, the existence of the atomic instructions/fence instructions will prevent such optimization. As a result, the SC version can be hundreds times slower depends on the loop iterations.

In the case where the programmers want to elect for performance, I implement a `@drf` annotation that allows certain parts of the code to "escape" from the SC semantics and get the original behaviors of the Julia language. Of course, now the programmer is responsible to clearly specify the exact guarantee their program provides.

5.2.5.1 **`@drf`** annotation for loops

Julia already provides a `@simd` macro for performance reasons. If a loop is annotated as `@simd`, it promises that the iterations are independent and may be reordered, and therefore allows the compiler to optimize to use *SIMD* instructions for this loop. The implementation of `@drf` annotation mimics the implementation of `@simd` annotation. If a loop is annotated `@drf`, an `Expr(:loopinfo, Symbol("julia.drfloop"))` is added to its end. When parsing this expression, this expression is translated to a `CallInst` that calls the special function `julia.loopinfo_marker`, and this `CallInst` is also marked to have `'julia.loopinfo' MetaData`, whose value is `julia.drfloop`.

During `AddSC` pass for function `F`, I will first add all basic blocks to a worklist. After adding all basic blocks of `F` to a worklist, I will iterate through all *users* of the special function `julia.loopinfo_marker`, which are all the `CallInst`s to it. If the `CallInst` is in `F`, I will check to see if it has a `'julia.loopinfo' MetaData` and its value matches `'julia.drfloop'`. If it does, it means I can remove all basic blocks of this loop from the worklist and I won't rewrite any loads and stores in this loop.

Notice that `julia.loopinfo_marker` is not a real function, but rather a marker to find loops with annotations. Therefore, those `CallInst`s can't be in the IR when generating the actual code. However, Julia's `LowerSIMDLoop` pass is already handling the removing of such `CallInst`s.

5.2.5.2 **`@drf`** annotations for functions

Besides annotating certain loops as `@drf`, I also allow programmers to specify certain functions in the program to have `@drf` semantics, i.e. loads and stores in the function body will not have SC semantics but will have the original Julia semantics.

The names of the functions to be treated as `@drf` will be passed in as a list. During `AddSC` pass, before I make any changes to the IR, I will get the function name from LLVM, demangle it, and compare it with the list of `@drf` functions, and I will only make changes to the IR if the current function is not a `@drf` function.

5.2.5.3 `@drf` annotations for modules

Similar to functions, programmers can specify certain Julia modules to be `@drf`. If a Julia module is `@drf`, all functions in the module will be treated as `@drf`.

The names of the modules to be treated as `@drf` are also passed in as a list, and I compare the name of the Julia module that the function belongs to to the list during `AddSC` pass to decide if I need to rewrite the current function. However, the Julia module name of a function is not readily available in the LLVM IR. To solve this problem, when generating the LLVM IR from Julia AST, I add the Julia module name of a function to be a MetaData of the LLVM `Function`. I can then retrieve the module name by checking the MetaData of the function during `AddSC` pass.

5.3 Experiments

I performed several performance evaluations for my implementation of SC Julia on X86. I ran all the evaluations on a 8-core (4 physical cores with hyper-threading enabled) Intel(R) Core(TM) i7-6700 machine. I compared the performance of my implementation (referred to as *SC-Julia* in this section) to that of the original v1.4.1 Julia but switched its LLVM version to v9.0.1.

5.3.1 BaseBenchmarks

The BaseBenchmarks benchmark suite is a collection of Julia benchmarks available for CI tracking. It is open-sourced on GitHub, and used by the Julia developers to track the performance of the Julia language. There are more than 3000 benchmarks in the benchmark suite categorized into different groups. Many of the benchmarks in BaseBenchmarks are micro-benchmarks, 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. I calculate the overhead of the tests using the median execution time reported for each test.

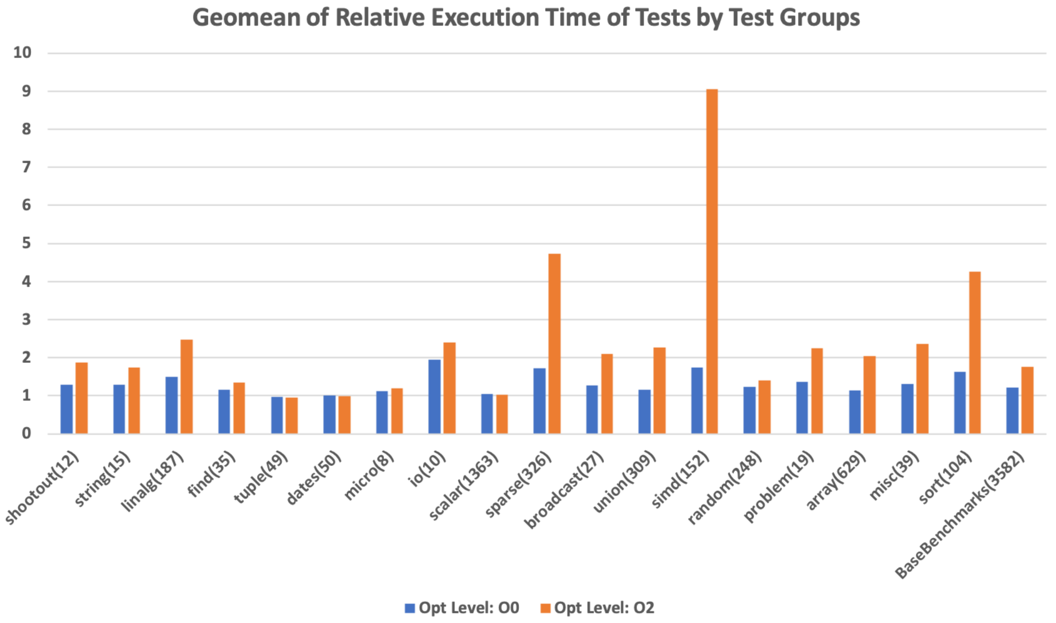


Figure 5.1: Geometric mean of relative execution time of SC-Julia over the baseline Julia. Each bar represents a test group in BaseBenchmarks, the number in the parenthesis after each test group name is the number of tests in that group. The last bar represents the whole benchmark suite.

I compute the relative execution time of SC-Julia over the baseline Julia for each test, and then calculate the geometric mean of relative execution time for the tests in each test group as well as the whole benchmark suite.

Figure 5.1 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 the parenthesis are the numbers of tests in the test groups. Figure 5.1 also shows the results of two different test settings. The orange bars on the right show the results when 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 running with O0 optimization level for the Julia compiler. Note that this optimization level is not the optimization level of the X86 compiler when the Julia compiler itself is built, but rather how many optimizations are enabled when the Julia compiler is compiling the base libraries or the input program.

From the figure I 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 25642%. 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 26154%. I conclude that with the current implementation, SC has a more acceptable overhead with a low optimization level setting for Julia.

While it may be consistent with people's intuition that SC would have a higher cost for a high optimization level, I perform a few other experiments in 5.3.2 to better understand why.

5.3.2 Understanding the Impact of SC: Compiler Optimizations vs Hardware Fences

There are in fact two factors that could explain the difference in the cost of SC for different optimizations levels. First, the baseline execution of the same program with O0 is much slower than the baseline execution with O2, therefore, even with the same amount of hardware fences, i.e., assuming the same additional absolute execution time coming from the hardware fences, with O2 SC-Julia is going to have a higher overhead. Second, O2 enables more optimizations passes, and those passes are also more aggressive when optimizing. One may assume that having additional atomic instructions/fence instructions in the IR will more adversely impact the compiler optimizations with O2.

To understand what contributes the most to the cost differences of SC with different optimization level, I modified LLVM to have a custom LLVM that all fence instructions in the IR will turn into no-op during code generation. For this experiment alone, I also modified the implementation of SC-Julia so that all atomic loads and stores will become `volatile` with FenceInstructions at the beginning of CodeGen. Comparing the execution time of SC

Julia using this LLVM with the execution time of baseline Julia using this LLVM, I will be able to see SC's impact on compiler optimizations for Julia.

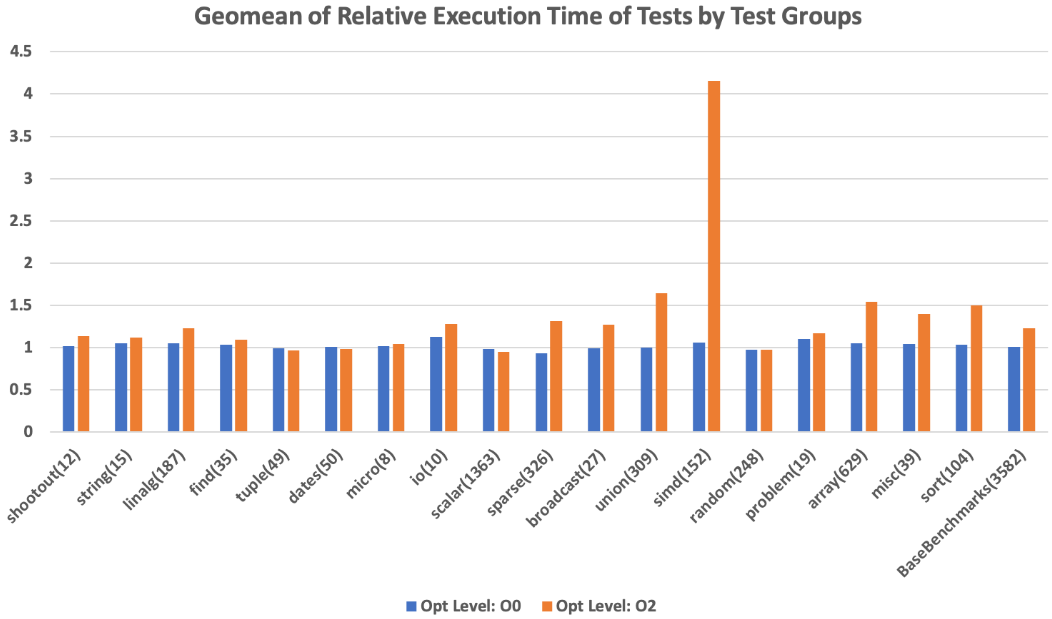


Figure 5.2: Geometric mean of relative execution time of SC-Julia without hardware fences over the baseline Julia without hardware fences. Each bar represents a test group in BaseBenchmarks, the number in the parenthesis after each test group name is the number of tests in that group. The last bar represents the whole benchmark suite.

Figure 5.2 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 1633%. For O2, the cost for all test groups are also much lower than SC-Julia. The average cost with O2 is 23%, and the maximum overhead of any single test is 6746%. Comparing it with the results in 5.3.1, I conclude that most of the cost of SC comes from the cost of the hardware fences, and that the main reason for the higher relative cost of SC for a high optimization level is that fences are relatively more expensive with a faster baseline.

```julia
@noinline function fill!(dest::Array{T}, x) where T
    xT = convert(T, x)
    @drf for i in eachindex(dest)
        @inbounds dest[i] = xT
    end
    return dest
end
```

Figure 5.3: Implementation of the library function `fill!` in Julia and with `@drf` annotation.

5.3.3 `@drf` annotations

While the cost of SC for most tests with a low optimization level is somewhat acceptable, there are certain tests (mostly microbenchmarks) in the benchmark suite that has a very high overhead. For example, `["array", "bool", "boolarray_true_fill!"]` and `["array", "bool", "bitarray_true_fill!"]` benchmark the performance of calling `fill!` to set all elements in a 1000000 elements bool array and 1000000 elements bit array to `true` respectively, and have overheads of 256.42X and 41.24X respectively.

Figure 5.3 illustrates how `@drf` annotations can help. The majority of the overhead 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`. Adding `@drf` annotation to the loop body will tell `AddSC` pass not to rewrite any load and store in the loop body, therefore allowing the `memset` optimization. Accordingly, now the `fill!` function is only guaranteed to work correctly if there are no data races on the array.

To gauge the potential effectiveness of `@drf` annotations, I measured the performance of SC-Julia with a few `@drf` annotations. First, I treat loops that already have `@simd` annotations as `@drf` loops. `@simd` in Julia promises that the iterations are independent and may be reordered, making them great candidate for `@drf` annotation. Additionally, I mark `Base` module in Julia as `@drf`. `Base` is a standard module in Julia and contains standard library functions of Julia. The distinction between the general user code and the library code in Julia makes standard library functions good candidates for `@drf` too.

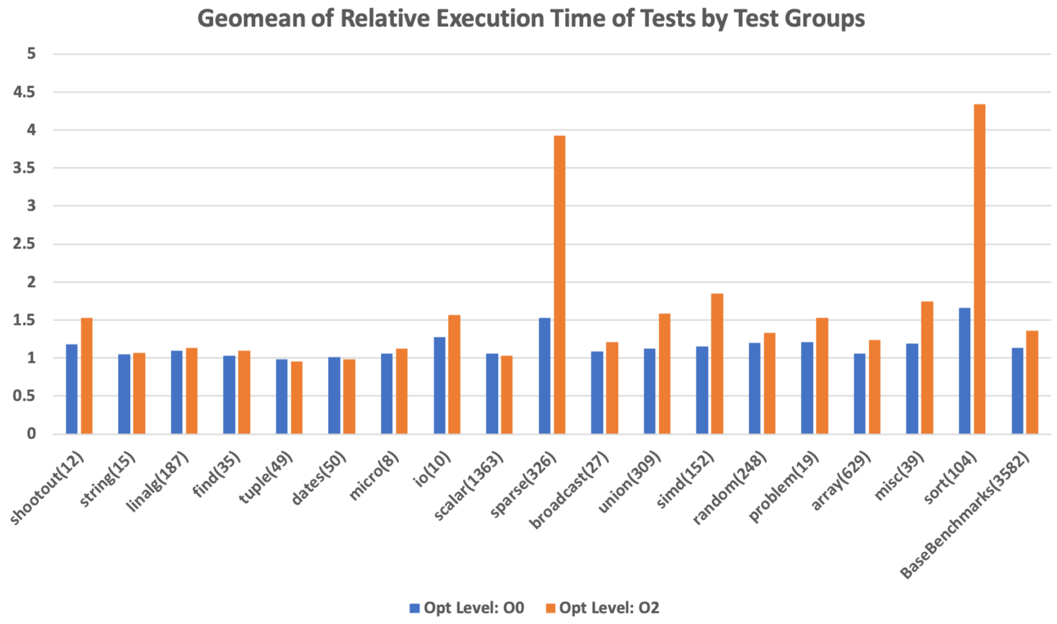


Figure 5.4: Geometric mean of relative execution time of SC-Julia over the baseline Julia for BaseBenchmarks, with `@drf` behaviors for `@simd` and `Base` module. O0 and O2 represents different optimization levels.

Figure 5.4 shows the average overhead of SC-Julia over the baseline Julia with O0 and O2 with `@drf` behaviors for `@simd` annotations and `Base` module. From the figure we can see that with `@drf` annotations for `@simd` and `Base` module, the average overhead for BaseBenchmarks with O0 is reduced from 22% to 13%, and with O2 it is reduced from 76% to 36%.

5.3.4 Hardware Instruction Selection

Another interesting finding I discovered is that the hardware barrier instruction used could affect the cost of SC greatly on X86. On X86, 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 a `XCHG` instruction or a sequence of `MOV; MFENCE` on X86.

A `XCHG` instruction is supposed to be faster than the sequence of `MOV; MFENCE` on X86. In LLVM, if a store has SequentiallyConsistent atomicity level and its size and alignment is supported, it will be translated into a `XCHG` instruction. A normal store is mapped to a `MOV` instruction, and a SequentiallyConsistent fence is mapped to a `MFENCE` instruction. To

```
using BenchmarkTools

arr = zeros(Int32, 100000)
function perf_loop(arr)
    for i = 1:100000
        arr[i] = i
    end
end

@btime perf_loop(arr)
```

Figure 5.5: A microbenchmark that uses a loop to set every element in an array.

measure the impact of different hardware barriers have on the cost of SC, I implemented another version of SC-Julia, and call it mSC-Julia. In this alternate implementation, instead of rewriting loads and stores to have SequentiallyConsistent atomicity level when supported, I 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`.

I first tested the performance with a microbenchmark that uses a loop to set every element in an array, as shown in 5.5. The execution time is 52.795 μs using the baseline Julia, 511.825 μs using SC-Julia, and 1300 μs using mSC-Julia. As shown by the results, switching to a slower `MFENCE` instruction can cause SC to have approximately twice as much overhead as the `XCHG` implementation for a tight loop.

Figure 5.6 shows the performance overhead of mSC-Julia over the baseline Julia for BaseBenchmark tests. The results are organized the same way as previous figures, x-axis shows different test groups, and y-axis shows the average relative execution time of tests in each test group. From the graph we can see this alternate implementation using `MFENCE` instruction has an average overhead of 50% for O0 and 130% for O2, both much higher than the cost of SC-Julia.

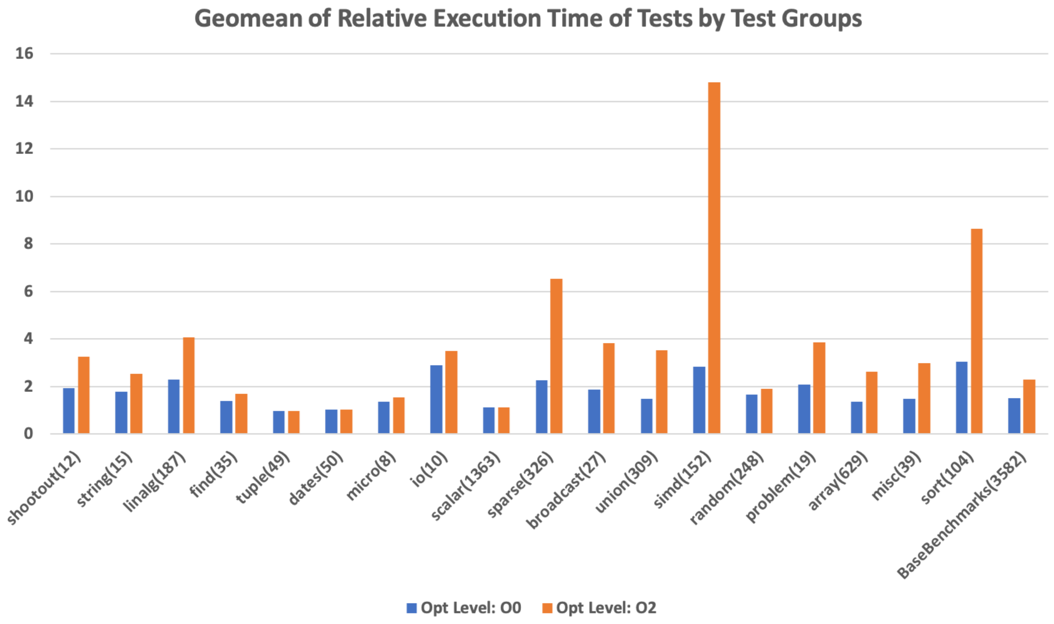


Figure 5.6: Geometric mean of relative execution time of mSC-Julia over the baseline Julia. O0 and O2 represents different optimization levels.

From the experiments above, I can see that the performance of hardware fences could have a great impact on the cost of SC, and future hardware optimizations for more efficient implementation of barrier instructions would also help bring down the cost of SC.

CHAPTER 6

Related Work

Language-Level Sequential Consistency I proposed the `volatile`-by-default semantics for achieving SC for Java and Julia and evaluated it on Intel X86 hardware and the ARM hardware. My work is, the first implementation of SC for Java in the context of a production JVM and hence the first realistic performance evaluation of the cost of SC for Java. The work to measure the cost of SC on ARM is, to the best of my knowledge, the first comprehensive study of the cost of providing SC for any language on ARM, which is a much weaker memory model than X86. The SC Julia compiler work is, to the best of my knowledge, the first implementation of SC for Julia.

Vollmer et al. [VSM17] implement the SC semantics for the Haskell programming language and demonstrate negligible overheads on X86. They also demonstrated low overheads for some benchmarks on ARM but did not do an extensive study due to the limited portability of Haskell libraries. The key takeaway is that a pure, functional programming language like Haskell naturally limits conflicting memory accesses among threads and so can support SC with low overhead. As such, these results do not extend to imperative languages like Java.

I evaluated the `volatile`-by-default semantics in a production JVM with modern features such as dynamic class loading and just-in-time compilation. In contrast, prior work has evaluated the cost of SC for Java in the context of an offline whole-program compiler, which admits more opportunities for optimization but is incompatible with modern JVMs. Shasha and Snir [SS88] propose a whole-program delay-set analysis for determining the barriers required to guarantee SC for a given program. Sura et al. [SFW05] implement this technique for Java and Kamil et al. [KSY05] do the same for a parallel variant of Java called Titanium. These works demonstrate low performance overhead for SC on both X86 and POWER.

Alglave et al. [AKN14] implemented SC for C programs similarly.

Other work has achieved language-level SC guarantees for Java [CTM07, AQL09] and for C [MSM11, SNM12] 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. [SNM12] is similar to my 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 I speculate purely at the JVM level. Finally, several works demonstrate testing techniques to identify errors in Java and C code that can cause non-SC behavior (e.g., [FF10, IM16]).

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 [SBZ15, SCB15, BZB15, ZBB17]. These approaches are implemented in the Jikes research virtual machine [AAB05] and evaluated only on X86. Work on region serializability for C has achieved good performance either through special-purpose hardware [LCS10, MSM10, SMN11] or by requiring $2N$ cores to execute an application with N threads [OCF13].

Memory Model Safety The notion of "safety" in the JMM disallows out-of-thin-air values [MPA05], but it has proven difficult to ensure while also admitting desired optimizations [BMN15]. Several recent works have defined new memory models that attempt to resolve this tension [BD14, JR16, PS16, LVK17, KHL17, OD18]. 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 my knowledge only the work by Ou and Demsky provides an empirical evaluation [OD18]; they demonstrate low overheads for C/C++ programs running on ARM hardware. My work adopts and empirically evaluates a significantly stronger notion of safety for Java than these works [MMM15], as it additionally preserves the program order of instructions and the atomicity of primitive types.

Weak Memory Model Performance for Java Demange et al. [DLZ13] define an X86-like memory model for Java. They present a performance evaluation that uses the Fiji real-time virtual machine [PZB10] to translate Java code to C, which is then compiled with a modified version of the LLVM C compiler [MSM11] and executed on X86 hardware. Ritson and Owens [RO16] modified the HotSpot compiler's code-generation phase for both ARM and POWER to measure the cost of different instruction sequences to implement the JMM.

CHAPTER 7

Conclusion

Safe languages like Java and Julia provide programming abstractions, like type and memory safety, to improve programmer productivity. Unfortunately, in the domain of concurrent programming, mainstream safe languages instead choose to adopt complex and error-prone semantics by default in the name of performance, breaking fundamental programming abstractions and failing to provide baseline guarantees for programmers.

My PhD work focuses on the problems with memory consistency models. Most mainstream safe languages adopt weak memory models or no memory models in the name of performance, and expose programmers to the relaxed/undefined semantics of the memory models, which is complex, unintuitive, and can violate critical program invariants.

I believe this performance-by-default approach conflict with the design philosophy of "safe" languages. Additionally, very little is known about the actual performance cost of a stronger memory model on modern hardwares for modern programming languages.

My thesis states that it is possible to measure the cost of strong and simple semantics for concurrent programming on *modern* hardwares for *modern* languages with *modern* language implementation techniques, and with optimizations, it is possible to have a simpler semantics for concurrent programming at a reasonable cost for certain languages.

In this thesis I define the `volatile`-by-default semantics as a natural way to make the memory consistency model of Java and other JVM-based languages safe-by-default and performant-by-choice. The `volatile`-by-default semantics protects most programmers from the vagaries of relaxed-memory-model behavior by providing sequential consistency by default, while still allowing expert programmers to avoid fence overheads on performance-critical libraries. I present VBD-HotSpot, a modification of Oracle's HotSpot JVM that enforces the volatile-by-default semantics on Intel X86 hardware and ARM hardware. To my knowledge this is the first implementation of SC for Java in the context of a production JVM and hence the first realistic performance evaluation of the cost of SC for Java. This is also the first comprehensive study of the cost of providing language-level sequential consistency for a production compiler on ARM.

My experiments indicate that while VBD-HotSpot incurs a significant performance cost relative to the baseline HotSpot JVM for some programs, it can be a practical choice today for certain applications on X86. My experiments also show that an the same technique that provides the volatile-by-default semantics for Java on Intel X86 hardware at modest cost in fact incurs considerable overhead on ARM.

I then present a novel *speculative* technique to optimize language-level SC and demonstrate its effectiveness in reducing the overhead of enforcing volatile-by-default. To my knowledge this is the first optimization approach for language-level SC that is compatible with modern implementation technology, including dynamic class loading and just-in-time (JIT) compilation. My experiments show that with optimizations, I can reduce the cost of SC on weaker hardware architectures like ARM significantly.

I have also explored ways to provide SC for Julia. I have implemented an SC enforcing Julia compiler that guarantees SC and measured the cost of SC for Julia. I have further analyzed different factors that might affect the cost of SC and explored possibilities of using annotations to reduce the cost of SC.

Longer-term, the goal of this line of research is to persuade mainstream languages to adopt such safer and stronger memory models as the default. To reach this goal, it would help greatly if the performance of strong language-level memory models like volatile-by-default can be competitive for more use cases even on weak hardwares. Static techniques that can reduce the number of fences needed and compiler optimizations limited or dynamic techniques that have lower runtime cost would be obvious avenues of future work. Furthermore, a more carefully selected group of "unsafe" annotations could help us reach a low overhead threshold much faster. The selection of those annotations may require manual efforts from domain experts of certain languages, or can be simplified if future work implements tools that can automatically profile and recommend the best candidates for “unsafe” annotations.

Additionally, until the above longer term goal is achieved, a promising direction to build on the work in this thesis is to build tools that can help identify and debug memory model related bugs. Having the ability to execute a program with a safe-by-default memory model with specified “unsafe” regions, one can possibly use techniques such as delta debugging to help pinpoint the exact code that leads to the memory model related bug.

REFERENCES

[AAB05] 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. "The Jikes Research Virtual Machine project: Building an open-source research community." *IBM Systems Journal*, **44**(2):399–418, 2005.

[AB10] Sarita V. Adve and Hans-J. Boehm. "Memory Models: A Case for Rethinking Parallel Languages and Hardware." *Commun. ACM*, **53**(8):90–101, August 2010.

[AH90] S. V. Adve and M. D. Hill. "Weak ordering—a new definition." In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 2–14. ACM, 1990.

[AKN14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. "Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion." In *Computer Aided Verification - 26th International Conference*, pp. 508–524, 2014.

[AQL09] Wonsun Ahn, Shanxiang Qi, Jae-Woo Lee, Marios Nicolaides, Xing Fang, Josep Torrellas, David Wong, and Samuel Midkiff. "BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support." In *42nd International Symposium on Microarchitecture*, 2009.

[BA08] H. J. Boehm and S. Adve. "Foundations of the C++ concurrency memory model." In *Proc. of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 68–78. ACM, 2008.

[BBB17] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J. W. Maessen, J. D. Mitchell, K. Nilsen, B. Pugh, and E. S. Sirer. "The "Double-Checked Locking is Broken" Declaration.", Accessed April 2017.

[BD14] Hans-J. Boehm and Brian Demsky. "Outlawing Ghosts: Avoiding Out-of-thin-air Results." In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pp. 7:1–7:6. ACM, 2014.

[BGH06] 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. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis." In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, New York, NY, USA, October 2006. ACM Press.

[BMN15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. "The Problem of Programming Language Concurrency Semantics." In Jan Vitek, editor, *Programming Languages and Systems - 24th European*

Symposium on Programming, volume 9032 of *Lecture Notes in Computer Science*, pp. 283–307. Springer, 2015.

[BNP] Jeff Bezanson, Jameson Nash, and Kiran Pamnany. “Announcing composable multi-threaded parallelism in Julia.”.

[Boe11] Hans-J. Boehm. “How to Miscompile Programs with "Benign" Data Races.” In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar’11, Berkeley, CA, USA, 2011. USENIX Association.

[Boe12] Hans-J. Boehm. “Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil.” In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES ’12, pp. 9–14. ACM, 2012.

[BZB15] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. “Valor: Efficient, Software-only Region Conflict Exceptions.” In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pp. 241–259. ACM, 2015.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. “The Java Memory Model: Operationally, Denotationally, Axiomatically.” In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pp. 331–346. Springer, 2007.

[CTM07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. “BulkSC: Bulk enforcement of sequential consistency.” In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pp. 278–289, 2007.

[DLZ13] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. “Plan B: A Buffered Memory Model for Java.” In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’13, pp. 329–342, New York, NY, USA, 2013. ACM.

[FF10] Cormac Flanagan and Stephen N. Freund. “Adversarial Memory for Detecting Destructive Races.” In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’10, pp. 244–254. ACM, 2010.

[FGP16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. “Modelling the ARMv8 architecture, operationally: concurrency and ISA.” In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 608–621. ACM, 2016.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation.” In *Proceedings of the 22Nd Annual ACM SIGPLAN*

Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, pp. 57–76. ACM, 2007.

[IM16] Mohammad Majharul Islam and Abdullah Muzahid. “Detecting, Exposing, and Classifying Sequential Consistency Violations.” In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pp. 241–252. IEEE Computer Society, 2016.

[Jav17] Accessed July 2017, 2017.

[JR16] Alan Jeffrey and James Riely. “On Thin Air Reads Towards an Event Structures Model of Relaxed Memory.” In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pp. 759–767, New York, NY, USA, 2016. ACM.

[JSR18] “JSR-133 Cookbook for Compiler Writers.” Accessed April 2018, 2018.

[Jul] “Julia Address Space Representation.”.

[KDD17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris.” In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 17:1–17:29, 2017.

[KHL17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. “A Promising Semantics for Relaxed-memory Concurrency.” In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pp. 175–189. ACM, 2017.

[KSY05] A. Kamil, J. Su, and K. Yelick. “Making sequential consistency practical in Titanium.” In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005.

[LA04] C. Lattner and V. Adve. “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, 2004.

[Lam79] L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs.” *IEEE transactions on computers*, **100**(28):690–691, 1979.

[LCS10] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. “Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions.” In *Proc. of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[LLVa] “LLVM Atomic Instructions and Concurrency Guide: Atomic orderings.”.

[LLVb] "LLVM Language Reference Manual.".

[LMM17] Lun Liu, Todd Millstein, and Madanlal Musuvathi. "A Volatile-by-default JVM for Server Applications." *Proc. ACM Program. Lang.*, **1**(OOPSLA):49:1–49:25, October 2017.

[LVK17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing Sequential Consistency in C/C++11." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pp. 618–632, New York, NY, USA, 2017. ACM.

[MBY15] 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. "MLlib: Machine Learning in Apache Spark." *CoRR*, **abs/1505.06807**, 2015.

[MMM15] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. "The Silently Shifting Semicolon." In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 177–189, 2015.

[MMS12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. "An Axiomatic Memory Model for POWER Multiprocessors." In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference*, volume 7358, pp. 495–512. Springer, 2012.

[MPA05] J. Manson, W. Pugh, and S. Adve. "The Java memory model." In *Proceedings of POPL*, pp. 378–391. ACM, 2005.

[MPM15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. "Use at Your Own Risk: The Java Unsafe API in the Wild." In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pp. 695–710. ACM, 2015.

[MSM10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. "DRFx: A simple and efficient memory model for concurrent programming languages." In *PLDI '10*, pp. 351–362. ACM, 2010.

[MSM11] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. "A Case for an SC-Preserving Compiler." In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[OCF13] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. "...And Region Serializability for All." In Emery D. Berger and Kim M. Hazelwood, editors, *5th USENIX Workshop on Hot Topics in Parallelism, HotPar'13*. USENIX Association, 2013.

[OD18] Peizhao Ou and Brian Demsky. "Towards Understanding the Costs of Avoiding Out-of-thin-air Results." *Proc. ACM Program. Lang.*, **2**(OOPSLA):136:1–136:29, October 2018.

[Ope17] Accessed July 2017, 2017.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO." In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pp. 391–407. Springer, 2009.

[PS16] Jean Pichon-Pharabod and Peter Sewell. "A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions." In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 622–633, New York, NY, USA, 2016. ACM.

[PZB10] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. "High-level Programming of Embedded Hard Real-time Devices." In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 69–82, 2010.

[RO16] Carl G. Ritson and Scott Owens. "Benchmarking Weak Memory Models." In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pp. 24:1–24:11, 2016.

[SA08] Jaroslav Sevcík and David Aspinall. "On Validity of Program Transformations in the Java Memory Model." In *ECOOP*, pp. 27–51, 2008.

[SBZ15] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. "Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability." In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 561–575, 2015.

[SCB15] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. "Toward Efficient Strong Memory Model Support for the Java Platform via Hybrid Synchronization." In Ryan Stansifer and Andreas Krall, editors, *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015*, pp. 65–75. ACM, 2015.

[SFW05] Z. Sura, X. Fang, C.L. Wong, S.P. Midkiff, J. Lee, and D. Padua. "Compiler techniques for high performance sequentially consistent Java programs." In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 2–13, 2005.

[SH97] Douglas C. Schmidt and Tim Harrison. "Double-checked Locking: An optimization pattern for efficiently initializing and accessing thread-safe objects." In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pp. 363–375. Addison-Wesley Longman Publishing Co., Inc., 1997.

[SMN11] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. "Efficient processor support for DRFx, a memory model with exceptions." In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pp. 53–66. ACM, 2011.

[SNM12] Abhayendra Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. "End-to-end Sequential Consistency." In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pp. 524 –535, june 2012.

[SS88] D. Shasha and M. Snir. "Efficient and correct execution of parallel programs that share memory." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **10**(2):282–312, 1988.

[TSM17] "TSM03-J. Do not publish partially initialized objects." Accessed July 2017, 2017.

[VSM17] Michael Vollmer, Ryan G. Scott, Madanlal Musuvathi, and Ryan R. Newton. "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, pp. 283–298. ACM, 2017.

[ZBB17] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. "Avoiding Consistency Exceptions Under Strong Memory Models." In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pp. 115–127. ACM, 2017.

[ZXW16] 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. "Apache Spark: A unified engine for big data processing." *Commun. ACM*, **59**(11):56–65, 2016.