

UNIVERSITY OF PENNSYLVANIA
DEPT. OF COMPUTER AND INFORMATION SCIENCE
PHILADELPHIA, PENNSYLVANIA, USA

IN PARTIAL FULFILLMENT OF THE WPE-II REQUIREMENT

**Memory Models and Implementations
for Shared-memory Multi-threaded
Programming Languages**

Author: Jianzhou Zhao

Committee Chair: Milo M. K. Martin

Committee Member: Stephanie Weirich

Committee Member: Steve Zdancewic

March 3, 2011

Abstract

In shared-memory multi-threaded programming, threads execute in parallel, and directly access shared data structures by passing memory references among these threads. To correctly implement shared-memory multi-threaded programming language, we need to design a memory model that defines the set of values that a read in a program is allowed to return.

To achieve high performance, most of compiler and hardware re-order shared memory accesses except when re-ordering violates single-threaded data dependence. However, re-ordering shared memory accesses with data races may lead to non-intuitive behaviors, or violate safety and security. To increase productivity and simplify debugging, a language-level memory model for a concurrent programming language must adopt stronger semantics by restricting the behavior of data races or inhibiting problematic optimizations, but its effective and efficient implementation needs complicated program analysis and specific hardware. Therefore, it is challenging to define a memory model that is easy to implement, and also allows programmers to productively design concurrent programs with high performance.

This paper surveys different language-level memory models (which are sequential consistency, data-race-free and DRFx), and presents their implementation techniques (such as static analysis, speculation, dynamic conflict detection, and software-hardware co-design) with examples and explanations. We also summarize their trade-offs, the memory model design space and implementation challenges, and discuss future research directions for programming disciplines.

Contents

1	Introduction	3
1.1	The Need for Memory Models of Shared-memory Multi-threaded Programming Languages	3
1.2	Challenges of Defining and Implementing Memory Models	4
1.3	Overview	5
2	The Sequential Consistency Memory Model	6
2.1	Delay Set Analysis	6
2.2	Practical Delay Set Analysis in Java	8
2.2.1	The System	9
2.2.2	Discussion	12
2.3	Enforcing Sequential Consistency by Speculation	12
2.3.1	BulkCompiler	15
2.3.2	Discussion	17
3	The Data-Race-Free Memory Model	18
3.1	Reuse of Classic Compiler Optimizations across Synchronizations	19
3.2	Discussion	20
4	The DRFx Memory Model	21
4.1	DRFx	23
4.1.1	Sufficient Requirements	24
4.1.2	Implementation	24
4.2	Discussion	25
5	Discussion	26
5.1	The Design Space of Memory Models	26
5.2	Summary of Implementation Techniques	29
5.3	Programming Disciplines and Hardware Support	31
6	Conclusion	32

1 Introduction

1.1 The Need for Memory Models of Shared-memory Multi-threaded Programming Languages

Multiprocessors are now ubiquitous, with hardware support for concurrent computation over shared memory, and multi-threaded primitives from libraries and programming languages. In shared-memory multi-threaded programming, threads on different processors execute in parallel, and directly access shared data structures by passing memory references among these threads.

In this architecture expert programmers can design programs with high performance. First, if threads read shared data more frequently than they update the data, directly accessing the data through the main memory reduces the space overhead of data replication. Second, a program can carefully partition shared data with complex data structures such as trees and graphs into disjoint sub-components, and assign the subcomponents to different threads that do not need to interfere each other. Third, if different threads have to read and write shared data simultaneously, the program can correctly choose suitable synchronization mechanism that prevents data races but still allows compiler and hardware optimizations.

However, in general this architecture allows arbitrary programs with data races and non-deterministic executions that result in behaviors difficult to reproduce. A program has data races if more than one thread can access a memory location simultaneously, and at least one such access is a store. Moreover, this architecture does not precisely define what compiler and hardware optimizations over shared data are legal. Consider the code in Figure 1 and assume that the global variables X and Y shared by Thread 1 and Thread 2 are initialized to 0. If we reason about the program by interleaving thread executions, the original Thread 1 is always terminating, and never outputs 0. However, the transformed Thread 1, in which the redundant read $r = X$ is eliminated, leads to non-termination, and may output 0 because of reordering $X=1$ and $Y=1$ in the transformed Thread 2. These transformations introduce new behaviors, although they preserve the sequential semantics of each thread independently.

To reason about the program we need to precisely define a memory model of a shared-memory multi-threaded programming language that defines the set of values that a read in a program is allowed to return. A memory model is the basic semantics of a programming language that the implementation (such as compiler or hardware) of the language must preserve. Therefore, designing a memory model of programming languages for building large and complicated software systems such as databases and network servers must meet several demanding requirements:

- **Productivity:** Programmers must have the ability to understand an execution intuitively, but still use widespread programming practices: global space, imperative programming, complex pointer-based data structures, and object-oriented programming. Moreover, the memory model should

int X = 0; int Y = 0; // Initially	
Thread 1 (original)	Thread 2 (original)
<pre> int r = X; if (Y) while (!r) { r = X; print r; } </pre>	<pre> X = 1; Y = 1; </pre>
Thread 1 (transformed)	Thread 2 (transformed)
<pre> int r = X; if (Y) while (!r) print r; </pre>	<pre> Y = 1; X = 1; </pre>

Figure 1: Sequential-Consistency-breaking transformations for concurrent code.

inhibit or detect execution that causes unpredicted behaviors.

- **Performance:** The memory model should allow as many optimizations as possible, and should not prevent threads on different processors from executing in parallel.
- **Implementation:** A programming language needs an effective and efficient implementation that transparently transforms source programs into target programs, and executes target programs while preserving semantics of source programs.

1.2 Challenges of Defining and Implementing Memory Models

However, defining an appropriate memory model that satisfies the above requirements is challenging. Researchers proposed memory models that make different trade-offs.

The most intuitive model is *sequential consistency* (SC) [48] that requires that all memory operations of a program appear to execute in a total order, and that the operations of a single thread appear to execute in the order described by the program. SC arguably provides an easy programming interface to programmers because the execution of a program is an interleaving of executions from different threads. However, hardware manufacturers (such as Intel, IBM and AMD) have chosen to support weak memory models [4] in which optimizations only need to preserve single-threaded semantics, such as transformations in Figure 1. Compilers also perform similar optimizations. Although hardware provides memory fences to prevent re-ordering, conservatively inserting memory fences at compile time leads to noticeable performance overhead [83]. An

alternative solution is to implement sequential consistency with hardware that enforces SC, but requires non-trivial hardware changes [8].

To overcome the limitations of sequential consistency, researchers observed that most programs are well-synchronized, and proposed the *data-race-free* (DRF) model [34, 2]—in which programs without data races yield sequential consistency while other programs’ behaviors are undefined. For example, the semantics of the program in Figure 1 is undefined because there are no synchronization operations that prevent the data races of the global variables X and Y. In the DRF memory model, the compiler and hardware can apply any transformation that preserves sequential semantics without introducing more data races within synchronization regions, and even across synchronizations with siloed analysis [40]. However, the implementation still needs to enforce sequential consistency for synchronization operations.

One problem of DRF is that undefined behaviors for programs with data races may lead to safety problems [54, 17], and also complicate debugging. The DRFx memory model [56] precisely requires that a non-SC program execution due to a data race raises an exception, and programs without data races execute without exception and preserve sequential consistency. It is expensive to instrument source code and detect SC violations at runtime. Therefore, DRFx uses a specific hardware to check violations.

An alternative memory model that defines behaviors for data races is the Java memory model [54] that formally defines legal transformation results for programs with data races. However, the situation of the Java memory model is still far from satisfactory because of its inherent complexity and new observations that some optimizations that were intended to be allowed are in fact prohibited by the current specification [89].

1.3 Overview

We have motivated the requirements and challenges of memory models and implementations for shared-memory multi-threaded programming languages. The rest of the paper gives a survey of important works [83] [8] [40] [56] in this field, compares them in various aspects, and attempts to identify future research directions. Section 2 shows two implementations of the sequential consistency memory model: one [83] targets hardware with a weak memory model; the other [8] is based on hardware that enforces sequential consistency [22]. Section 3 explains how to reuse classical optimizations in the DRF memory model [40]. Section 4 considers a compiler approach for supporting the DRFx memory model over the hardware that detects sequential consistency violations [56]. Finally, Section 5 summarizes the design space of memory model and implementation techniques, and discusses possible future work. Section 6 concludes. We present some detailed discussions in the appendices.

2 The Sequential Consistency Memory Model

The most intuitive language-level memory model is sequential consistency [48] that requires memory operations from different threads to execute in a total order, and operations from each individual thread to appear in the order specified by the program.

int X = 0; int Y = 0; // Initially	
Thread 1	Thread 2
1.1) X = 1;	2.1) r1 = Y;
1.2) Y = 1;	2.2) r2 = X;

Figure 2: Re-ordering.

Sequential consistency is an intuitive memory model, but it imposes strong constraints on re-ordering, introducing and eliminating shared memory accesses within a thread that are safe for single-threaded programs. In Figure 2, single-threaded semantics allows any order of shared memory instructions, while the result $r1 = 1$ and $r2 = 0$ (by executing the sequence 2.2, 1.2, 2.1, 1.1) is not a sequentially consistent outcome.

Because re-ordering, insertion and deletion are primitive compiler transformations [43], sequential consistency may disallow many standard compiler optimizations for shared memory accesses, such as common subexpression elimination, dead code elimination, loop-invariant code motion, loop transformations, in-lining, *etc.* [59] On the other hand, most hardware manufacturers support weak consistency [4, 27], but enables most optimizations for regular data accesses that only preserve intra-thread data dependency [79]. To preserve sequential consistency for shared memory accesses, hardware provides *memory fences* to prevent re-ordering of shared memory accesses. However, protecting any potential shared accesses without further analysis can dramatically decrease performance.

In the next subsection we explain how to minimize performance overhead to preserve sequential consistency by preventing optimizations of hardware and compiler in an ideal setting [78]. Section 2.2 presents the compiler techniques for sequentially consistent Java programs over weak consistent hardware with a slowdown of 10% on average [83]. Then Section 2.3 presents BulkCompiler [8] that is based on sequentially consistent hardware, and outperforms weak consistent programs with 37% on average [22].

2.1 Delay Set Analysis

As discussed, both compiler optimizations and weak consistent hardware preserve the sequential semantics for each individual thread, while the result of an execution depends on the execution order E of shared memory accesses. In Figure 2, 1.1 and 2.2 are shared memory accesses of X, and the other pair of shared memory accesses is 1.2 and 2,1 for Y. Given an execution of the program, we

can observe the dependency of each pair of shared memory accesses. Consider the execution sequence

$$2.2)r2 = X, \quad 1.2)Y = 1, \quad 2.1)r1 = Y, \quad 1.1)X = 1$$

Here, the shared memory accesses of X have a write-after-read dependency, and the shared memory accesses of Y have a read-after-write dependency:

$$2.2)r2 = X <_E 1.1)X = 1; \quad 1.2)Y = 1 <_E 2.1)r1 = Y$$

The result of this sequence is $r1 = 1$ and $r2 = 0$ that follows the sequential semantics for each thread because the two statements in each thread are free to reorder without breaking intra-thread data dependency.

However, if the language-level memory model enforces that memory operations from each thread must appear to execute in the program order specified by the thread's code, the result is illegal. Intuitively the code of each thread imposes a program order P :

$$1.1)X = 1 <_P 1.2)Y = 1; \quad 2.1)r1 = Y <_P 2.2)r2 = X$$

The P and E in this example contain a cycle (1.1, 1.2, 2.1, 2.2, 1.1) (shown in Figure 3 (a)) that prevents *serializing* the memory operations in a total order without breaking data dependency imposed by the execution. So it is not sequentially consistent. Figure 3 (b) shows a sequentially consistent execution without cycles.

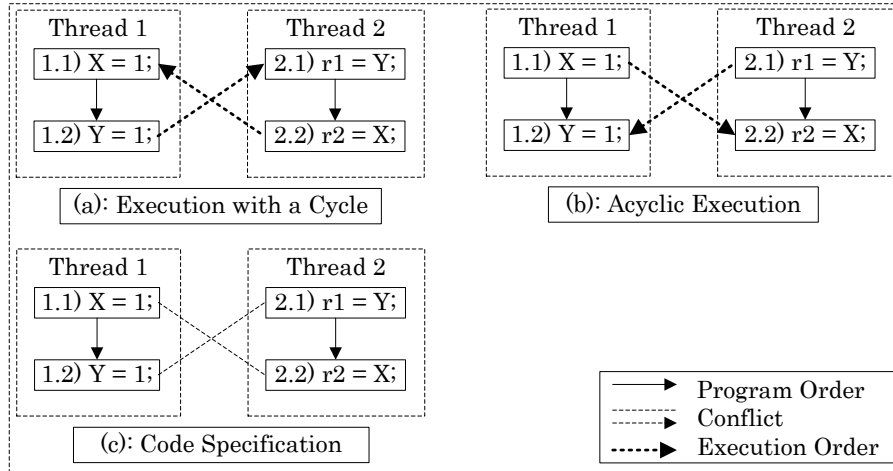


Figure 3: Specification and Executions

More precisely, we define a code by a tuple $\langle V, P, C \rangle$ that defines a graph where V is the set of variable accesses executed by the program; P is a set of *program edges* that represent the partial order on variable accesses V required by the program; C is a set of *conflict edges* that define the conflict relation on

accesses. Two accesses to the same variable *conflict* if they are from different threads, and at least one is a write. An execution order E is a proper orientation of the conflict relation C . An execution is sequentially consistent iff $E \cup P$ has no cycles. We observe that any cycle in $C \cup P$ can lead to a cycle in $E \cup P$. If we can enforce a delay for every pair of uPv that is an edge on a cycle of $C \cup P$, then it is sufficient to ensure its execution is sequentially consistent ($E \cup P$ has no cycles) (See Appendix A).

The SC compiler should inhibit any optimizations that may reorder a delayed pair of shared memory accesses, and should also insert costly *memory fences* to force hardware not to reorder them. Therefore, performance depends on the precision with which a compiler can determine the shared memory access orders that must be enforced. Shasha and Snir [78] show how to find the minimal delay set to guarantee sequential consistency. Their analysis is precise because they assume:

- straight-line code with no branching control flow,
- the target location in memory for each access is unambiguously known when the analysis is performed, and
- the number of threads in the program is known when the analysis is performed.

Krishnamurthy and Yelick [46] show that the precise delay set analysis is NP-complete. Appendix A explains how this precise delay set analysis is performed. The next section discusses how to perform delay set analysis that is approximate and fast, but precise enough to compile Java programs for sequential consistency.

2.2 Practical Delay Set Analysis in Java

In a general-purpose language, such as Java, the code specification $\langle V, P, C \rangle$ of a program cannot always be accurately constructed:

- A variable may contain a reference to different memory locations at different points in an execution. We need alias analysis to disambiguate memory locations. Also, an array access depends on its index value. At compile time neither alias analysis nor value analysis are precise. So a V node may represent accesses to multiple memory locations.
- The number and code of dynamic threads may not be available during analysis because a program spawns threads at runtime. Loops, if-then constructs and functions also complicate intra-thread control flow graph. Therefore, a V node may represent multiple instances of memory accesses by multiple threads at compile time, and we can only conservatively determine whether a memory access is shared by different threads.
- Because of imprecise V nodes, it is difficult to determine the conflict edges C that construct cycles. Although events, locks, and thread `start/join`

in Java enforce execution orders between different threads at runtime, we cannot analyze these orders for C edges precisely.

With an approximated code specification, Sura et al. [83] proposed that:

Theorem 1 *For a program edge from A to B , re-ordering A and B may break sequential consistency only if the code specification contains a path from B to A that begins and ends with a conflict edge between accesses in different threads.*

Figure 4 (a) shows that to test if a program edge from A to B cannot be a delay edge, we test if there exist two nodes C and D such that:

Condition 1

1. there are conflict edges between A and D , and between B and C , and
2. some instances of C and D are in different threads from A and B , and
3. there may be an execution order: (B, C, D, A) .

Conservative analysis can result in unnecessary delayed edges. For example, we may find cycles that are not critical: (A, B, C, E, F, D, A) in Figure 4 (b), or consider paths that are impossible at runtime (Figure 4 (c)). Some of these limitations can be reduced to single-threaded analysis problems. Sura et al. focuses on refining the algorithm by designing more precise concurrent analysis [84].

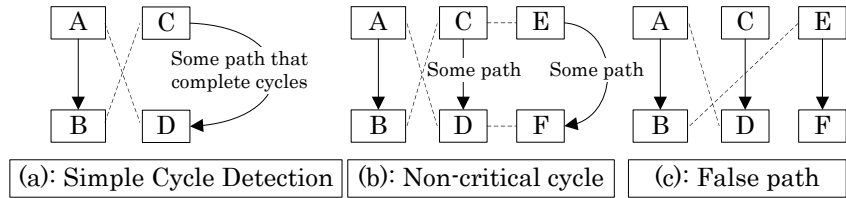


Figure 4: Simplified Cycle Detection

2.2.1 The System

Figure 5 gives the overview of the system based on the Just-In-Time (JIT) weak consistent Jikes Research Virtual Machine (Jikes VRM) [9]. The underlying architectures are Intel Xeon and IBM Power3.

Program Analysis. Before the first thread is spawned the compiler does not apply any analysis and affect any optimizations because single-threaded programs preserve sequential consistency by default. When the first thread **start** occurs, the compile starts inter-thread analyses over the whole program.

First, the system uses thread escape analysis to determinate which variables may be accessed outside a thread, then applies type-based alias analysis over

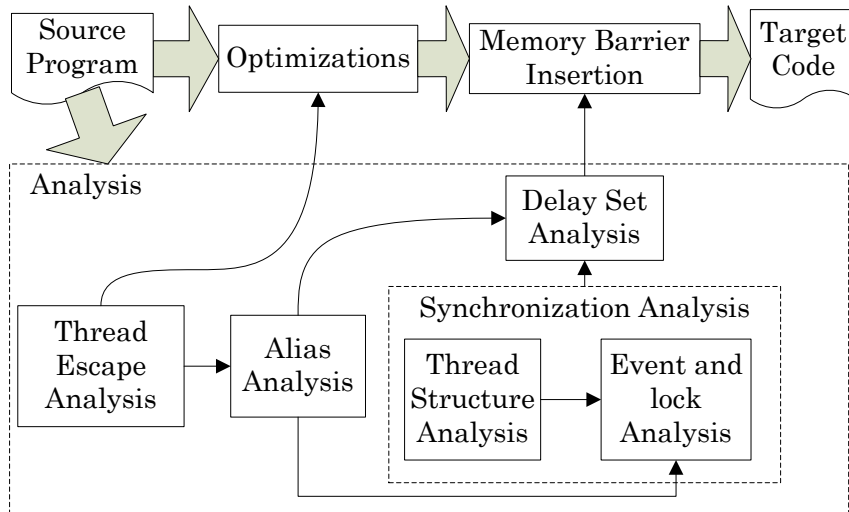


Figure 5: Compiler Infrastructure

potentially escaping variables to find conflict edges. Second, thread structure analysis computes instructions that may happen in parallel. For example, in Figure 6 (a) B and E can run in parallel, while A and D cannot run in parallel with E . Third, event and lock analysis remove ordered conflict edges that cannot be in cycles, as illustrated in Figure 6 (b) and (c).

After analysis, the compiler maintains an explicit call graph with a summary of analysis information for each method. Due to polymorphism and dynamic class loading, some type information is not available at compile time. In this case, the analysis optimistically assumes information for unavailable types, and incrementally updates the summary when the information is available later.

When testing whether a program edge between A and B should be delayed, instead of considering all possible individual program points that conflict with A or B , the analysis considers all methods that contain accesses that conflict with A or B and cannot be refined by method summaries of the above analyses. Appendix C explains these analyses in more detail.

Transformations. The compiler inhibits optimizations if they re-order, eliminate or insert memory accesses to thread escaping variables, rather than applying more precise but more expensive delay set analysis (Appendix B), because:

- Most of the benchmarks have no significant change in performance due to these optimizations. If all these optimizations are inhibited, experiments show 7 out of 10 benchmarks have no overhead, and the rest have a slowdown of 4% on average.
- Program optimizations can change the delay set analysis. We must update

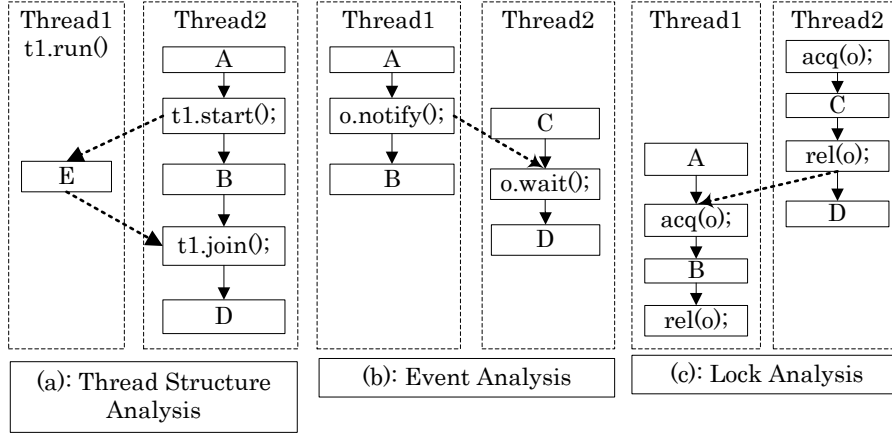


Figure 6: Synchronization Analysis

or re-compute the set of delay edges after each optimization if we inhibit optimizations in terms of delay edges.

Therefore, the delay set analysis is only used to insert memory fences when generating target code [29].

Complexity and Performance. The table below summarizes complexity of the program analyses in the system.

Analysis	Complexity	Property
Thread Escape	$n * (n - n_c) * m * n_c^n$	partial context-sensitive
Alias	n	type-based
Thread Structure	$n^2 * s$	inter-procedural, inter-thread
Event	$n_t^2 * e * t$	method summary
Lock	$n^2 * l$	inter-procedural, inter-thread
Delay Set	$n^2 * m^2$	method summary

Here, n , n_c and m are the numbers of program points, call sites and methods; s , e and l are the number of thread starts, events and locks; t is the number of thread types. The table does not consider analysis dependency. So the total complexity of delay set analysis is the sum of the complexity of all analyses.

If we do not consider the compile time in the JIT compiler, the implementation of sequential consistency based on delay set analysis shows a slowdown of 10% on average over the default weak consistency model, while the average slowdown of sequential consistency without delay set analysis (that inserts memory fence for every shared variable access) is 26.5 times. So the analysis has a high impact on performance. Thread escape analysis is the most expensive analysis. For the benchmark used in the paper, a delay set analysis based on a trivial escape analysis that assumes almost all memory references escape introduces an

average slowdown of 23 times. So the delay set analysis is sensitive to thread escape analysis. Other synchronization analyses improve delay set analysis by 80%, but the effect is variant for specific programs.

Jikes VRM is a JIT compiler, so the real execution time also includes the analysis time. Due to complexity of inter-thread analysis and recompilation caused by dynamic class loading, the analysis time of 3 out of 10 benchmarks is more than 10 times slower than the weak consistent compilation. The compilation overhead can be amortized over a long running time. However, recompilation is important. Otherwise the system must conservatively analyze unavailable information and lead to more memory fence insertion.

2.2.2 Discussion

In this paper the performance of sequentially consistent execution depends on the accuracy of the static analysis (Appendix C). Analysis over a JIT compiler leads to more precise results because we can assume optimistic information for unavailable components, and update analysis summary when they are available. In a static compiler we can only conservatively analyze unavailable components. But the system has several limitations.

First, all the analyses use only simple alias analysis because the benchmarks do not require powerful alias analysis or shape analysis. But, in general, there exist programs that use recursive data structures such as trees, and different threads work on data in distinct parts of the data structure. The type-based alias analysis considers all distinct subtrees to be shared because they are of the same tree type. Moreover, in a programming language with weaker types, such C++, alias analysis cannot benefit from types. There also exist applications that analyze large amounts of data stored in arrays by using different threads access disjoint regions in a shared array. An array analysis that does not depend on index values will consider these disjoint regions to be shared. In all the cases, the analysis in this paper must have significant performance degradation.

Second, most of the benchmarks have no significant change in performance due to inhibiting the compiler optimizations for escaping variables because they are almost free of data races. However, the experiments in [84] show that the part of code that has data races has a slowdown of 49 times by inhibiting the compiler optimizations. Therefore, programs with data races in hot methods can have large slowdowns because of both memory fence insertion and disabling compiler optimizations.

Third, it is challenging to verify transformations based on inter-procedural and inter-thread analysis. Mechanizing loop-based intra-procedural optimizations with formal proofs is still an open question [85]. Moreover designing a verified, efficient and effective analysis algorithm is more difficult.

2.3 Enforcing Sequential Consistency by Speculation

Enforcing SC by identifying cycles in $C \cup P$ at compile time leads to noticeable slowdown in weak consistent hardware because of inaccurate static analysis.

An alternative SC implementation is forcing memory operations from different threads to execute in a total order at runtime. A straightforward implementation inhibits any compiler and hardware optimizations, only allows one thread to execute one instruction in its program order, but limits performance significantly. The optimization of this implementation is based on two observations about the properties of cycles in $C \cup P$ and program applications.

Serialized Chunk Execution. Recall that an execution that violates sequential consistency imposes an inconsistent execution order with program order— $E \cup P$ has cycles (Figure 3 (a)). Actually, as illustrated in Figure 7 (a), given a cycle in $C \cup P$, if E orients all conflict edges in the same direction, then there cannot be a cycle in $E \cup P$ no matter how the compiler or hardware reorders instructions within a thread. Moreover, an execution without conflict edges is trivially sequentially consistent.

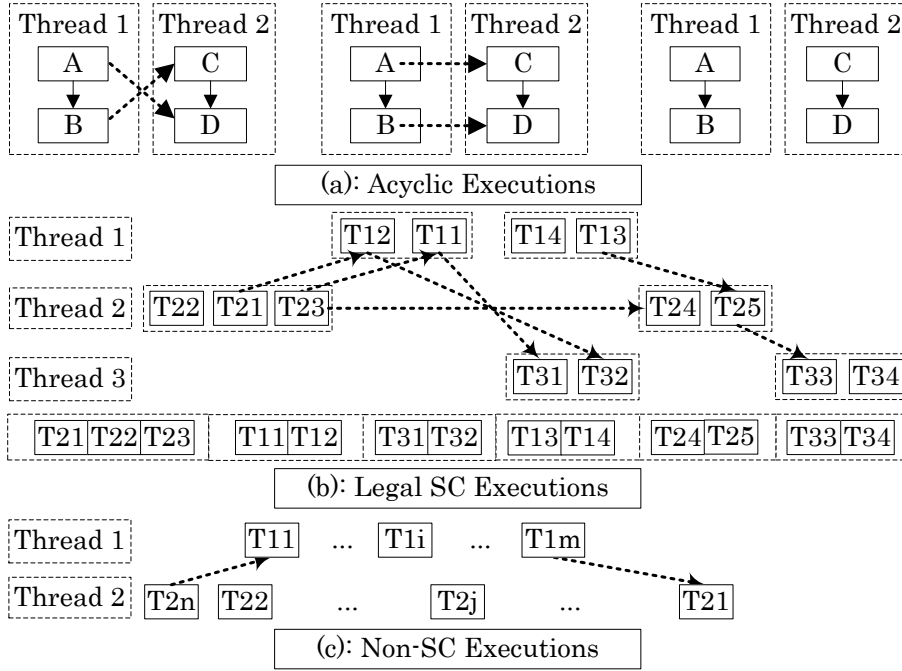


Figure 7: Ensuring SC Executions at Runtime

If the runtime enforces a total order for chunks of program, then any compiler and hardware optimization within a chunk is legal, and we may amortize the overhead to enforce the total ordered chunk execution. Consider Figure 7 (b). Here, T_{ij} denotes the j -th instruction in the program order of the i -th thread. Although instructions within each thread are re-ordered, we can still obtain a total order of all instructions consistent with the program and execution order because of the total order of chunks. More precisely, the runtime should ensure

that all chunks are *serialized*:

- **Req1.** Chunks from individual threads maintain program order.
- **Req2.** Chunks from all threads maintain a single sequential order.

Every sequentially consistent execution is trivially equivalent to a chunk execution with chunk size of one. However, a non-SC execution cannot be a chunk execution. In Figure 7 (c) the execution that reorders $T21$ and $T2n$ is not sequentially consistent because $(T11, T1m, T21, T2n, T11)$ is a cycle. Because reordering only happens within a chunk, $T21$ and $T2n$ are in a chunk c , but there must exist another chunk c' that is both earlier and later than c . This is impossible.

A trivial implementation is that each time one thread can execute a chunk of program with optimizations within the chunk atomically and in isolation, while the other threads stall. However, we want all threads to execute chunks concurrently.

Speculative Execution. Applications in practice usually have the following properties:

- Most of the code of an application is well-synchronized;
- Most of the critical sections in synchronized blocks are low-contention because applications assign disjoint shared data to different threads or they are rarely invoked, although the low-contention code cannot be lock-free because of shared variables accesses;
- Most of shared-variable accesses are reads. For example, the data insertion procedure takes most of the time to search in a linked list, and only updates an element in the list when the position to insert is found.

Therefore, speculative execution [26, 70] is proposed to enable more concurrent computation. Different threads can first create checkpoints of their current status, and then speculatively execute without synchronization. During speculative execution a thread buffers memory updates locally, and commits these updates to be visible when the speculative execution is finished without conflicting speculative executions from other threads. If two threads conflict in speculative executions, one of them will squash its execution, and restart from its checkpoint. In most cases, two speculative executions compute using disjoint shared data, or only read shared data in parallel without any conflict.

Moreover, speculative execution enables more optimizations. Compiler and hardware do not transform program across synchronizations because single-threaded data flow analysis is not preserved by synchronization operations. However, speculative execution is atomic—no other threads can interfere with values of shared variables in a speculative execution. Therefore, the optimization in Figure 1 may be correct in term of a speculative execution.

With the above considerations, Ahn et al. [8] proposed BulkCompiler that at compile time partitions a program into chunks that only need limited resources

to record their runtime information—memory locations read or written, applies optimizations across synchronization within each chunk, and then executes the program by the BulkSC hardware [22] that speculatively executes chunks with the maximum number (2,000) of dynamic instructions, and preserves sequential consistency by ensuring the total order of committed chunks with a global arbiter. Although software can also implement speculative executions [36], its performance overhead neutralizes the speed-up from compiler optimizations. Appendix D presents BulkSC. The next section presents BulkCompiler.

2.3.1 BulkCompiler

BulkCompiler lifts the BulkSC idea to compile time, and provides a complete implementation of sequentially consistent Java. Moreover, because at compile time we can apply optimizations to larger regions with the whole program analysis than BulkSC, BulkCompiler outperforms the weak Java Memory Model by an average 37%.

Creating Chunks at Compile Time. The first problem is how to partition a program into chunks that maximize compiler optimization, but minimize squashes. Traditional compilers only transform shared variables within regions separated by synchronization operations. Merging regions that contain shared variables across synchronization operations into a chunk can enable more optimizations for speculative execution. Although a larger chunk can increase optimization regions, it leads to more squashes. First, more threads may interfere shared variables, and result in data collision. Second, the data cache will overflow. Third, during the long period of execution, more uncatchable events, such as system calls and interrupts, could occur and squash a chunk.

To address the problems BulkCompiler uses a profile-driven infrastructure. Given a program, it first profiles whether a synchronization variable has high contention or low contention. A high contention synchronization block is in a chunk that only contains the block, while low contention blocks can be grouped with more adjacent statements to a larger chunk.

With the profiling information BulkCompiler first uses thread escape analysis (Appendix C) to identify potential shared variables, initializes each chunk with one shared variable, then merges these chunks with adjacent statements that are within the same control structure, and may fit in the cache. But a chunk in a high contention synchronization block does not contain any statements outside the block. To compute whether a chunk can fit in the cache without inter-procedural or loop analysis, BulkCompiler inlines functions and transforms loops to be with a constant iteration count of the innermost loop.

Safe Version of the Chunk Code. The second problem is how to define actions at different events such as system calls, cache overflow and data collision. Committing the current execution or restarting the execution with reduced chunk size breaks SC, because the compiler transforms code in the chunk created at compile time.

To handle these events, BulkCompiler generates a safe version of each chunk code at compile time, and optimizes the safe version only within synchronization

blocks. If the runtime rollbacks and restarts the same chunk several times, it switches to the safe version of the chunk, and lets BulkSC control its execution. Because the optimizations for the safe version is conservative, it is safe to let BulkSC handle the events.

int X = 0; int Y = 0; // Initially	
Thread 1	Thread 2
X = 1; while (Y==0);	while (X==0); Y = 1;

Figure 8: Handshake between two threads.

Moreover, the safe version of a code is also important to ensure that a transformed program makes progress if its original program is not always deadlocked in a fair scheduler. Consider the code in Figure 8. If the scheduler is fair, both threads can eventually make progress. However, if they execute in two chunks, neither of them can complete because they must read the other’s update. BulkCompiler runtime counts the number of completed instructions, and switches to the safe version if the number is high.

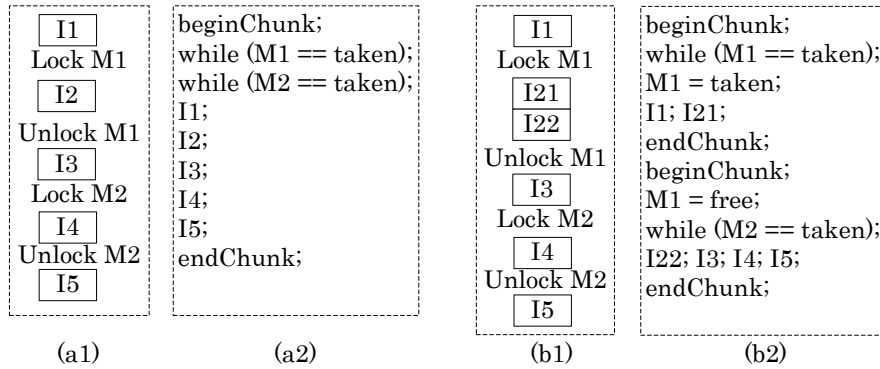


Figure 9: BulkCompiler Compilation

Compilations. With the above considerations, Figure 9 illustrates the transformations to extend optimization regions before BulkCompiler applies classic optimizations (a(1) and b(1) are the original programs, a(2) and b(2) are the transformed programs). Ideally if there were no safe version of the chunk code, and all chunks contain complete synchronized blocks, we could delete `while(M1==taken);` and `while(M2==taken);` (they check if the locks are taken by other threads) from Figure 9 (a). Because of atomic chunk execution, no threads can observe the status of locks. However, this invariant is broken if a synchronized block is separated into different chunks, or executed in safe version. Figure 9 (b) shows the former case where we need to explicitly

change lock status such that no other threads can obtain the lock between the two atomic chunks. The safe version of a synchronized block is in the similar status because BulkSC can partition the block into separate chunks. Therefore, Figure 9 (a) still needs to check lock status at the beginning of the chunk with spinning. Note that it accesses locks by plain loads, which are much cheaper than lock acquires; although checking all the locks at the beginning may reduce concurrency, such effect is insignificant because we apply this transformation to low-contention critical sections.

To preserve sequential consistency for safe version BulkCompiler inserts fences for shared variables by thread escape analysis. These fences only inhibit compiler optimizations, but do not restrict hardware optimizations because BulkSC takes fences as *no-ops*.

After the above transformations BulkCompiler applies classic optimizations in Hotspot Java Virtual Machine [66]. Experiments show that it outperforms the weak Java Memory Model by an average 37%.

2.3.2 Discussion

The primary reason that BulkCompiler outperforms the weak Java Memory Model is that high-performance speculative sequentially consistent executions by the hardware BulkSC extends optimization regions. However, it is interesting to compare BulkCompiler with compilers over hardware that supports speculative weak consistent executions [14]. They may have compatible performance for well-synchronized programs because they have the same optimization scope, while for applications that frequently use shared variables in hot methods, we may observe slowdown of BulkCompiler due to chunk squashes.

The other reason of performance improvement may be relative to properties of Java applications. First, Java programs have many low-contention critical sections in the form of synchronized methods—often in thread-safe Java libraries, which provides the potential optimization space. Second, among the five improved optimizations, two of them eliminate redundant null check and range check from loops that contain synchronization operations. Java runtime needs to insert null checks for every object reference, and range checks for every array access, but C/C++ does not have such overhead by default. Although BulkCompiler enables more register allocations because of extending optimization scopes across synchronizations, larger chunks may increase register pressure that causes performance overhead by register spilling in C++ [40]. The in-lining preprocessing may also increase register pressure. It is interesting to evaluate how each improved optimization affects performance.

Third, profiling information is important for BulkCompiler to minimize squashes by creating tight chunks for high-contention critical sections. The well-structured *synchronized* blocks in Java enable synchronization variables profiling. However, SC does not distinguish synchronization and ordinary operations, so it is difficult to profile the code with implicit locks (in Figure 8). Moreover, the chunk creation analysis depends on in-lining and loop tiling to compute if a chunk fits in the cache. When a function recursively traverses trees

and graphs, or is external, the analysis conservatively assumes it can cause cache overflow, and inhibits optimizations for the function. In that case the chunk instrumentation overhead may become more significant.

3 The Data-Race-Free Memory Model

Although sequential consistency gives users an intuitive programming environment with interleaved executions through a unique central memory, it is expensive to implement: we need either conservative analysis at compile time, or specific hardware that vendors are unwilling to commit.

Moreover, while sequential consistency seems to be the simplest model, it does not prevent common sources of concurrency bugs. To reason about a program users need to consider interleaved executions from different threads. The number of all possible interleaved executions of a program may be extremely large. Additionally, if low-level executions do not preserve atomicity of high-level instructions, the behavior of a program depends on the granularity of interleaving steps. For example, the final value of $X=100,000$; $X=600,000$ may be 125,536 if at low-level the second assignment occurs between the high and low half of the first assignment. There is also much work on enforcing atomicity, but in general such enforcement is also expensive, and does not always exist [78].

Recall the code specification in Figure 3, we observed that both of the problems arise because that sequential consistency does not restrict possible orientation of conflict edges C statically, so both compiler or hardware must protect against any possible cycles in $C \cup P$, and the number of cycles increases the search space of finding concurrency bugs. As observed in Figure 7 if a memory model imposes execution order, the memory model implementation can get consistent execution for free.

Programming languages already provide *synchronization* mechanisms (locks, monitors, or transactional memory) for enforcing execution order between threads. For example, in Figure 6 (c), locks ensure that C happens before the $\text{rel}(o)$ in Thread 2; the $\text{acq}(o)$ in Thread 1 happens before B ; the $\text{rel}(o)$ in Thread 2 happens before the $\text{acq}(o)$ in Thread 1; transitively C must happen before B . Moreover, most of applications are well-synchronized with directed conflict edges enforced by synchronization operations. If we only consider well-synchronized programs to be correct, the memory model can guarantee sequential consistency only if a program is well-synchronized, but does not guarantee anything otherwise. This memory model is called *data-race-free* (DRF) [3, 5, 34]. More precisely, a program has data races if more than one thread can access a memory location simultaneously in a sequentially consistent execution, and at least one such access is a store; a program is well-synchronized if all of its executions are free of data races.

DRF avoids the above flaws in the sequentially consistent memory model: i) because all non-synchronization operations between synchronization operations execute atomically, any compiler or hardware optimizations that are correct in single-threaded semantics are safe to apply within synchronization operations;

ii) a possible execution must be interleaving with respect to regions separated by synchronization operations. The implementation of DRF only needs to ensure that synchronization operations enforce execution order of non-synchronization operations.

Ada [39], UPC [47], OpenMP [38], and C++ [17, 13] define their language-level memory models similar to DRF. Appendix E formally defined the data-race-free memory model. Section 3.1 represents the compiler technique for automatic reuse of classic compiler optimizations in DRF [40]. We discuss and conclude in Section 3.2.

3.1 Reuse of Classic Compiler Optimizations across Synchronizations

int X = 0; int Y = 0; // Initially	
Thread 1	Thread 2
<pre> r1 = X; lock l; Y = 1; unlock l; do { lock l; r2 = Y; unlock l; } while (r2 == 1); r3 = r1; </pre>	<pre> do { lock l; r4 = Y; unlock l; } while (r4 == 0); X = 11; lock l; Y = 2; unlock l; </pre>

Figure 10: Copy-propagating $r1 = X$ to $r3 = r1$ in Thread 1 preserves data-race-freedom, but does not preserve semantics.

In DRF an optimization is safe if a transformed program preserves data-race-freedom and the semantics of the original program. More precisely, a program reflects data-race-freedom if a transformed program contains data races, then so does the original program. Most existing compiler or hardware optimizations satisfy the properties because they transform programs only between synchronization operations by preserving sequential semantics. However, optimizations across synchronization operations may be unsafe. Figure 10 shows a data-race-free program. If $r1 = X$ were copy propagated to $r3 = r1$, the final value of $r3$ is 11, which is impossible in the original program. Although the transformation preserves data-race-freedom, Thread 2 updates X asynchronously.

Serial compilers, such as gcc, takes synchronization operations as opaque functions to disable any optimizations across them, even if some data operations are not asynchronously interfered by other threads. Although there is some work that allows moving data operations outside a critical section into the section [16, 88], copy-propagating X is equivalent to moving data operations

inside a critical section out of the section. This is not safe in general. To enable the transformation, X must satisfy a stronger property than data race freedom, that is X is free of interference from other threads within the confines of all execution paths from $r1 = X$ to $r3 = r1$. For example, while Thread 1 is executing the path from $r1 = X$ to $r3 = r1$, no other threads update X . This property is called “siloed”, and explained in Appendix F more precisely.

At the high-level siloed analysis relies on alias analysis and procedural concurrency graph (PCG) to identify all possible interference between threads. A PCG is an undirected graph whose nodes are user-defined functions. An edge between two nodes indicates that the two functions may run in parallel. Each edge between $f1$ and $f2$ has an immediate interference function $Ii(f1, f2)$ that includes all variables with which $f1$ and $f2$ can conflict. Initially $Ii(f1, f2)$ contains all aliased variables between $f1$ and $f2$ indicated by an alias analysis. Then the paper [40] presents four refinements to improve a PCG’s precision:

1. Functions only executed by the main thread do not run in parallel with each other.
2. Functions only executed by the main thread and before any thread *spawn* do not run in parallel with other functions.
3. A function, that is in a thread $T2$ spawned by a thread $T1$ and does not spawn other threads, can only interfere with variables accessed by statements following the spawn site of $T1$ if $T1$ does not join $T2$.
4. The functions with disjoint lock sets do not run in parallel with each other.

Note that the refinements are not expensive: the second and the third refinements are the most expensive analyses that require flow-sensitive intra-procedural analysis. The compiler executes iteratively. In each iteration it applies siloed analysis to check if more optimizations across synchronizations are enabled. The iteration terminates when the analysis does not enable more optimizations. The average analysis time is 6% of the baseline. However, results on the benchmark show that performance improvements of up to 41% are possible, with an average improvement of 6% across all the tested programs over all thread counts. The last refinement is the most effective one that benefits from DRF. Intuitively, if two conflicting accesses from the two functions could execute in overlapped execution paths without being protected by a same lock, there would be a data race after reordering them to be adjacent.

The paper also reported a secondary effect of enabling optimizations in larger regions—register pressure. The register spilling overhead neutralizes the speed-up from enabled optimizations. The authors suppressed this problem by factoring out loops within optimization regions into functions.

3.2 Discussion

DRF allows most serial optimizations, and also simplifies optimizations across synchronizations: the analysis needs to focus on synchronizations only where

asynchronous accesses may occur; DRF provides effective and efficient refinement algorithms. In sequential consistency, shared variables may be asynchronously accessed at any program point. Therefore, researchers proposed specific intermediate representations to model such asynchronous accesses [80, 45, 59, 75, 49, 65, 73]. These optimizations are expensive and conservative due to the complexity of the intermediate representations.

However, we cannot reuse any serial optimizations for free. An optimization that may introduce data races can break data-race-freedom. Consider the following examples.

The original	The transformed
<code>r = X;</code>	<code>r = X; X = X;</code>
<code>for (i=0;i<n;i++) { X = r; ... }</code>	<code>X = r; for (i=0;i<n;i++) { ... }</code>

Suppose the original program in the first example is data-race-free although other threads still read `X` simultaneously. However, inserting an “identity” assignment `X = X` introduces a data race of `X`, and turns the behavior of the transformed program to be undefined. Similarly suppose `n` is always 0 at runtime in the second example. The original program does not update `X`, but the transformed program introduces a data race by speculatively updating `X`. Note that both of the transformations are legal in sequential settings, but we cannot reuse them in DRF. Such optimizations should be modified to ensure speculative data accesses occur only when the original program also executes them.

Moreover, the discussion of this section is based on a simple version of DRF. The realistic language-level memory model adopts DRF with compromises. For example, the C++ memory model [17] allows experts to program with different consistent atomics by annotations—from sequential consistency to weak consistency. To implement such a model, the compiler may still need delay set analysis to enforce sequentially consistent atomics, and other analyses to enforce weak consistent atomics.

4 The DRFx Memory Model

The primary reason that data-race-free can reuse sequential optimizations between synchronizations is that the memory model requires programmers to ensure that source programs are free of data races, while it does not define the behavior of programs with data race. Therefore, the implementation does not need to apply any expensive analysis at either compile time or runtime. As shown in the last section, optimizations across synchronizations also benefit from data race freedom.

Although this design choice significantly improves performance, programmers cannot estimate the damage caused by a data race, and enforcing data race freedom statically is difficult. Debugging and testing parallel programs in sequential consistency is already hard because programs execute nondeterministically, and data races normally lead to either data corruption or crashes long after the data races are actually executed. Compiler optimizations that mistakenly assume the absence of data races may result in nonintuitive behavior that complicates debugging and testing further.

int X = 0; int Y = 0; // Initially	
Thread 1	Thread 2
<code>r1 = X;</code> <code>Y = r1;</code>	<code>r2 = Y;</code> <code>X = r1;</code>
Is r1 = r2 = 42 allowed?	

Figure 11: Out-of-thin-air.

Moreover, undefined behavior breaks security properties in the programming languages with strong type, such as Java. Consider the two threads with data races in Figure 11. If the compiler speculatively loaded X and Y with value 42 into r1 and r2, storing 42 into Y and X would validate the speculative loads, and result in an output *out of thin air* [54]. Although no compiler or hardware behaves in this way, this violates the basic safety property, *e.g.*, we cannot prove that a password must be inputted by a user.

However, it is challenging to define a program with data races, such that it is easy to understand, but also safe and of high performance. First, it is difficult to define well-formed executions of programs with data races. Section 2 shows that the intuitive interleaving semantics decreases performance. The Java Memory Model [54] attempts to allow as many safe optimizations as possible but disallows optimizations with out-of-thin-air problems. However, the result is not entirely satisfactory, in that it is overly complicated to implement, and prohibits some optimizations that were intended to be allowed [89]. The behavior of weak executions does not simplify debugging and testing either.

Second, a compiler cannot precisely preclude data races statically. There are a variety of static type and effect systems proposed to ensure data race freedom [32, 1, 30, 32, 76, 7, 19, 87, 41]. Due to the lack of thread structure and alias information, automatic analysis is conservative to determine shared variable accesses, lock sets, and temporal order between memory accesses [69, 68]. Semiautomatic analysis requires programmer annotations [15] to increase analysis accuracy, but incurs programmer burden. It is also difficult to analyze third-party libraries without source code.

Third, detecting data races at runtime has significant performance overhead. At the case a compiler cannot determine whether an access is of data race, the runtime can continuously monitor and dynamically detect that a “problematic” data race is about to execute and raise an exception, instead of executing the

data race [57, 11, 10, 31, 18, 28, 60]. Since a memory model is designed for any phase of development, dynamic data races must be cheap for always-on use.

A precise detector usually monitors data races by checking *happen-before* order with vector lock algorithms. In practice memory locations to access can be as small as a byte; the number of threads is possibly hundreds or thousands; the runtime cannot simply remove lock information because two conflicting data accesses can be arbitrarily far away. This can slow down execution by a factor of 8 or more for software-based detection [31]. Hardware-based detection is efficient, but it must monitor unbounded program regions with limited resources, which requires significant changes to the architectures [60, 6].

4.1 DRFx

Although none of the above approaches solve the problem, the last one provides a better trade-off among expressive programming, debugging and safety: programs without data race still execute sequentially consistently with possible optimizations, and subtle failures from accidental data races are reported directly at the point of occurrence, and produce easily describable outcomes.

Importantly, the observations in Figure 7 show that if we only detect data races that violate sequential consistency, the runtime only needs to maintain information for memory operations in uncommitted chunks, but removes information of committed chunks. Like BulkCompiler in Section 2.3, the implementation preserves sequential consistency as long as all committed chunks can be *serialized*, and only raises exceptions if there are data races in uncommitted chunks. Although data races can occur in existent chunk executions, they do not break sequential consistency. If the compiler can partition the program into chunks with memory operations that hardware-based detection can monitor with limited resources, and restrict optimizations within each chunk, the hardware can be both simple and efficient. Such a memory model is called DRFx [56].

DRFx guarantees two properties [56]:

1. If an execution of a program violates sequential consistency, it eventually terminates with an *Memory Model (MM) exception*, and the observable behavior of the execution before raising the MM exception is a prefix of sequentially consistent execution of the program.
2. Any execution of a program without data races is sequentially consistent and does not raise an MM exception.

The first property guarantees *soundness*—the implementation of DRFx must identify all non-SC executions, and *safety*—the detector must be precise enough to detect a violation before it causes visible effects. The *safety* requirement assumes unsafe behavior only happens through an observable effect such as termination and system calls. So the detection accuracy depends on observable granularity. For example, changing data values via data races is not detected immediately if we do not consider it as visible effects, although soundness ensures

the execution eventually terminates. However, any visible effects should be only from a sequentially consistent execution. This also simplifies debugging and testing. The second property guarantees SC for well-synchronized programs as data-race-free does.

4.1.1 Sufficient Requirements

The paper [56] proposed the sufficient requirements for compilers and hardware to implement the DRFx memory model. First, the compiler applies sequential optimizations only within chunks separated by fences, and the hardware ensures all chunks are committed in a total order in terms of the fence after each chunk. Moreover, each synchronization is in its own chunk. By doing so, if a transformed program has a sequentially consistent behavior, its original program must have an equivalent sequentially consistent behavior.

Second, to ensure that an MM exception must happen before non-SC effects are observed, we also require that each system call is in a singleton chunk, and has only thread-local arguments; non-terminating and system exceptions should not prevent detecting the MM exception.

Third, compiler optimizations do not introduce extra shared variable accesses; the data collision detected by the hardware must be from uncommitted chunks; committed chunks are total-ordered and do not depend on uncommitted chunks. This follows that the committed chunks have the same sequentially consistent behavior that the original program has, and that if there is a data conflict in an execution of a transformed program, its original program must have a data race.

Appendix G gives informal proofs that an implementation that satisfies the above requirements preserves DRFx.

4.1.2 Implementation

The implementation must ensure that the runtime can detect MM exceptions with respect to source programs in a simple and efficient way. A longer chunk enables more optimizations, introduces less performance and traffic overhead to enforce chunk order, but requires more resources to detect MM exceptions.

Experiments show that inserting fences for each chunk has a significant performance degradation, about 92% on average. The interesting design of the implementation is to determine the chunk size by compiler and hardware cooperation. At compile time the compiler conservatively partitions chunks with size limited by the cache (unlike BulkSC that limits chunks by dynamic instruction numbers), and strictly restricts optimizations within chunk. To increase the optimization opportunity at runtime, the compiler only requires hardware to insert fences around synchronizations and system calls, but lets the hardware decide whether to insert fences between other chunks.

If the cache is not overflowed, the hardware continues executing the next chunk without committing the current chunk, and also applies optimizations across chunks. At cache overflow, which can only be caused by successive chunks,

the hardware commits the previous chunk. Therefore, both the compiler and the hardware preserve sequential semantics within chunks, while the hardware enables larger optimization scopes. Like BulkCompiler, to compute chunk size precisely the compiler in-lines functions and unblocks loops to improve compiler optimizations. The total overhead is about 3.25% on average with the data-race-free baseline.

The hardware extends the local cache to detect conflict *lazily*. Each process has a buffer to store the time stamps of the beginning and the end of a chunk, and locations read or written by the chunk. When a chunk is completed, the hardware checks whether the local accesses conflict with remote processors. On detecting a conflict, an MM exception is thrown. Otherwise, the local processor clears the buffer, and continues. Therefore, the hardware does not need checkpoint, rollback and cache overflow mechanism.

4.2 Discussion

The DRFx hardware is simpler than BulkSC [22] because the different properties between sequential consistency and DRFx: for programs with data races, sequential consistency defines interleaving semantics, while DRFx raises an MM exception. First, the DRFx hardware only raises an MM exception on detecting conflict, and does not need rollback and checkpoint. However, BulkSC rollbacks and restarts the execution until chunks are *serialized*.

Second, the global arbiter is significant in BulkSC to provide a total order and atomicity for committed chunks because local updates are only buffered before committed. Each processor in DRFx does not buffer local updates, but directly accesses the shared memory. So a processor either depends on updates from committed chunks that must be sequentially consistent, or from other uncommitted chunks. On committing, if uncommitted chunks depend on each other, the runtime raises an exception. Therefore, DRFx does not need a global arbiter. Third, speculative execution also complicates BulkSC in the corner case. It must reduce chunk size or pre-arbitrate threads to ensure that a program can progress. However, it is not safe to apply lock speculative elision for DRFx because we need to distinguish data races in the original program and the ones introduced by speculation.

DRFx also makes other trade-offs to simplify the design. First, DRFx assumes that sequentially consistent exceptions can delay until observable behavior occurs. By doing so, the DRFx implementation detects conflict lazily, rather than doing eager detection that precisely raises an exception at the earliest conflict. Eager detection is useful to find problematic code when debugging. However, it increases traffic overhead, and complicates hardware design.

Second, DRFx assumes that sequentially consistent violation is the main resource of bugs caused by data race. However, arguably atomicity violation [32, 76, 7] can also lead to bugs that DRFx does not prevent. For example, password management by a well-synchronized library should not leak information by intervening shared accesses. A critical section indicated by an atomic region can access memory arbitrarily larger than the size of conflict de-

tection buffers. Therefore, detecting atomicity violation within synchronized chunk requires hardware to handle unbound resource in a complicated way [53].

5 Discussion

We have presented different memory models and implementations for shared-memory multi-threaded programming languages. In this section we first study the design space of memory models, then summarize implementation techniques, and finally turn to discuss future research directions.

5.1 The Design Space of Memory Models

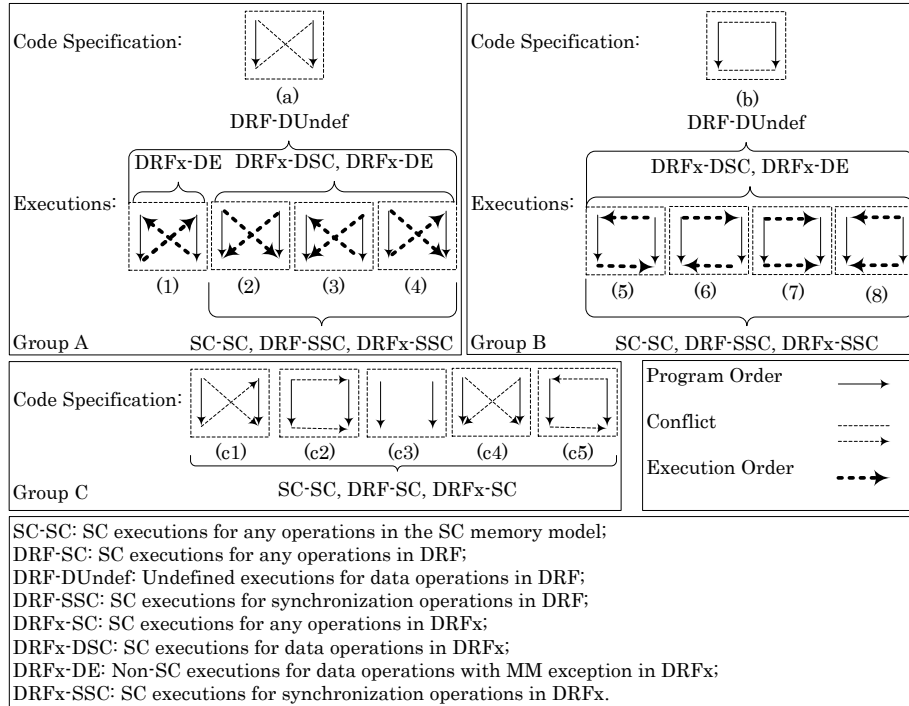


Figure 12: The Design Space of Memory Models

A memory model for shared-memory multi-threaded programming defines what a memory load returns—more precisely, what an execution behaves like when multiple threads access shared memory locations. From the perspective of programmers and implementations, a memory model should be

1. Friendly to program with and reason about,
2. Of high performance, and

3. Easy to implement.

However, as discussed in the previous sections, it is challenging to meet the above requirements altogether. A primary trade-off is how much a memory model supports sequential consistency that provides an intuitive view when designing, reasoning about and debugging programs, but is expensive to implement. Although data-race-freedom, a programming discipline of concurrency programming languages, reduces the overhead of ensuring sequential consistency, it is difficult to define semantics for programs with data races.

Figure 12 illustrates the design space of memory model based on the code specification of a program 2.1. Note that the code specifications in Figure 12 are not exhaustive, but focus on typical cases. There are three categories of code specification with respect to whether possible executions are sequentially consistent, and whether a program is free of data race:

- (A) a program has data races, and some of its executions are not SC;
- (B) a program has data races, but all its executions are SC;
- (C) a program is free of data race, and all its executions are SC.

The code specifications in Group C are well-synchronized, therefore, have sequentially consistent executions for free. All these memory models provide sequential consistency for all memory operations in Group C. They are different from the trade-offs in Group A and B that are not free of data race.

Although sequential consistency only needs to inhibit the execution (1) in Group A, to detect such executions at compile time precisely is non-trivial (Section 2.1 2.2), NP-complete even for a trivial language. Due to conservative analysis we observe a slowdown of 10% on average over the default weak consistency model because of disabling optimizations and memory fence insertions. Moreover, the overhead is still optimistic because the analysis may be optimized to speed up only the benchmark, but more conservative for general applications (Section 2.2.2).

BulkCompiler (Section 2.3) enforces sequential consistency by executing chunks in a total order at runtime, such that each chunk runs atomically and in isolation (Figure 7). Because most practical programs are well-synchronized (in Group C), and have chunk-serialized executions, the runtime should enforce chunk-serialized executions (in Group A and B) infrequently. So we can speculatively execute chunks, rollback and restart on detecting a conflict by an effective hardware—BulkSC. This technique does not insert costly memory fences, and the speculation also enables larger optimization scopes. Therefore, it outperforms the weak Java Memory Model by an average 37%.

On the other hand, data-race-free simply leaves programs with data races (in Group A and B) undefined such that it does not need to enforce any semantics with expensive compiler or hardware techniques because well-synchronized programs are common. This model enables most of sequential optimizations by

default, and also unblocks optimizations across synchronizations by data-race-freedom. The benchmark shows that performance improvements of up to 41% are possible, with an average improvement of 6%.

However, we still have to require synchronization operations to be SC, because they provide synchronization mechanism to develop well-synchronized programs. This restriction does not primarily affect performance of lock primitives because i) existent implementations of lock primitives already rely on memory fences to enforce sequential consistency; ii) in practice locks are used to synchronize a large number of memory accesses that provide sufficiently large scopes for hardware optimizations.

But this restriction affects performance of *atomic* variables in C++x0 or *volatile* variables in Java. These variables are designed to synchronize singleton variables, *e.g.*, a flag to indicate a read-only data has been initialized. Protecting them by locks to enforce sequential consistency is unrealistic because inserted locks are costly, and also blocking optimizations. Actually to enforce sequential consistency over fine-grain synchronization variables in DRF needs the similar techniques in the sequential consistency memory model. The Java Memory Model [54] attempts to enable as many optimizations as possible without out-of-thin-air, but the complex formalism is not entirely satisfactory. The C++ Memory model [17, 13] extends the happen-before order with weak consistent memory operations that give expert programmers a way to write very carefully crafted, but portable, synchronization code that approaches the performance of assembly code. Therefore, data-race-free does not totally solve the problems that sequential consistency has, but hides them in a corner.

Moreover, the data race freedom assumption introduces another problem—the undefined behavior of programs with data races, because detecting data races statically is undecidable, and dynamic full data race detection is also expensive. By the similar observation to BulkCompiler, DRFx addresses this problem by dynamically detecting SC violations and raising an exception—the execution (1) in Group A. The detected violation is conservative because it raises exceptions on detecting conflict, rather than restarting chunks to search for conflict-free executions. For example, the executions (2-8) in Group A and Group B may also be prohibited if conflicting accesses are too close to each other. But executions without exceptions are guaranteed to sequentially consistent. Therefore, the DRFx implementation is based on a hardware simpler than BulkSC. The total overhead is about 3.25% on average with the data-race-free baseline. Siloed analysis is still available in DRFx, and can improve performance further because it is stronger than detecting data race by requiring a variable be not interfered by other threads along any execution path. However, like data-race-free DRFx also suffers from enforcing sequential consistency for synchronizations.

In summary Figure 13 illustrates comparison between these memory models and weak memory models with respect to the requirements: programmability, performance and implementation.

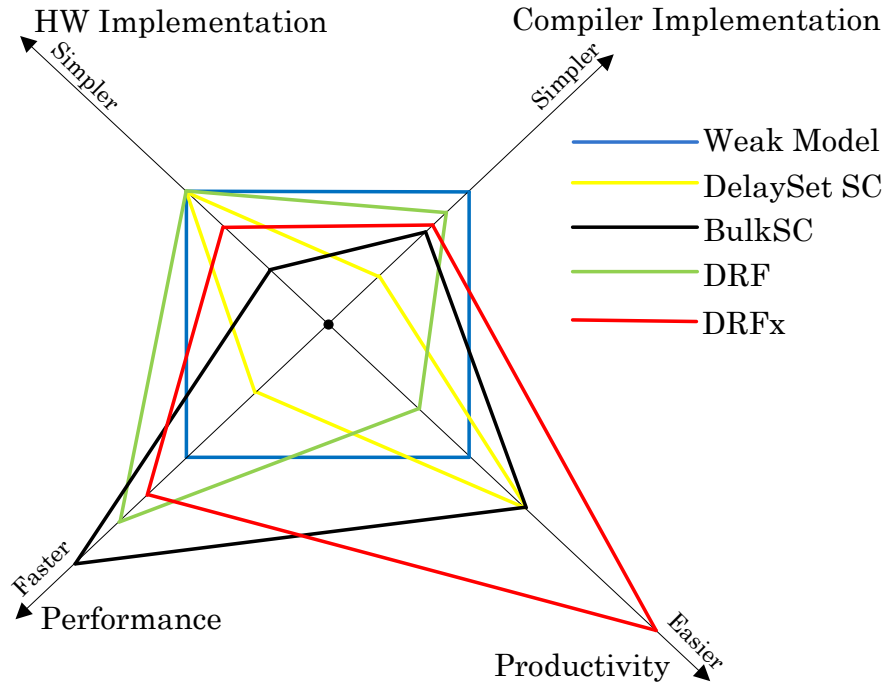


Figure 13: Comparison of Memory Models

5.2 Summary of Implementation Techniques

Note that the surveyed memory models are for hardware that does not distinguish synchronization operations and ordinary operations, and does not provide sequential consistency by default; and for programming languages that have expressive programming practices such as global address space, imperative languages, objected oriented programming, complex and pointer-based data structures, dynamic ownership of shared variables, flexible thread structures, *etc.* Implementing memory models for such hardware and language features inherently requires sophisticated program analysis and software/hardware co-design.

Program Analysis. We summarize the static and dynamic program analyses to implement memory models in Figure 14. The arrows denote dependency between these analyses. The analyses in the top box provide basic properties of shared-memory multi-threaded programs: *alias analysis*, *concurrency analysis* and *thread escape analysis*. The analyses in the bottom box extend the basic analyses to implement memory models: inhibiting optimizations to enforce sequential consistency by *delay set analysis*, optimizing programs across synchronizations in data-race-free by *siloed analysis*, partitioning programs into chunks in sequential consistency and DRFx by *resource analysis*.

Figure 14 also gives potential analyses to implement memory models:

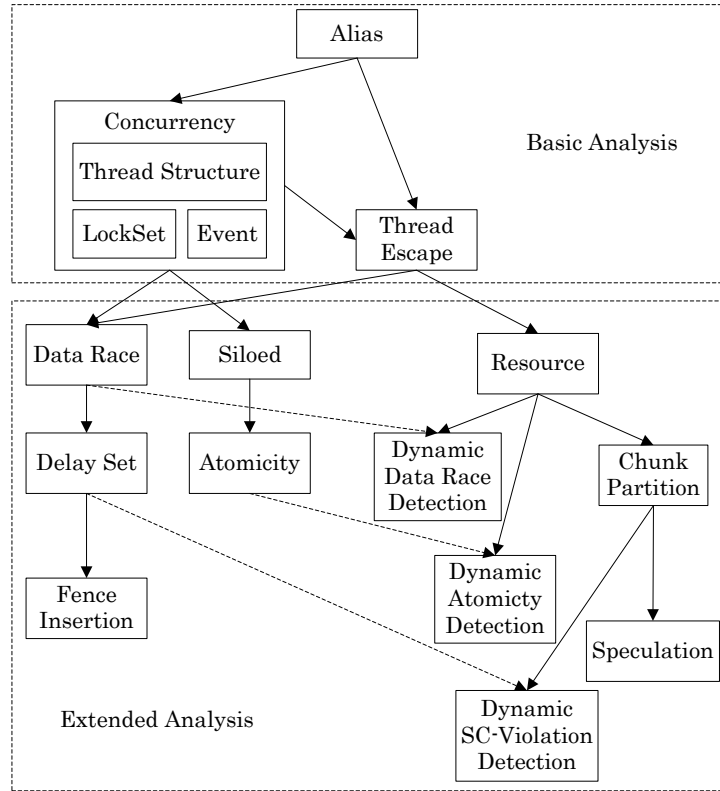


Figure 14: Program Analysis

- **Atomicity Analysis:** This is a stronger property than preventing SC violations in DRFx to improve debugging and testing. The siloed analysis along finer-grained paths between synchronizations ensures atomicity.
- **Refining resource analysis:** Existent chunk partitioning algorithms use simple resource analysis. Actually the runtime only needs to buffer shared data to detect conflicts because most of data are private. Removing private data by thread escape analysis can result in larger chunks to enable more optimizations, also reduce the runtime and memory overhead when detecting data race, atomicity, sequential consistency violation in software.
- **Hybrid Invariance-Violation Detection** (indicated by dot arrows): The trade-off between imprecise analysis at compile and efficient detection by specific hardware is detecting violation by instrumented programs. However, it suffers from runtime overhead. It is interesting to evaluate whether refined resource analysis and delay set analysis improve dynamic analyses.

Software/Hardware Co-design. The software/hardware co-design in

these memory model implementations is primarily to reconcile language-level memory models and hardware memory models. Although hardware provides high-performance, it does not fit in high-level requirements. First, we have argued that common users at least need sequentially consistent synchronization operations and atomic variables to design well-synchronized programs with reasonable productivity and performance overhead. However, most of hardware memory models only preserve weak consistency. Although they provide memory fences to enforce causality, it is not clear how to map atomic variables to existent hardware instructions without sacrificing performance. The experimental results in Section 2.2 show the performance overhead.

Second, the existent hardware does not support debugging and testing programs with data races by default. In the level of hardware memory model it is even difficult for expert programmers to understand the informal and complicated definitions of reordering. In the level of programming languages the damage caused by a data race bug usually occurs far away from where it happened, behaves non-intuitively, and is hard to reproduce by non-deterministic executions. However, detecting full data races by only software is still expensive [31]. Although the proposed hardware is efficient, it will take a long time for vendors to commit these techniques.

5.3 Programming Disciplines and Hardware Support

One research direction to address these challenges is developing programming disciplines for shared-memory multi-threaded languages with hardware support to implement memory models. First, we can raise the level of abstraction with threads to simplify thread escape and concurrency analysis. Deterministic Parallel Java (DPJ) [15] proposes a region-based type and effect system that uses region-based memory management, and summarizes each method by memory read and write effects. With these information DPJ checks if concurrent methods access disjoint memory regions, such that each method can execute atomically, and the program is deterministic. Therefore, DPJ can compile programs to well-structured task-based concurrency libraries such as OpenMP [38], TBB [71] and Cilk [33] with performance comparable to programs directly designed by threads.

Second, to support expressive programming, expert programmers can annotate programs and use assertions or dependent types to facilitate reasoning about complex programs at compile time—*e.g.*, explicitly developing programs with synchronizations or non-deterministically, changing ownership of shared variables (shared variables can be temporally private to a thread) [57], proving properties of recursive data structures and termination of recursions, summarizing libraries code and polymorphism. Those annotations can be seen as program documentations. On the other hand, researchers also proposed algorithms to automatically infer annotation [86]. Techniques based on SMT solvers may further increase inference precision in large-scale programs [72].

Third, it is impossible to precisely analyze any programs or infer any annotations, but it may be feasible to improve precision to important regions or

variables of a program where accuracy is vital. Different analyses are applied independently on disjoint regions of the program with boundaries that compose analysis results from different regions, such that each individual analysis is precise and scaled within each region [44]. Such regions or variables can be identified by heuristic information, profiling, and users annotations.

Fourth, to design more expressive but still safe programs, hardware should check properties that compiler cannot ensure—such as non-determinism, data race, atomicity and SC-violation. Moreover, hardware should support efficient sequentially consistent atomic operations.

6 Conclusion

In this paper we have studied the design space of memory models for shared-memory multi-threaded programming languages with their motivations, implementation techniques, challenges and open questions. We also discussed future research directions—developing programming disciplines and hardware to achieve a memory model that is easy to program, feasible to implement, and of high performance.

References

- [1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006.
- [2] Sarita Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4:613–624, 1993.
- [3] Sarita V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, University of Wisconsin-Madison, Madison, WI, USA, 1993.
- [4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.
- [5] Sarita V. Adve and Mark D. Hill. Weak ordering a new definition. *SIGARCH Comput. Archit. News*, 18:2–14, May 1990.
- [6] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19:234–243, April 1991.
- [7] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05*, pages 233–242, New York, NY, USA, 2005. ACM.
- [8] W. Ahn, S. Qi, M. Nicolaidis, J. Torrellas, J. W. Lee, X. Fang, S. Midkiff, and David Wong. BulkCompiler: high-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42*, pages 133–144, New York, NY, USA, 2009. ACM.

- [9] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, January 2000.
- [10] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded c. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 149–158, New York, NY, USA, 2008. ACM.
- [11] Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. *SIGPLAN Not.*, 44(6):98–109, 2009.
- [12] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In *Proceedings of the 20th international conference on Theorem Proving in Higher Order Logics*, Jul 2007.
- [13] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011.
- [14] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *In Proc. 36th Intl. Symp. on Computer Architecture*, 2009.
- [15] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. *SIGPLAN Not.*, 44:97–116, October 2009.
- [16] Hans-J Boehm. Reordering Constraints for Pthread-Style Locks. In *Proc. 12th Symp. Principles and Practice of Parallel Programming*, 2005.
- [17] Hans-J Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM.
- [18] Michael D. Bond, Katherine E. Coons, and Kathryn S. Mckinley. PACER: Proportional Detection of Data Races. *SIGPLAN Not.*, 45(6):255–268, 2010.
- [19] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proc. OOPSLA '01*, pages 56–69, Tampa Bay, FL, oct 2001.
- [20] John Boyland. An operational semantics including 'volatile' for safe concurrency. *Journal of Object Technology*, 8(4):33–53, June 2009.
- [21] Stephen Brookes. Full abstraction for a shared variable parallel language. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109, 1993.
- [22] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, New York, NY, USA, 2007. ACM.

- [23] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 115–125, New York, NY, USA, 2003. ACM.
- [24] Fred C. Chow, Sun Chan, Shin M. Liu, Raymond Lo, and Mark Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267, London, UK, 1996. Springer-Verlag.
- [25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13:451–490, 1991.
- [26] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT 2008: 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.
- [27] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, 21(2):9–21, February 1988.
- [28] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. *SIGPLAN Not.*, 42(6):245–255, June 2007.
- [29] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 285–294, New York, NY, USA, 2003. ACM.
- [30] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 219–232, Vancouver, Canada, jun 2000.
- [31] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, 2009.
- [32] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.
- [33] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998.
- [34] Kouros Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM.
- [35] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, 1998.
- [36] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

- [37] Aquinas Hobor, Andrew W. Appel, and Francesco Z. Nardelli. Oracle semantics for concurrent separation logic. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems, ESOP'08/ETAPS'08*, pages 353–367, Berlin, Heidelberg, 2008. Springer-Verlag.
- [38] Jay P. Hoeflinger and Bronis R. De Supinski. *The OpenMP Memory Model*, 2005.
- [39] International Organization for Standardization. *Ada 95 Reference Manual*, 1995. ANSI/ISO/IEC-8652:1995.
- [40] Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-J Boehm, and Dhruva R. Chakrabarti. A Technique for the Effective and Automatic Reuse of Classical Compiler Optimizations on Multithreaded Code. In *POPL*, 2011.
- [41] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07: Proceedings of the 19th international conference on Computer aided verification*, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.
- [42] G. Kahn. Natural semantics. In G. Goos and J. Hartmanis, editors, *Proc. 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.
- [43] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Validation of gcc optimizers through trace generation. *Softw. Pract. Exper.*, 39(6):611–639, 2009.
- [44] Yit P. Khoo, Bor Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10*, pages 436–447, New York, NY, USA, 2010. ACM.
- [45] Jens Knoop, Bernhard Steffen, Jürgen Vollmer, and J. Urgen Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 18, 1996.
- [46] Arvind Krishnamurthy and Katherine Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, 38, 1996.
- [47] William Kuchera and Charles Wallace. The upc memory model: Problems and prospects. Technical report, Technical Report LBNL (draft), 2004.
- [48] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, August 1979.
- [49] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing, LCPC '97*, pages 114–130, London, UK, 1998. Springer-Verlag.
- [50] Jaejin Lee and David A. Padua. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [51] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1999.
- [52] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, December 2009.

- [53] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10*, pages 210–221, New York, NY, USA, 2010. ACM.
- [54] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.
- [55] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFX: a simple and efficient memory model for concurrent programming languages. Technical report, UCLA Computer Science Department, November 2009.
- [56] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFX: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10*, pages 351–362, New York, NY, USA, 2010. ACM.
- [57] Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. Dynamically checking ownership policies in concurrent C/C++ programs. *SIGPLAN Not.*, 45(1):457–470, 2010.
- [58] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 402–422, London, UK, UK, 1990. Pitman Publishing.
- [59] C. Mohan and H. Pirahesh. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. Report rj 7342, IBM Almaden Research Center, San Jose, CA, feb 1990.
- [60] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: signature-based data race detection. *SIGARCH Comput. Archit. News*, 37:337–348, June 2009.
- [61] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338, New York, NY, USA, 2007. ACM.
- [62] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.
- [63] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '98/FSE-6*, pages 24–34, New York, NY, USA, 1998. ACM.
- [64] Diego Novillo. Memory SSA- A Unified Approach for Sparsely Representing Memory Operations. In *Proc of the GCC Developers' Summit*, July 2007.
- [65] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA Form in the Presence of Mutual Exclusion. In *In 1998 International Conference on Parallel Processing*, pages 356–364, 1998.

- [66] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot(tm) Server Compiler. In *In USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.
- [67] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981. Tech. Rep. DAIMI FN-19.
- [68] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.
- [69] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical Static Race Detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.
- [70] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.
- [71] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.
- [72] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 131–144, New York, NY, USA, 2010. ACM.
- [73] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25:70–116, January 2003.
- [74] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming*. Elsevier, 1989.
- [75] Vivek Sarkar. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, pages 94–113, London, UK, 1998. Springer-Verlag.
- [76] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '05, pages 83–94, New York, NY, USA, 2005. ACM.
- [77] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Z. Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53:89–97, July 2010.
- [78] Dennis Shasha and Marc Snir. Efficient and correct programs that share execution of parallel memory. *ACM Trans. Program. Lang. Syst.*, 10:282–312, 1988.
- [79] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design*. Wiley-IEEE Press, 1st edition, 1996.
- [80] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 260–272, New York, NY, USA, 1993. ACM.

- [81] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.
- [82] Martin Strecker. Compiler verification for C0 (intermediate report). Technical report, Université Paul Sabatier, Toulouse, April 2005.
- [83] Zehra Sura, Xing Fang, Chi L. Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, New York, NY, USA, 2005. ACM.
- [84] Zehra N. Sura. *Analyzing threads for shared memory consistency*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2004.
- [85] Jean B. Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–92, New York, NY, USA, 2010. ACM.
- [86] Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. Inferring Method Effect Summaries for Nested Heap Regions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 421–432, Washington, DC, USA, 2009. IEEE Computer Society.
- [87] Jan W. Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, New York, NY, USA, 2007. ACM.
- [88] Jaroslav Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2009.
- [89] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.
- [90] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39:131–144, June 2004.
- [91] Chi L. Wong. *Thread escape analysis for a memory consistency-aware compiler*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

Appendix A: Precise Delay Set Analysis

Consider a program with fixed number of threads in straight-line code and precise memory locations for each access.

The Model. We define a code by a tuple $\langle V, P, C \rangle$ that defines a graph where V is the set of variable accesses executed by the program; P is a set of *program edges* that represent the partial order on variable accesses V required by the program; C is a set of *conflict edges* that define the conflict relation on accesses.

One can think $\langle V, P, C \rangle$ as a *code specification* of correct execution. P defines the order of memory operations specified by the code. C specifies how the program may communicate through shared memory, and the execution of one instruction may affect another. Within the same thread, conflict edges are directed, whose order is determined by traditional dependency analysis, and enforced by any compile optimizations and hardware. Conflict edges between accesses in different threads are directed if synchronization operations enforce the execution order. In general, when threads execute concurrently, either one of two accesses related by a conflict edge may be executed first. The outcome of an execution may differ depending on which of these two accesses executes first. Figure 3 (c) shows the code specification of Figure 2.

An execution E is a proper orientation of the conflict relation C . An E is *consistent* with P , if $E \cup P$ does not contain cycles. An execution E is *correct* if E is consistent with P ; that is E can be extended to a total order such that the accesses occur in the linear sequence in the order indicated by P . Figure 3 (b) is correct, but Figure 3 (a) is not correct.

Statically one can control the order of execution of operations by introducing D . By uDv we denote that the access v is delayed until the access u is executed. Implementations should ensure that E is consistent with D . Within a thread, compiler and hardware should not move u after v by any optimizations. Synchronization operations can enforce delay across different threads, but such constraint requires expensive inter-thread coordination. Therefore, [78] address how to determine D that fulfill $D \subseteq P$. Such a delay relation always exists. Let $D = P$, then any execution must be consistent with P . But this is not interesting since it prohibits most optimizations. We are looking for the minimal delay set that affects performance as little as possible.

Minimal Delay Set. We observed that cycles in $E \cup P$ must be cycles in $C \cup P$. Suppose we enforce a delay uDv for every pair of uPv that is an edge on a cycle of $C \cup P$. Then every cycle of $C \cup P$ is also a cycle of $C \cup D$. It follows that every cycle of $E \cup P$ is also a cycle of $E \cup D$. However, implementations enforce the $E \cup D$ is acyclic. So choosing delays is sufficient in this way ensures E is correct.

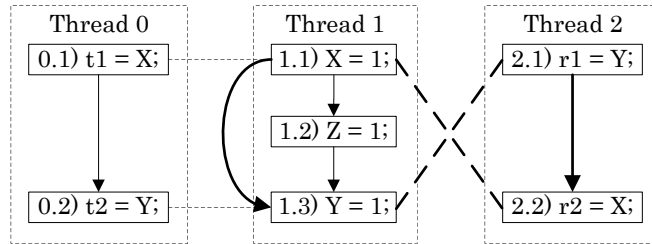


Figure 15: Critical Cycles

Actually it is not necessary to consider all cycles in $C \cup P$. In Figure 15, there are three cycles:

- (0.1, 0.2, 1.3, 2.1, 2.2, 1.1, 0.1),
- (1.1, 1.2, 1.3, 2.1, 2.2, 1.1), and
- (1.1, 1.3, 2.1, 2.2, 1.1)

First, all the cycles contain the C edges (1.3, 2.1) and (2.2, 1.1). In other words, any inconsistent executions must contain orientations from the two edges that construct

the *minimal counterexamples*. Second, (1.1, 1.3, 2.1, 2.2, 1.1) is a simplest cycle that includes the two edges. If we delay the P edges (1.1, 1.3) and (2.1, 2.2) in this cycle, none of the inconsistent executions from other cycles is possible.

Let Φ be the subsets of C edges that are acyclic and inconsistent with P , the minimal elements in Φ by set inclusion are *minimal inconsistent executions*. If the subsets of edges had a cycle, all accesses in the cycle would be in the same location. This is contradictory to intra-thread data dependency and the cache coherence protocols across threads. So we do not consider the subsets of edges that have cycles. A cycle is *critical cycle* if it is a simple cycle of $C \cup P$ and has no chords in P and C . A P edge in a critical cycle is a *critical pair*. We have the following theorem [78]:

Theorem 2 (Critical Cycles)

1. D enforces correctness iff, for every minimal inconsistent execution S , $S \cup D$ has a cycle.
2. The edges from $C - P$ in a critical cycles are a minimal inconsistent execution.

Note that it follows Theorem 1.

Therefore, rather than considering all P edges in all cycles in $C \cup P$, it is sufficient to consider the P edges in critical cycles. Moreover, a cycle in $C \cup P$ is critical iff it fulfills the following conditions [78]:

Theorem 3

1. the cycle contains at most two accesses from any thread; these accesses occur successively on the cycle.
2. the cycle contains either zero, two or three accesses to any variables; the accesses occur consecutively on the cycle. The possible configurations are `read X C write X`, `write X C read X`, `write X C write X`, or `read X C write X C read X`.

Intuitively, if two accesses from a thread were not successive, there would be a P chord; if there were consecutive accesses to the same variable that is not in these configurations, there would be a C chord. So the above conditions are sufficient.

Krishnamurthy and Yelick [46] show that the precise delay set analysis is NP-complete by reducing the Hamiltonian path problem to the problem of finding the minimal delay set, and the execution time for precise delay set analysis is exponential in the number of threads in a program. Note that the analysis is in an ideal setting with fixed number of threads in straight-line code, and precise memory locations for each access. S-level code specification [58] is proposed to represent more general programs, which models edges by state transformation functions, and detects cycles by using integer programming techniques [74]. But it is not always possible to define integer programming for state transformation, and solving integer programming is expensive.

Appendix B: Inhibiting Optimizations by Delay Set Analysis

Delay set analysis also inhibits other optimizations [50, 51]. First, we can reduce determining whether it is safe to eliminate memory accesses to checking if it reorders delay edges. For example, Table 16 shows how eliminating redundant read after write, redundant read after read and redundant write after write are equivalent to reordering.

They break SC if there is another thread $X=2;Y=1;$. In general a memory access V can be eliminated if single-threaded data flow analysis ensures that V can be eliminated by its dominator V' , and there is no statement V'' between V' and V such that (V'', V) is a delay edge.

int X = 0; int Y = 0; // Initially			
Transformation	Original	Transformed	Equivalent reordering
read after write	X = 1;	X = 1;	X = 1;
	r1 = Y;	r1 = Y;	r2 = X;
	r2 = X;	r2 = 1;	r1 = Y;
read after read	r1 = X;	r1 = X;	r1 = X;
	r2 = Y;	r2 = Y;	r3 = X;
	r3 = X;	r3 = r1;	r2 = Y;
write after write	X=1;	X=1;	X=1;
	r1=Y;	r1=Y;	X=1;
	X=1;	r2=X;	r1=Y;
	r2=X;		r2=X;

Figure 16: Eliminating memory accesses is equivalent to reordering.

Transformations typically do not introduce extra memory accesses. It is safe to insert a memory access V that already exists at V' in the original program if it is legal in term of single-threaded data flow analysis, and there is no statement V'' between V and V' such that (V'', V') is a delay edge. Table 17 illustrates that speculation reorders delay edges.

Thread 1 and 2 (Original)	Thread 1 and 2 (Transformed)
	X = 42; Y = 42;
	r1 = 42; r2 = 42;
	r3 = Y; r4 = X;
r1 = Y; r2 = X;	if (r3 != 42) { if (r4 != 42) {
X = r1; Y = r2;	r1 = r3; r2 = r4;
	X = r3; Y = r4;
	}
	}

Figure 17: Speculative memory accesses is equivalent to reordering.

Appendix C: Refining Delay Set Analysis in Java

Thread Escape Analysis and Alias Analysis. To improve the first condition in Condition 1, we first use thread escape analysis to determinate which variables can be accessed outside a thread, then apply type-based alias analysis over potentially escaped variables to find conflict edges.

Java is a strongly typed language that does not allow a reference variable refers to an object of a type that is not compatible with the declared type of the reference. The type-based analysis assumes two objects are aliased if they are of compatible type. The thread escape analysis also benefits from types to speed up analysis convergence: if a field in an object of a used-defined non-thread type escapes, then the fields in all objects of the same type escape; if a field in an object of a thread type escapes, then the field in all objects of the thread type escapes. We analyze thread fields more precisely because they are the root clause of escaping references.

Two root clauses for a reference to escape a thread are:

- The reference variable that refers to a static field escapes because static fields be shared by different objects of the class.
- When a thread $T2$ spawns a thread $T1$, the reference to $T1$ may be accessed by both $T1$ and $T2$. At this case, consider the method $M2$ of $T2$ that spawns $T1$ and the constructor $C1$ of $T1$. If the object of $T1$ can be used outside $M2$ and $C1$, then $T1$ escapes. Otherwise, all fields of $T1$ that can be accessed by $M2$ and $C1$ escape.

There are four categories of statements that cause a reference to escape because of another escaping reference: load or store to an object field, read or write access to an array element, unary or binary operations over reference, and method calls. The analysis does not distinguish elements in an array by their index values. When computing if a method can cause more objects to escape, its parameter escapes only if it may escape at some call sites of the method, rather than conservatively assuming all parameters may escape.

Experiments show that the effect of delay set analysis is highly sensitive to the accuracy of escape analysis. For the benchmark used in [83], a delay set analysis based on a trivial escape analysis that assumes almost all references that access memory escape introduces an average slowdown of 23 times, while a delay set analysis based on the above escape analysis only has an average slowdown of 10%.

Thread structure Analysis. Figure 6 (a) shows the execution order imposed by thread `start` and `join`. All instances of E in the thread spawned by `start` cannot run until all instances of A that before the `start` are completed, and must be completed before all instances of D that after the `join` that matches the `start`. All instances of B and E can run in parallel.

The analysis is similar to May-Happen-In-Parallel analysis [63]. It first conservatively matches `join` and `start` if they are of exactly the same variable intra-procedurally, and no one updates the variable along the path from `start` to `join`. Then, given any program point A , the thread structure analysis inter-threadly computes the threads that may be spawned after A —`StartAfter(A)`, the threads that may be joined before A —`JoinBefore(A)`, and the threads that may run in parallel with A —`Concurrent(A)`.

The analysis refines the second condition of Condition 1. By the union of the `Concurrent` from all program points in the `run` method of a thread, we can check if the a thread satisfies *single thread constraint*—at most one thread of that type can execute at any given time. For example, the `main` thread trivially satisfies the single thread constraint. All potential conflict edges inside such threads can be safely removed.

Moreover, if a thread `start` S is not in `StartAfter(A)`, `JoinBefore(A)` and `Concurrent(A)`, then A must be in the code of the thread spawned by S . If S is only in `JoinBefore(B)`, then we can conclude that A must occur before B , because

B cannot run until the thread terminates. Similarly, we can conclude more order relations to improve the third condition of Condition 1. Experiments show that lock analysis improves performance of 5 out of 10 programs by 70%.

Event and lock Analysis. Figure 6 (b) and (c) illustrate events and locks in Java. The execution within `synchronized(o)` is mutual exclusive by a lock for the object o . Within `synchronized(o)` a thread can be suspended at `o.wait()`, and resumed by other threads by `o.notify()`.

For a program point A , the analysis computes `LockSet(A)`—the locks that protect A , `SyncAfter(A)`—the objects for `wait` and `notify` that may occur after A , and `SyncBefore(A)`—the objects for `wait` and `notify` that may occur before A . A occurs before B if there exist a `notify N` and a `wait W` for an aliased object, such that N is in `SyncAfter(A)`, W is in `SyncBefore(A)`. We can ignore conflict edges ordered by `wait` and `notify`, and also conflict edges for program points of the same set of locks.

Experiments show that lock analysis improves performance of 3 out of 10 programs by 10%. However, event analysis is difficult to derive useful execution orders because of conservative alias analysis, flow-insensitive analysis and the timed version of `wait`.

Delay Set Analysis. When testing whether a program edge between A and B should be in the set of delay edges, instead of considering all possible individual program points that conflict A or B , the analysis considers all methods that contain accesses conflict A or B . Since the delay set analysis is based on type-based alias analysis, we can still benefit types.

First of all, for each method M , we determine the method summary information `DirectWrites(M)` that is the set of all types that M can writes directly, and `AllWrites(M)` that is the set of all types that M can writes directly or by methods M calls. Similarly we determine `DirectReads(M)` and `AllReads(M)`. Then for each node A in the code specification, we first define `ConflictMethods(A)` to be:

- $\{M | T \in \text{DirectWrites}(M)\}$ if A reads a location of type T
- $\{M | T \in \text{DirectWrites}(M) \text{ or } \text{DirectReads}(M)\}$ if A writes a location of type T
- $\{M | \exists T, T \in \text{AllReads}(N) \text{ and } \text{DirectWrites}(M) \text{ or } T \in \text{AllWrites}(N) \text{ and } \text{DirectWrites}(M) \text{ or } T \in \text{AllWrites}(N) \text{ and } \text{DirectReads}(M)\}$ if A invokes method N

Then we refine `ConflictMethods(A)` by the above analyses. If either `ConflictMethods(A)` or `ConflictMethods(B)` is empty after refinement, the edge (A,B) is not a delay edge.

Appendix D: BulkSC

To ensure that all committed chunks can be *serialized*, it is sufficient to enforce two rules [22]:

1. Updates from a chunk are not visible to other chunks until the chunk completes and commits.
2. Loads from a chunk have to return the same values as if the chunk was executed at its commit point. Otherwise, the chunk would have “observed” a changing global memory state while executing.

Figure 18 gives the BulkSC infrastructure [22].

Each process speculatively executes chunks of the maximum number (2,000) of dynamic instructions. Before executing each chunk the process i creates a checkpoint

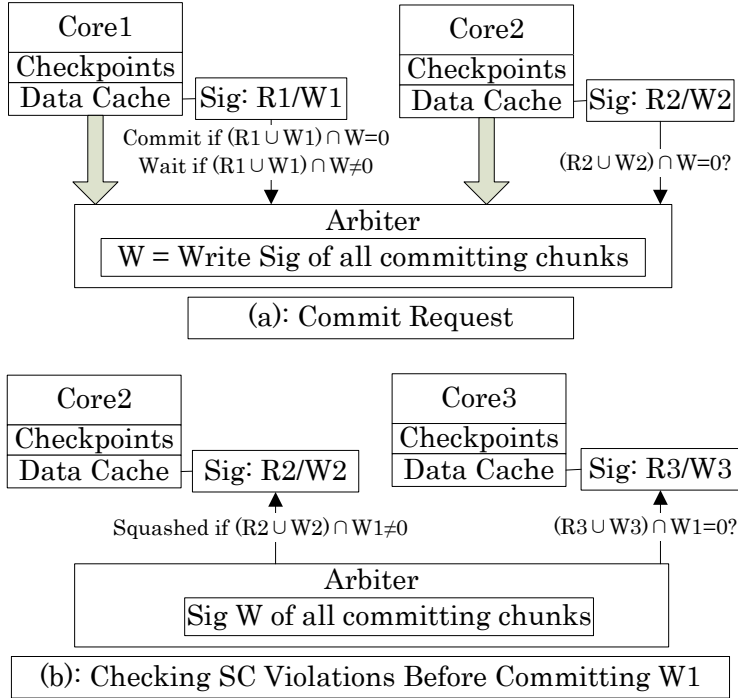


Figure 18: BulkSC Infrastructure

for all local registers. During execution the process buffers memory updates in its cache, and generates signatures R_i and W_i (hash values) for locations read and written. When chunk i is finished (reaching the maximum dynamic instruction number, or an event occurs—cache overflow, system calls, interruptions), the processor sends a commit request with R_i and W_i to the arbiter that coordinates all processes to commit chunks concurrently.

The arbiter maintains the signature W for the memory updates of all the concurrent-committing chunks. A coming request is guaranteed if R_i and W_i do not overlap with W (Figure 18 (a)), otherwise the process i can observe non-atomic executions of other committed chunks. If R_i and W_i do not overlap with W , the arbiter broadcasts W_i to all other processes that may access W_i . The process j whose R_j and W_j overlap with W_i needs to be squashed (clearing its signature, invalidating the cache, and restarting the chunk from its checkpoint). Finally the arbiter adds W_i into W to commit W_i . Before a location in W is committed completely, no process can access this location.

After receiving commit permission, the process i clears its signature, and start a new chunk. Moreover, a process can execute successive chunks concurrently if they follow intra-thread data dependency, and are committed in program order. But if a previous chunk is squashed, all following chunks must be squashed.

The system ensures that if there is no conflict, then the committed chunks are *serialized*, and preserve sequential consistency. Although it squashes executions because of coarse-grain chunk execution, it affects performance very slightly because experiments

show that the percentage of squashed instruction is low (1-2%).

Because synchronizations in a chunk do not induce any fences or constrain reordering due to atomic chunk execution, optimizations within a chunk can across synchronization operations. The paper also applies optimizations to reduce data bandwidth and false sharing: i) speculatively taking local updates not used by other processors as *private* data, and only sending Wi that are potentially shared by other processors because a processor usually updates a location in multiple chunks without any intervening access to the location by other processors; ii) sending Wi alone to the arbiter whose W is frequently empty, and sending Ri only when W is nonempty because the commit process is fast, and most shared data are temporally private. Experiments show that BulkSC offers performance comparable to release consistency.

BulkCompiler adds new instructions to BulkSC: *beginAtomic* that finishes the current chunk, creates checkpoint, takes the program counter as the entry of safe version of the code; *endAtomic* that simply lets BulkSC control the chunk; *endAtomicAndCut* that terminates the chunk before BulkSC control the execution; *etc.* Low contention chunks are created by *beginAtomic* and *endAtomic* to allow more optimizations by hardware at the end; while high contention chunks are created by *beginAtomic* and *endAtomicAndCut* to terminate the chunk to minimize squashes.

Appendix E: Formalization of Data-Race-Free

This section presents two equivalent definitions of data-race-free [17, 20, 88]: one is defined by *happen-before* order; the other is by *simultaneous conflicts*. They present different views of data-race-free—a denotational view and an operational view. In either definition we assume the language allows distinguishing between synchronization operations (lock *acquire* and *release*) and data (non-synchronization) operations.

Type 1 data-race-free. Given an execution we first observe a total order S of all synchronization operations; a W relation that defines the write a read sees. Then we define

- A release rel *synchronizes with* (SW) an acquire acq if acq can see rel in term of W ;
- A *happens before* (HB) B if they are ordered transitively by SW and P ; and
- A write is a *visible side effect* to a read if the write is the latest update of the same location the read accesses in term of HB .

We define an execution is *type 1 consistent* if

1. Each thread execution is internally consistent, given the values read from memory in the execution;
2. S is consistent with HB , and W relates a read to its *visible side effect*;
3. A thread can only release a lock after it acquires the lock, and must release the lock if it acquires a lock.

A consistent execution contains a *type 1 data race* if two conflict data operations are unordered by HB . We specify *type 1 data-race-free* as:

- If a program (on a given input) has a consistent execution with a type 1 data race, then its behavior is undefined.
- Otherwise, the program (on the same input) behaves like one of its consistent executions.

The first condition of type 1 consistent execution ensures that the implementation of type 1 data-race-free must preserve sequential semantics for each individual threads. The second condition requires synchronization operations execute in SC, and enforce execution order for conflict accesses: a release of a lock must happen before an acquire of the same lock; all data operations before a release in program order must happen before the release; all data operations after an acquire in program order cannot happen before the acquire. The third condition enables the optimization that moves data operations before an acquire after the acquire. For example,

int X = 0; // Initially	
Thread 1 (original)	Thread 2
X = 1; lock 1;	unlock 1; r = X;
Thread 1 (transformed)	Thread 2
lock 1; X = 1;	unlock 1; r = X;

Consider the program that only contains the two threads, and does not satisfy the third condition. If we reordered X = 1 and lock 1 in Thread 1, we would observe that r = 0 in Thread 2, which cannot happen in the original program. Such optimizations are valid if the third condition holds [16].

Intuitively, if a program has no type 1 consistent execution of type 1 data race, the order $S \cup HB$ can be extended to a total order over all memory operations, which constructs a sequential consistent execution. We can prove [17, 20, 88]

Theorem 4 *Type 1 data-race-free provides sequential consistent semantics to programs whose type 1 consistent executions do not contain any type 1 data race.*

Type 2 data-race-free. The type 1 data-race-free is based on the denotational semantics of a program that defines an execution as a set of thread traces with happen-before order to specify threads communication. A more intuitive definition should say that a program has no data race if there is no sequential execution such that two conflicting accesses execute in different threads simultaneously. We can define this type of data race free operationally.

Two conflicting memory accesses from different threads are *type 2 data race* if they are adjacent in a sequential consistent execution. We specify *type 2 data-race-free* as:

- If a program (on a given input) has a sequential consistent execution with a type 2 data race, then its behavior is undefined.
- Otherwise, the program (on the same input) behaves like one of its sequential consistent executions.

First, given a type 1 consistent execution of a program with *type 1 data race*, there must exist a sequentially consistent execution with *type 1 data race*. By Theorem 4, the longest prefix of the type 1 consistent execution without *type 1 data race* must have a sequential consistent execution. We have a *type 1 data race* by extending this sequential consistent execution with the first data race in the type 1 consistent execution.

Second, we can also obtain a *type 2 data race* from this sequential consistent execution with *type 1 data race*. Consider the earliest pair of accesses with *type 1 data race*. No operations between the pair prevent from ordering them to be adjacent.

Third, it is easy to see that the first *type 2 data race* in an SC execution with a total order can map to a *type 1 data race* by taking S and W as restrictions of the total order. Therefore, we have [17, 20, 88]

Theorem 5 *Type 1 data-race-free is equivalent to type 2 data-race-free.*

The discussion of this section is based on a simple version of data-race-free. The realistic language-level memory model adopts data-race-free with compromises. For example, the C++ memory model [17] allows experts to program with different consistent atomics—from sequential consistency to relaxed consistency, and extends the *happen-before* order with orders imposed by these atomics to define data races.

Appendix F: Sharpening Data Flow Information at Synchronizations

This section motivates how to sharpen the data flow information at each synchronization by siloed analysis. The intermediate representation is HSSA [24, 64] that is a variant of SSA form [25], uses χ nodes to denote *may-defs* and μ nodes to denote *may-uses*. For example, the HSSA form of Thread 1 in Figure 10 is

```

r1 = X;
   $\mu$ (Y);
lock 1;
Y = 1;
   $\mu$ (Y);
unlock 1;
  X1 =  $\chi$ (X);
  Y1 =  $\chi$ (Y);
...

```

Here, $\mu(Y)$ indicates that other threads can asynchronously use Y when `lock 1` or `unlock 1` executes; $X1 = \chi(X)$ and $Y1 = \chi(Y)$ other threads can asynchronously define X and Y when `unlock 1` executes. Given a synchronization s in a function f , the *may-defs*(s) is all variables affected by χ nodes; the *may-uses*(s) is all variables affected by μ nodes. In this case they disable data flow analysis of X and Y .

As an example we only consider how to sharpen *may-defs* (sharpening *may-uses* is similar). The most conservative *may-defs*(s) is the union of all variables s can update transitively— $W(s)$, and all variables that can reach s — $DU(s)$. If we can precisely compute $CW(s)$ that includes all variables that other threads can update when s executes, the precise *may-defs*^t(s) is

$$W(s) \cup (DU(s) \cap CW(s)) \subseteq \text{may-defs}(s)$$

It is impossible to compute $CW(s)$ precisely, so we analyze its complementary part $NC(s)$ that includes all variables that other threads cannot update when s executes.

Figure 19 illustrates the analysis space with the universal to be all variables in the program. Our goal is to identify a subset of variables in $NC(s) \cup DU(s)$ that represent

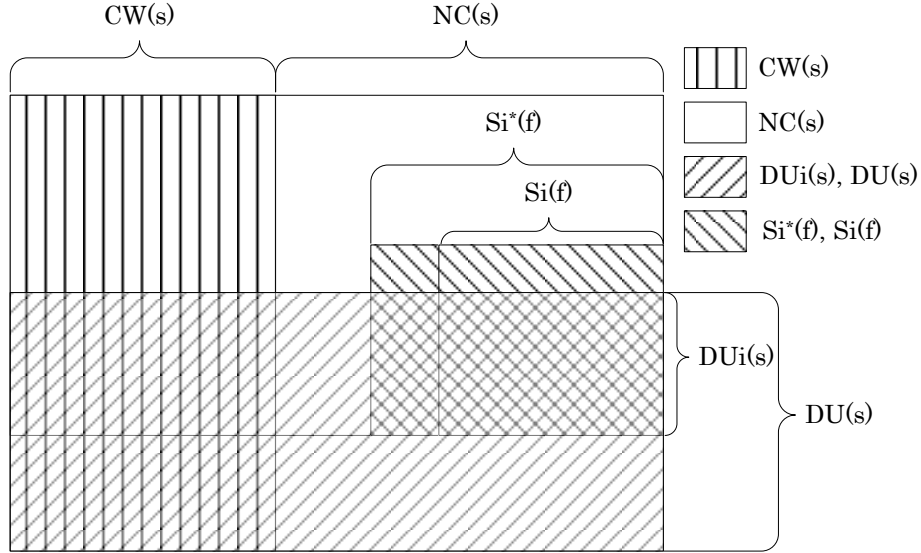


Figure 19: Sharpening may-defs by Siloed analysis

the variables that can reach s , but are free of interference from other threads. Since this work focuses on enabling optimizations for intra-procedural immediate accesses, we further reduce the interesting set to be $Si^*(f) \cup DUi(s)$ that is a subset of $NC(s) \cup DUi(s)$. Here $DUi(s)$ includes all variables accessed immediately, and can reach s ; $Si^*(f)$ includes all variables accessed immediately in f that are free of interference when any path in f executes. Note that $Si^*(f)$ conservatively considers all paths in f , rather than paths that reach s . We have

$$DUi(s) \cap CW(s) \cap Si^*(f) = \emptyset$$

$$(may-defs(s) - (DUi(s) \cap Si^*(f))) \cup W(s) \supseteq may-defs^t(s)$$

In practice we compute an approximation $Si(f)$ that is smaller than $Si^*(f)$:

$$Si(f) = (Ri(f) \cup Wi(f)) - \hat{I}i(f)$$

Here, $Ri(f)$ and $Wi(f)$ are variables accesses by f immediately; $\hat{I}i(f)$ includes all immediate accesses in f that may be interfered by other threads when any path of f executes. More precisely a variable X of memory access m in f is in $\hat{I}i(f)$ if

- There exists a function f' in which there is a conflicting access m' to a location aliasing X .
- There are no statements in f that may prevent transforming m to be running in parallel with m' .

We use alias analysis and procedural concurrency graph (PCG) to compute $\hat{I}i(f)$ (Section 3.1).

With $Si(f)$ we sharpen $may-defs(s)$ to be

$$(may-defs(s) - (DUi(s) \cap Si^*(f))) \cup W(s)$$

Intuitively the removals may enable more optimizations across synchronizations.

Appendix G: Informal Proofs of the DRFx Requirements

Condition 2 (The DRFx requirements for Compiler and Hardware)

1. *Chunk partition is valid:*
 - (a) *Each thread is partitioned into chunks with single-entry.*
 - (b) *Each synchronization is in a singleton chunk.*
 - (c) *Each system call is also in a singleton chunk, and takes only thread-local arguments.*
 - (d) *Each chunk has exactly one fence as its first instruction.*
2. *Compiler optimizations must preserve the partition of each thread and the sequential semantics of each chunk without extra shared variable accesses.*
3. *Given two conflicting accesses u and v in an execution without MM exception, if uEv (recall that E is execution order), then they are weakly-fence-ordered uWv —the closest fence executed after u in program order occurs earlier than the closest fence executed after v in program order, or u is in a committed chunk but v is in an uncommitted chunk.*
4. *If an execution raises an MM exception for two conflicting accesses u and v ,*
 - (a) *u and v must be from two uncommitted chunks in different threads;*
 - (b) *All committed chunks constitute an execution without MM exception;*
 - (c) *Accesses in committed chunks do not depend on uncommitted executions— for all wz , if wWz , then z does not depend on w by E .*
5. *Non-terminating and system exception should not block detecting MM exceptions.*

Consider a relaxed consistency hardware that provides only cache coherence—all threads see the same order of write to a location, but does not preserve sequential consistency in general. It also provides fence to enforce local writes to be globally visible, and fence execution is globally ordered. More precisely, a *well-formed relaxed execution* must satisfy that the read-after-write execution order E_{wr} is consistent with intra-thread data and control dependency Dep enforced by data dependency and fence, but $E \cup Dep$ may contain cycles.

Each well-formed relaxed execution represents a unique behavior. Suppose $E \cup Dep$ is acyclic, we can simply compute the final behavior in term of a total extension of $E \cup Dep$, although the result may be non-sequentially-consistent. In general we can only construct a total extension of $E_{wr} \cup Dep$. To compute final behavior, we record a history of writes for each memory location. A read loads the value recorded in the history indicated by E if there exists one, otherwise loads the default value from memory. At the end of execution, we update memory by the last write in each history in term of E .

An execution without MM exception implies sequential consistency. If a well-formed relaxed execution of a program is free of MM exception, Condition 2 (3) ensures that there exists an equivalent chunk-*serialized* behavior of the program. Because fences enforce commit order, $E \cup Dep$ must be acyclic. Moreover there must exist a total extension of $E \cup Dep$ that is a sequence of chunk execution ordered by

the execution of closest fences after each chunk. Otherwise the condition (3) does not hold. Given each chunk in the sequence of chunk execution, if two operations do not execute in program order, we can safely reorder them because hardware preserves data dependence in a well-formed relaxed execution. Eventually we can get an equivalent sequence of chunk execution where each chunk executes in program order—*i.e.*, a equivalent chunk-*serialized* behavior of the program.

Because Condition 2 (2) requires that the compiler preserves semantics of serial chunk, if a transformed program has a chunk-*serialized* behavior, the original program must have an equivalent chunk-*serialized* behavior. Clearly a chunk-*serialized* execution is a sequential consistent execution. Therefore, we have [56]:

Lemma 6 *If a well-formed execution of a program in the implementation that satisfies Condition 2 is free of MM exception, then the program has an equivalent sequentially consistent behavior.*

An execution with MM exception implies a data race. Consider the case that a well-formed relaxed execution of a program raises an MM exception for accesses u and v . We are proving that the program also has data race. From Condition 2 (4.a) we have u and v are in two uncommitted chunks c_u and c_v from different threads, and no fences in those two threads follow them. From Condition 2 (4.b) (4.c) we also know that the prefix execution for all committed chunks has an equivalent chunk-*serialized* behavior of the program.

The difficulty to prove that the program also has a data race (type 2) is that the conflicting accesses between u and v can be unreachable from sequentially consistent execution. If so, intuitively they must be caused by early conflicting accesses within all uncommitted chunks, and that conflicting access changes control follow. For example, before u in c_u an early read z for a control condition loads a value written by w in an uncommitted chunk c_w . Then we can check if z and w are the data race reachable sequentially consistently. We repeat the above procedure until we can find a type 2 data race. The paper claims that the procedure always terminates [55]. A trick case is:

int X = 0; int Y = 0; // Initially	
Thread 1	Thread 2
$\mathbf{r1} = \mathbf{X};$ $\mathbf{if} (\mathbf{r1})$ $\quad \mathbf{Y} = 1;$	$\mathbf{r2} = \mathbf{Y};$ $\mathbf{if} (\mathbf{r2})$ $\quad \mathbf{X} = 1;$
Is $\mathbf{r1} = \mathbf{r2} = 1$ allowed?	

Clearly the program is free of data race, because any sequentially consistent execution assigns $\mathbf{r1}$ and $\mathbf{r2}$ to be 0, and the \mathbf{if} clause does not execute. However, speculative loads \mathbf{X} and \mathbf{Y} as 1 can be self-fulfilled by executing the two \mathbf{if} clauses that indeed assign \mathbf{X} and \mathbf{Y} with 1. Suppose hardware behaved this way, the above procedure would be stuck at the conflicting pairs $(\mathbf{r1}=\mathbf{X}, \mathbf{X}=1)$ and $(\mathbf{Y}=1, \mathbf{r2}=\mathbf{Y})$. Fortunately, the acyclic $E_{wr} \cup Dep$ requirement prevents this case— E_{wr} must be consistent with control dependency. Therefore, the procedure always terminates because the new candidates must precede the old ones in term of $E_{wr} \cup Dep$.

If a transformed program has a data race, the original program should also have a data race. Consider the earliest type 2 data race in an execution of the transformed program, which is between u and v from chunks c'_u and c'_v . Because the compiler does

not transform programs across chunks, all completed chunks before this data race are *serialized*. Because this is the first data race we can remove other uncompleted chunks except c'_u and c'_v , and dynamic instructions after u and v in c'_u and c'_v . Therefore, we have an execution:

$$c'_1 \cdots c'_j \hat{c}'_u \hat{c}'_v uv$$

Here, c'_i is dynamic chunk; \hat{c}'_u are dynamic instructions from c'_u before u ; \hat{c}'_v are dynamic instructions from c'_v before v ; \hat{c}'_u and \hat{c}'_v have no conflict. Due to Condition 2 (2), there must exist an execution of the original program:

$$c_1 \cdots c_j$$

Such that $c_1 \cdots c_j$ is sequentially consistent, and leads to the same behavior as $c'_1 \cdots c'_j$.

Consider c_u and c_v in the original program that correspond to c'_u and c'_v . Because Condition 2 (2) requires the locations read and written by the original program be a superset of the transformed one, $c_1 \cdots c_j c_u$ and $c_1 \cdots c_j c_v$ must reach u and v respectively. Consider the execution

$$c_1 \cdots c_j \hat{c}_u$$

where \hat{c}_u denotes dynamic instructions from c_u such that \hat{c}_u is ready to execute u . However, since \hat{c}_u updates more values than \hat{c}'_u , if the extra updates by \hat{c}_u changes control flow of c_v , $c_1 \cdots c_j \hat{c}_u c_v$ does not reach v . If that is the case, c_v must read the write from \hat{c}_u —that is a type 1 data race. Otherwise, if v is still reachable, we have a type 2 data race. Note that Condition 2 (2) also inhibits speculations.

Therefore, we have [56]:

Lemma 7 *If a well-formed relaxed execution of a program in the implementation that satisfies Condition 2 raises an MM exception, then the program has a data race.*

Lemma 6 7 establishes the DRF and soundness properties for DRFx. Moreover, Condition 2 (1.c) and (5) ensure the safety property.