Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results

PEIZHAO OU, University of California, Irvine, USA
BRIAN DEMSKY, University of California, Irvine, USA

Eliminating so-called "out-of-thin-air" (OOTA) results is an open problem with many existing programming language memory models including Java, C, and C++. OOTA behaviors are problematic in that they break both formal and informal modular reasoning about program behavior. Defining memory model semantics that are easily understood, allow existing optimizations, and forbid OOTA results remains an open problem. This paper explores two simple solutions to this problem that forbid OOTA results. One solution is targeted towards C/C++-like memory models in which racing operations are explicitly labeled as atomic operations and a second solution is targeted towards Java-like languages in which all memory operations may create OOTA executions. Our solutions provide a per-candidate execution criterion that makes it possible to examine a single execution and determine whether the memory model permits the execution. We implemented and evaluated both solutions in the LLVM compiler framework. Our results show that on an ARMv8 processor the first solution has no overhead on average and a maximum overhead of 6.3% on 43 concurrent data structures, and that the second solution has an average overhead of 3.1% and a maximum overhead of 17.6% on the SPEC CPU2006 C/C++ benchmarks.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; *Compilers*;

Additional Key Words and Phrases: concurrency, memory models, compilers

1 INTRODUCTION

Programming language memory models define the semantics of loads and stores in a multi-threaded program. Most programming language memory models guarantee sequential consistency for race free programs [Adve and Hill 1990]. This guarantee is fragile — a single data race voids the sequential consistency guarantee for the entire program. Indeed, programs with data races have undefined semantics under the C/C++11 memory model. While researchers have explored providing stronger guarantees to racy programs [Marino et al. 2011; Singh et al. 2012], these approaches may require hardware support to achieve competitive performance.

Memory models assign undefined semantics to racy programs as data races violate assumptions made by compiler optimizations. For existing compilers, assigning meaningful semantics to racy programs is extremely complicated. The language semantics must capture behaviors that arise from both compiler and processor optimizations. While the C and C++ memory models do not even attempt to assign semantics to such programs, Java is intended to support the safe execution of untrusted code. Thus, Java must ensure safety for racy programs and the Java Memory Model attempts to assign semantics to such programs.

A similar situation exists in C and C++. While the C and C++ languages do not define the behaviors of racy programs, they do include support for special `atomic` operations. These atomic operations

Authors' addresses: Peizhao Ou, Electrical Engineering and Computer Science, University of California, Irvine, USA, peizhaoo@uci.edu; Brian Demsky, Electrical Engineering and Computer Science, University of California, Irvine, USA, bdemsky@uci.edu.

2475-1421/2018/11-ART136
https://doi.org/10.1145/3276506

provide support for developers to make tradeoffs between the ordering guarantees provided and the overhead incurred. The weakest guarantee provided is specified using the `memory_order_relaxed` memory order. Such operations only enforce coherency and cannot be used to implement synchronization. Effects of compiler optimizations can in some cases be visible to relaxed atomics. Relaxed atomics in C and C++ have qualitatively similar behaviors to memory accesses in Java.

1.1 The Problem

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = 42;`

Fig. 1. "Real Example". With `x=y=0` initially, can `r1=r2=42`?

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Fig. 2. Canonical Out-of-Thin-Air Example. With `x=y=0` initially, can `r1=r2=42`?

A key challenge in programming language memory models is prohibiting out-of-thin-air behaviors or satisfaction cycles. This problem is well known [Batty et al. 2013, 2015a; Manson et al. 2005; Pichon-Pharabod and Sewell 2016] and has been described in detail [Boehm and Demsky 2014]. Figure 1 presents an execution that real processors produce. A processor might reorder the store of 42 to `x` in Thread 2, Thread 1 can then read the value 42 from `x` and store it in `y`, and finally Thread 2 can load 42 from `y`.

Figure 2 presents an out-of-thin-air example with the same reads-from[1] relationship between loads and stores as the previous example. If both loads read from the subsequent stores, the C and C++ memory model formalism admits an execution in which `r1=r2=42` (or any other value), conjuring the value of 42 "out of thin air". The key difference between these two examples is that in the problematic example, the stores depend on the previous loads.

Note that if we directly write these examples in assembly, no processor will produce the problematic results in Figure 2. Processors preserve a notion of dependency — a processor core will not make a speculative store visible to other cores. Compilers in general do not preserve dependencies — compiler optimizations can easily optimize away dependencies (e.g., an `if` statement in which both branches store the same value to the same variable). Compiler optimizations conspire with relaxed hardware implementations to create the problem.

Although there is agreement that Figure 2 represents OOTA behavior, the precise definition of OOTA is disputed. Consider the example (from Boehm and Demsky [2014]) shown in Figure 3. An optimizing compiler may discover that the load `r1=x` in Thread 1 will always return the value of 42 no matter whether the conditional branch is taken or not. Hence it can replace the store `y=r1` with `y=42`[2], and then through the same reads-from relationship between loads and stores as Figure 2, the execution in which `r1=r2=42` is allowed. While some researchers argue that this is OOTA behavior and should be disallowed, other researchers may argue that this is legitimate behavior because the value of 42 in this example arises from the untaken branch.

1.2 Consequences

As previously noted [Batty et al. 2013; Boehm and Demsky 2014], allowing out-of-thin-air results is disastrous. Serious issues of allowing OOTA results include:

(1) **OOTA Results Break Formal Modular Reasoning:** As noted by Batty et al. [2013], OOTA executions can break certain types of compositional reasoning about programs. In particular,

[1]We use the term *reads-from* or *rf* hereafter to refer to the relation that maps stores to the loads that retrieve the stored value. For example, "$(s, l) \in rf$" means that load l reads its value from store s.

[2]In fact, this optimization is implemented in GCC and Clang/LLVM for non-atomic memory accesses.

Thread 1	Thread 2

`r3 = x;`
`if (r3 != 42)`
` x = 42;`
`r1 = x;`
`y = r1;`
	`r2 = y;`
	`x = r2;`

Fig. 3. With `x=y=0` initially, can `r1=r2=42`? While there seems to be general consensus that the execution shown in Figure 1 is not OOTA behavior and that the execution shown in Figure 2 is OOTA behavior, researchers may not have consensus on whether the execution in this example is OOTA behavior.

even if the guarantees provided by each component satisfy the assumptions of all other components, OOTA results allow executions in which two components mutually violate their own guarantees and thus violate the assumptions of the other component (circularly justifying their violation of their guarantees).
In languages that allow OOTA results, compositionality requires proving that each component in a composition is non-interfering (i.e., that it does not write to the memory locations of other components). Indeed, some analyses and tools that are based on the C/C++ memory model either simply assume the non-existence of OOTA behavior or require a stronger version of the C/C++ memory model that prohibits OOTA behavior [Kokologiannakis et al. 2017; Meshman et al. 2015; Norris and Demsky 2013; Ou and Demsky 2015].

(2) **OOTA Results Break Informal Modular Reasoning:** While developers rarely formally prove their software correct, OOTA results can even break informal reasoning about programs. Indeed, for many programs it may be necessary to avoid including accesses to relaxed atomics in the code base. For example, simply exposing an interface to relaxed stores to a virtual machine interpreter is likely sufficient to allow OOTA results that can produce arbitrary executions.

(3) **OOTA Results Can Affect Race-Free Computations:** OOTA results can induce race-free computations to produce surprising results. The following example is courtesy of Sarita Adve:

Thread 1	Thread 2
if (x) y=1;	if (y) x=1;

Even with `x=y=0` initially, OOTA results allow this computation to set both `x` and `y` to 1.

1.3 Potential Solutions

Researchers have proposed several different basic approaches to solving the out-of-thin-air problem. We next discuss the different basic approaches. These approaches fall into two primary categories. The first category attempts to eliminate the problem by changing the memory model specification without significant changes to the compiler and the second category attempts to eliminate the problem by providing stronger guarantees.

1.3.1 Approaches that Primarily Affect the Language Specification.

Precisely Specifying the Effects of Existing Optimizations: Forbidding OOTA behaviors by precisely specifying the effects of existing optimizations is one potential solution. This approach is tempting as it incurs no runtime overheads and requires no modifications to either compilers or processors. Researchers have proposed event-structures-based memory models [Jeffrey and Riely 2016; Pichon-Pharabod and Sewell 2016] that were later shown not to be compiled to ARM without additional fences in some cases [Kang et al. 2017].

Attempts at forbidding OOTA executions by precisely specifying the effects of optimizations have to date yielded complicated memory models. Indeed, Batty et al. [2015b] show that there is no per-candidate-execution solution to the problem. For example, Kang et al. [2017] propose a memory model based on a semantics that claims to resolve the OOTA problem, and the proof of its compilation correctness has been shown by Podkopaev et al. [2017]. While this approach can potentially solve the OOTA problem, a less complex but slightly stronger memory model may still be desirable if the overhead is acceptably small.

The Java Memory Model also attempted this approach [Manson et al. 2005], but the approach has since shown to be unsound with respect to standard compiler optimizations [Ševčík and Aspinall 2008]. Moreover, the JMM is extremely complicated for both compiler developers and application developers to understand. It is also complicated to use the constraints placed on OOTA executions by the JMM to prove correctness properties for concurrent programs. Indeed, merely verifying whether the JMM allows a given concrete execution is undecidable [Botinčan et al. 2010].

Case-Based Approaches: Another approach is to constrain the usage of atomics to specific cases and then provide simple semantics for those cases. The most well known example of this approach is the classic “data race freedom implies sequential consistency” memory model used by most multi-threaded programming languages [Adve and Hill 1990]. In this model, if there is no sequentially consistent execution with a data race, then the system guarantees that all executions are sequentially consistent. Other work enumerates common use cases for relaxed atomics and provides semantics for those use cases [Sinclair et al. 2017].

There are two basic challenges with this approach: (1) memory model developers must ensure that the cases handled cover the important usage scenarios and (2) bugs can produce behaviors that fall outside the well defined cases and then the memory model may provide little or no guarantees as to the program’s behaviors.

1.3.2 *Approaches that Provide Stronger Guarantees.*

Approach 1: Forbid Load-Store Reordering: A conceptually simple approach to forbidding OOTA executions is to forbid load-store reordering [Boehm and Demsky 2014; Lahav et al. 2017]. Precisely, the memory model requires that **sequence-before** ∪ **reads-from** is acyclic. This greatly simplifies the memory model for both compiler and application developers, but can potentially incur significant runtime costs.

Implementing this approach requires changes to compiler optimizations and the potential generation of a fence-like operation. The cost of this approach depends on both the details of the memory model and the hardware architecture. While x86 processors already provide this behavior without requiring fences, architectures like ARM or PowerPC may incur higher overheads. We believe that this approach is likely to be acceptable for memory models like C/C++11 as it only affects relaxed loads and stores[3]. However, this approach affects all loads and stores in Java programs, and thus is likely to be less acceptable in the context of Java.

Approach 2: Preserve Dependencies: Earlier work suggested but did not implement one potential approach to forbidding OOTA executions — require the compiler to preserve a simple, syntactic notion of dependency [Boehm and Demsky 2014]. Effectively, this approach provides a syntactic definition for a **dependency** relationship and then requires that **dependency** ∪ **reads-from** is acyclic. This is a strictly weaker guarantee than the previous approach. It is worth noting that the Linux kernel memory model does not have out-of-thin-air values because it essentially respects

[3]Under C/C++11, non-atomic loads and stores cannot race, or the program has no semantics. Thus, reordering cannot be observed.

syntactic dependencies as hardware does [Alglave et al. 2018]. McKenney et al. [2016] have proposed an approach based on preserving semantic dependencies rather than syntactic dependencies. For example, they allow reducing an expression with syntactic dependency to a constant if it is known to always result in that constant value (e.g., reducing "`r1=x*0`" to "`r1=0`"). This trades off the simplicity of the memory model specification for the degree of compiler optimizations that are allowed. In this paper, we explore an approach of preserving syntactic dependencies.

While this approach does not require the addition of any extra fence instructions, it does constrain the optimizations performed by the compiler. The primary concern with this approach is that the overheads were previously unknown and feared to be high.

1.4 Contributions

This paper makes the following contributions:

- **A dependency-based approach to forbidding OOTA:** It presents a dependency-preserving approach to forbid out-of-thin-air executions.
- **An approach to preserving load-store ordering to forbid OOTA:** It presents an approach to preserving load-store ordering to extend C/C++-like language memory models to forbid out-of-thin-air executions.
- **Implementations of both approaches in the LLVM compiler:** It presents implementations of both approaches to forbidding OOTA in the LLVM compiler.
- **Evaluation:** It evaluates the overhead of both approaches on an ARMv8 processor. It shows that the average overhead of preserving dependencies relative to compiling with full optimizations (-O3) is 3.1% on the SPEC CPU benchmarks for a prototype implementation that is likely amenable to further optimizations. It shows that under our experimental setting preserving load-store ordering has no overhead on average and worst-case overhead of 6.3% on concurrent data structure benchmarks.

The remainder of this paper is structured as follows: Section 2 presents our extensions to the programming language memory model. Section 3 describes our approach to extending the LLVM compiler to preserve our dependency notion. Section 4 presents our approach to extending the LLVM compiler to preserve load-store ordering. Section 5 evaluates both approaches. Section 6 presents related work and Section 7 concludes.

2 MEMORY MODEL EXTENSIONS THAT DISALLOW OOTA BEHAVIORS

In this section, we first discuss a dependency-preserving memory model that disallows out-of-thin-air behaviors by defining a notion of dependency and preserving it. While it is desirable that we formalize this memory model in the context of an existing programming language, e.g., C/C++ or Java, both the C/C++ and Java languages are complex and the formalization would exceed the scope of this paper. Therefore, we introduce a simple language that captures the core features of an imperative programming language in Section 2.1. Then, we define the notion of dependency based on this language and describe how we preserve such dependencies in Section 2.2. Last, we discuss a load-store-order-preserving memory model that prevents out-of-thin-air behaviors in Section 2.3.

2.1 The Language

Figure 4 presents the core syntax of our language[4]. To simplify the illustration, we only support one value type — *numerals*. When we read from or write to a global variable/memory location, we explicitly use the **load** or **store** keywords to distinguish them from assignments to local variables. Our language is based on the static single assignment (SSA) form [Rosen et al. 1988], where we can

[4]The toy language here is purely for the purpose of simplifying illustration, and our actual implementation is for C/C++.

have phi functions (ϕ) at the end of an `if/else` block or in the beginning of a `while` loop's header. The syntax starts with *Program*, which has an optional declaration of global variables followed by a list of function definitions. Figure 5 shows an example code snippet written in our language. In this example, we declare three global variables `x`, `y`, and `z` and perform load/store from/to these global locations in line 3, 6 and 10, respectively. Line 9 is the ϕ function for the `if/else` conditional branch, meaning that if the condition "`r2==0`" is `true`, local variable `r5` will be assigned with `r3`; otherwise, it will be assigned with `r4`.

Var	::=	Variable Names
Func	::=	Function Names
Num	::=	Constant Numerals
op_b	::=	C-like binary operators
op_u	::=	C-like unary operators
Expr	::=	*Num* | *Var* | *FuncCall* | *Expr* op_b *Expr*
		| op_u*Expr* | **load** *Expr*
VarList	::=	*Var* (“,” *Var*)*
ExprList	::=	*Expr* (“,” *Expr*)*
Phi	::=	*Var* “=” “ϕ” “(” *Var* “,” *Var* “)”
PhiList	::=	(*Phi* (“;” *Phi*)*)?
Stmt	::=	**skip** | *Stmt* “;” *Stmt* | *Var* “=” *Expr* |
		store *Expr* “,” *Expr* | *FuncCall* |
		return *Expr*? |
		if *Expr* **then** *Stmt* **else** *Stmt* **fi** *PhiList* |
		while *PhiList* *Expr* **do** *Stmt* **od**
FuncDef	::=	*Func* “(” (*VarList*)? “)”
		begin *Stmt* “;” **end**
FuncCall	::=	*Func* “(” *ExprList*? “)”
Program	::=	(*VarList* “;”)? *FuncDef*+

Fig. 4. The core syntax of our language

```
1: x, y, z;
2: main() begin
3:   r1 = load &x;
4:   r2 = r1 * 0;
5:   if r2 == 0 then
6:     store &y, 1;
7:     r3 = 0
8:   else r4 = 1 fi
9:   r5 = ϕ(r3, r4);
10: store &z, r5;
11:end
```

Fig. 5. An example code snippet written in our language

2.2 Language-Level Dependency Notion

(a) Data dependency	(b) Explicit control dependency	(c) Address dependency	(d) Implicit control dependency

`r1 = load &x;`
`r2 = r1 * 0;`
`store &y, r2;`
	`r1 = load &x;`
	`if r1 != 0 then`
	` store &y, 1`
	`else`
	` skip`
	`fi;`
		`r1 = load &x;`
		`store r1, 0;`
		`r2 = load &z;`
		`store &y, r2;`
			`r1 = load &x;`
			`if r1 != 0 then`
			` store &y, 1`
			`else`
			` store &z, 0`
			`fi;`
			`r2 = load &z;`
			`store &y, r2;`

Fig. 6. Does the last store to y depend on the first load "`r1 = load &x`" in each of theses examples? Assume z=1 before each execution.

While there is general agreement about extreme examples that conjure new values and exhibit out-of-thin-air behavior (e.g., the example shown in Figure 2), there is no consensus on the exact definition of an out-of-thin-air execution. In this paper, we broadly define an out-of-thin-air execution to be any execution in which the behavior of an operation is circularly involved in causally justifying its own behavior. To prohibit such executions, we define a conservative syntax-based notion of dependency that maps loads in a thread to all stores in the thread whose behavior the loads may affect. We also provide a proof sketch of a theorem about the causality of executions in our memory model in Appendix C. The precise definition of the notion of dependency for our language with the operational semantics is shown below:

$$\text{Const.Expr:}\ \frac{}{\langle const, V, dep, D, PC, FD\rangle \rightarrow \langle\langle const, \emptyset\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Var.Expr:}\ \frac{}{\langle var, V, dep, D, PC, FD\rangle \rightarrow \langle\langle V[var], D[var]\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Unary.Expr:}\ \frac{\langle E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}{\langle op_u\ E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle op_u\ \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Binary.Expr:}\ \frac{\begin{array}{c}\langle E_1, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_1, \mathcal{D}_1\rangle, V', dep', D', PC', FD'\rangle\\ \langle E_2, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle \mathcal{V}_2, \mathcal{D}_2\rangle, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\langle E_1\ op_b\ E_2, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_1\ op_b\ \mathcal{V}_2, \mathcal{D}_1 \cup \mathcal{D}_2\rangle, V'', dep'', D'', PC'', FD''\rangle}$$

$$\text{Load.Expr:}\ \frac{\langle Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Addr}}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle \quad \mathcal{V}_{\text{load}} = \text{load}(\mathcal{V}_{\text{Addr}})}{\langle \text{load}\ Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{load}}, \mathcal{D} \cup \{fresh\_load\}\rangle, V', dep', D', PC', FD'\rangle}$$

$$\text{Assignment:}\ \frac{\langle E, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle}{\langle var = E, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V'[var := \mathcal{V}], dep', D'[var := \mathcal{D}], PC', FD'\rangle}$$

$$\text{Store:}\ \frac{\begin{array}{c}\langle Addr, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Addr}}, \mathcal{D}_{\text{Addr}}\rangle, V', dep', D', PC', FD'\rangle\\ \langle Val, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle \mathcal{V}_{\text{Val}}, \mathcal{D}_{\text{Val}}\rangle, V'', dep'', D'', PC'', FD''\rangle \quad s := \text{store}(\mathcal{V}_{\text{Addr}}, \mathcal{V}_{\text{Val}})\end{array}}{\begin{array}{l}\langle \text{store}\ Addr, Val, V, dep, D, PC, FD\rangle \rightarrow\\ \quad\langle skip, V'', dep'' \cup ((\mathcal{D}_{\text{Addr}} \cup \mathcal{D}_{\text{Val}} \cup PC'' \cup FD'') \times \{s\}), D'', PC'', FD'' \cup \mathcal{D}_{\text{Addr}}\rangle\end{array}}$$

$$\text{Composition.Skip:}\ \frac{}{\langle skip; S, V, dep, D, PC, FD\rangle \rightarrow \langle S, V', dep', D', PC', FD'\rangle}$$

$$\text{Composition.Left}\ \frac{\langle S_1, V, dep, D, PC, FD\rangle \rightarrow \langle S_1', V', dep', D', PC', FD'\rangle}{\langle S_1; S_2, V, dep, D, PC, FD\rangle \rightarrow \langle S_1'; S_2, V', dep', D', PC', FD'\rangle}$$

$$\text{Phi.Taint:}\ \frac{}{\langle\langle v, \mathcal{D}\rangle, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V, dep, D[v := (D[v] \cup \mathcal{D})], PC, FD\rangle}$$

$$\text{assign\_phi}(v = \phi(v_0, v_1); phiList, \mathcal{V}_P) \Rightarrow v = v_{\mathcal{V}_P}; \text{assign\_phi}(phiList, \mathcal{V}_p)$$

$$\text{assign\_phi}(\epsilon, \mathcal{V}_P) \Rightarrow \epsilon \qquad\qquad \text{assign\_phi}(;, \mathcal{V}_P) \Rightarrow \epsilon$$

$$\text{taint\_phi}(v = \phi(v_0, v_1); phiList, \mathcal{D}) \Rightarrow \langle v, \mathcal{D}\rangle; \text{taint\_phi}(phiList, \mathcal{D})$$

$$\text{taint\_phi}(\epsilon, \mathcal{V}_P) \Rightarrow \epsilon \qquad\qquad \text{taint\_phi}(;, \mathcal{V}_P) \Rightarrow \epsilon$$

$$\text{If.True:}\ \frac{\begin{array}{c}\langle cond, V, dep, D, PC, FD\rangle \rightarrow \langle\langle true, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle\\ \langle S_1, V', dep', D', PC' \cup \mathcal{D}, FD'\rangle \rightarrow \langle skip, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\begin{array}{l}\langle \text{if}\ cond\ \text{then}\ S_1\ \text{else}\ S_2\ \text{fi}\ phi, V, dep, D, PC, FD\rangle \rightarrow\\ \quad\langle \text{assign\_phi}(phi;, 0)\ \text{taint\_phi}(phi;, \mathcal{D})\ skip, V'', dep'', D'', PC,\\ \qquad FD' \cup \{l \mid l \in \mathcal{D} \land \text{hasReachableStore}(S_2)\}\rangle\end{array}}$$

$$\text{If.False:}\ \frac{\begin{array}{c}\langle cond, V, dep, D, PC, FD\rangle \rightarrow \langle\langle false, \mathcal{D}\rangle, V', dep', D', PC', FD'\rangle \\ \langle S_2, V', dep', D', PC' \cup \mathcal{D}, FD'\rangle \rightarrow \langle skip, V'', dep'', D'', PC'', FD''\rangle\end{array}}{\begin{array}{l}\langle \text{if } cond \text{ then } S_1 \text{ else } S_2 \text{ fi } phi, V, dep, D, PC, FD\rangle \rightarrow \\ \quad \langle \text{assign\_phi}(phi;, 1)\ \text{taint\_phi}(phi;, \mathcal{D})\ skip, V'', dep'', D'', PC, \\ \quad\quad FD'' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S_1)\}\rangle\end{array}}$$

$$\text{While.Taken:}\ \frac{\begin{array}{c}\langle \text{assign\_phi}(phi;, 0)\ skip, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle true, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle S, V'', dep'', D'', PC'' \cup \mathcal{D}, FD''\rangle \rightarrow \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle \text{while } phi\ cond \text{ do } S \text{ od}, V, dep, D, PC, FD\rangle \rightarrow \langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC\rangle, \\ \quad V''', dep''', D''', PC''', FD'''\rangle\end{array}}$$

$$\text{While.Untaken:}\ \frac{\begin{array}{c}\langle \text{assign\_phi}(phi;, 0)\ skip, V, dep, D, PC, FD\rangle; \rightarrow \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle false, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle \text{taint\_phi}(phi;, \mathcal{D})\ skip, V'', dep'', D'', PC'', FD''\rangle \rightarrow \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle \text{while } phi\ cond \text{ do } S \text{ od}, V, dep, D, PC, FD\rangle \rightarrow \\ \quad \langle skip, V''', dep''', D''', PC, FD''' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S)\}\rangle\end{array}}$$

$$\text{Loop.Taken:}\ \frac{\begin{array}{c}\langle \text{assign\_phi}(phi;, 1)\ skip, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle true, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle S, V'', dep'', D'', PC'' \cup \mathcal{D}, FD''\rangle \rightarrow \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V, dep, D, PC, FD\rangle \rightarrow \\ \langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V''', dep''', D''', PC''', FD'''\rangle\end{array}}$$

$$\text{Loop.Untaken:}\ \frac{\begin{array}{c}\langle \text{assign\_phi}(phi;, 1)\ skip, V, dep, D, PC, FD\rangle \rightarrow \langle skip, V', dep', D', PC', FD'\rangle \\ \langle cond, V', dep', D', PC', FD'\rangle \rightarrow \langle\langle false, \mathcal{D}\rangle, V'', dep'', D'', PC'', FD''\rangle \\ \langle \text{taint\_phi}(phi;, \mathcal{D})\ skip, V'', dep'', D'', PC'', FD''\rangle \rightarrow \langle skip, V''', dep''', D''', PC''', FD'''\rangle\end{array}}{\begin{array}{l}\langle\langle \text{loop } phi\ cond \text{ do } S \text{ od}, PC_{old}\rangle, V, dep, D, PC, FD\rangle \rightarrow \\ \quad \langle skip, V''', dep''', D''', PC_{old}, FD''' \cup \{l \mid l \in \mathcal{D} \wedge \text{hasReachableStore}(S)\}\rangle\end{array}}$$

We formalize the program execution state as the tuple $\delta = \langle N, V, dep, D, PC, FD\rangle$, where N represents a computational node (e.g., an expression or statement), V represents a mapping from an expression to its concrete value, dep represents a dependency set, which is a subset of the Cartesian product of the load set and store set in an execution. For example, "$(l, s) \in dep$" means that store s depends on load l; D represents a dependency mapping from an expression to the set of loads the expression depends on, PC represents the set of loads on which the current instruction has explicit control dependency, and FD represents the set of loads on which future stores should depend. Essentially, the rules of our semantics focus on recording which loads an expression or statement depends on in each step of the execution, and when we finish executing the program, the final result is recorded in dep — the complete dependency relation between loads and stores in the execution. The details of dependency rules follow:

(1) Expressions: In general, an expression has data dependency on its subexpressions, meaning that the expression depends on whatever loads its subexpressions depend on. In our operational semantics, an expression E can be reduced to a pair $\langle\mathcal{V}, \mathcal{D}\rangle$, in which $\mathcal{V}$ represents the concrete value to which E is evaluated, and $\mathcal{D}$ represents the set of loads E depends on. The *Const.Expr* rule means a constant numeral is evaluated to itself and does not depend on any loads; the *Var.Expr* rule means a variable var retrieves its concrete value recorded in V and its dependency set recorded in D; the *Unary.Expr* and *Binary.Expr* rules are specifically for unary and binary expressions, respectively; the *Load.Expr* rule means that a load expression has data dependency on its address. It is important to note that in the dependency relation we define, loads are the sources and stores are the sinks. More specifically, given a load instruction l and its address *Addr* (which depends on $\mathcal{D}$), l depends on the union of $\mathcal{D}$ and l itself, denoted as $fresh\_load$ (since l becomes the source in the dependency relation after we execute the load). In the *Load.Expr* rule, "$\mathcal{V}_{load} = \text{load}(\mathcal{V}_{Addr})$" means reading the value from address $\mathcal{V}_{Addr}$ and assigning it to $\mathcal{V}_{load}$.

(2) Assignments: For an assignment statement "*var* = *E*", the left-hand side variable *var* has data dependency on the right-hand side expression *E*. Thus, the *Assignment* rule assigns the concrete value of *E* to *var* and also passes the dependencies of *E* to *var*.

(3) Stores: For a store statement *s*, "`store Addr, Val`", where the address `Addr` has dependency on load set $\mathcal{D}_{Addr}$ and the storing value has dependency on load set $\mathcal{D}_{Val}$, *s* has data dependency on $\mathcal{D}_{Addr} \cup \mathcal{D}_{Val}$. For example, in 6 (a), since the storing value `r2` has data dependency on "`r1 = load &x`", so store "`store &y, r2`" depends on "`r1 = load &x`". In addition, *s* should depend on the set of loads on which it has explicit control dependency, which in our operational semantics is recorded in *PC*. For example, in 6 (b), store "`store &y, 1`" depends on "`r1 = load &x`" because of explicit control dependency.

However, it is not sufficient to only consider explicit data and control dependencies. Consider the question of whether the last store "`store &y, r2`" depends on the first load "`r1 = load &x`" in Figure 6 (c). At first glance, it may appear to be that "`store &y, r2`" only depends on "`r2 = load &z`" but is independent of "`r1 = load &x`" since "`store &y, r2`" does not have an explicit data dependency or control dependency on "`r1 = load &x`". However, if we consider the question of what value the memory location `y` will hold at the end of the execution (assuming `z=1` before each execution), we can see that the answer becomes either the value `1` or `0` depending on whether `r1` points to the memory address of `z`, which means "`store &y, r2`" actually depends on "`r1 = load &x`". Technically speaking, if we can tell by some static analysis (e.g., some sort of points-to analysis) that `r1` always points to a different memory address than `z`, then "`store &y, r2`" does not depend on "`r1 = load &x`". However, if we rely on such analysis in our rules, we will end up with an extremely complicated dependency notion; moreover, the fact that C/C++ allows pointer arithmetics and pointer conversions would further complicate the dependency notion because the precise addresses of loads and stores may not be always known. Hence, instead of incorporating the details of the points-to analysis in our definition of dependency, we take a very conservative approach. In our semantics, if the address of a store depends on some load, then we require that all subsequent stores also depend on that load. We refer to this type of dependency as an *address dependency*. Back to the example shown in Figure 6 (c), we simply conservatively say that the store "`store &y, r2`" depends on the load "`r1 = load &x`", even if the compiler knows `r1` will never point to the address of `z`. As a result, in our implementation, if the compiler wishes to reorder a store *s*′ ("`store &y, r2`" in this example) up above another store *s* ("`store r1, 0`" in this example), it has to make *s*′ depend on any loads that the address of *s* depends on ("`r1 = load &x`" in this example). Appendix A also shows an out-of-thin-air example which involves address dependencies.

Consider the example shown in Figure 6 (d), in which we need to answer the same question of whether the last store "`store &y, r2`" depends on the first load "`r1 = load &x`". Similar to the example shown in Figure 6 (c), the store "`store &y, r2`" does not have a data dependency or explicit control dependency on "`r1 = load &x`"; however, given `z=1` initially, the value that the memory location `y` will hold can actually be value `1` or `0`, depending on whether the condition "`r1 != 0`" is true. The essential reason why we have such dependency is that in an untaken branch there exists a store (i.e., "`store &z, 0`" in the `else` branch) which overwrites a memory location that is later read. Similar to address dependency, a fine-grained definition of this type of dependency would require the introduction of a program analysis and complicate our dependence notion. Instead, we take a conservative approach by stating that if the condition of a control flow block (i.e., an `if/else` block or `while` loop) depends on a load, and the untaken branch has a syntactically reachable store, then all subsequent stores after the conditional branch also depend on that load. We refer to this type of dependency as *implicit control dependency*.

We can see that the address dependency set and implicit control dependency set share two commonalities: (1) along with the execution of a program, both dependency sets will only be augmented by adding more loads; and (2) whenever a future store statement is executed, we must ensure that the future store depends on the loads in these two sets. Hence, our operational semantics uses *FD* to record the union of address dependency and implicit control dependency, on which all future stores depend.

(4) Phi (ϕ) functions: A phi function "`v=`ϕ`(`$\texttt{v}_0$`,`$\texttt{v}_1$`)`" with respect to condition *cond* is essentially an assignment statement that selects its right-hand side value associated with *cond*, whether *cond* is from an `if/else` branch or a `while` loop. For an `if/else` branch, $\texttt{v}_0$ comes from the `if` branch, and $\texttt{v}_1$ comes from the `else` branch. For a `while` loop, $\texttt{v}_0$ comes from outside the loop, and $\texttt{v}_1$ comes from inside the loop. For example, if the phi function is associated with an `if/else` branch, and condition *cond* is true, then the phi function effectively becomes "`v=`$\texttt{v}_0$" with an extra (explicit) control dependency on *cond*. Thus, the phi variable `v` depends on whatever loads $\texttt{v}_0$ and *cond* depend on. In our rules, the *assign_phi* function transform phi functions to assignments so that we can apply the *Assignment* rule for data dependency, and the *taint_phi* function and the *Phi.Taint* rule together taint the phi variables with the explicit control dependency on the condition. For example, the phi variable `r5` in line 9 in Figure 5 has an explicit control dependency on the `if` condition, which depends on the load "`r1 = load &x`". As a result, by the *Store* rule, the store "`store &z, r5`" in line 10 also depends the "`r1 = load &x`".

(5) If/else branches: An `if/else` branch can potentially introduce explicit and implicit control dependencies, as shown in Figure 6 (b) and (d). In addition, we must ensure that the phi variables associated with the branch also have dependencies on the appropriate right-hand side variable and a dependency on the `if` condition as discussed above. The *If.True* and *If.False* rules are applied when the `if` condition is true or false, respectively. The `hasReachableStore()` function returns true or false for whether or not a given block of statements has a syntactically reachable store (i.e., potential store). Note that the explicit control dependency set *PC* returns to its original state after we execute the `if/else` branch, and we track implicit control dependencies by applying the `hasReachableStore()` function on the untaken branch.

(6) While loops: In our dependency rules, a `while` loop can be unrolled indefinitely and viewed as if they were nested `if/else` branches. However, in terms of formalization, unlike normal `if/else` branches, we need to distinguish the first time we enter the `while` loop from later loop continuations for two reasons: (1) we need to assign the phi functions differently depending on whether we enter the loop for the first time or not; and (2) when we finish executing a loop, since we need to recover the explicit control dependency set *PC*, we need to record the old *PC* set based on the two different cases. Thus, we define the *While.Taken* rule for cases where we enter the loop for the first time and the loop condition is `true`, and the *While.Untaken* rule for those where we enter the loop for the first time and the loop condition is `false`. Note that once a `while` loop is taken for the first time, we change the loop keyword from `while` to `loop` as an indicator that the loop has been taken at least once. Also, when a `while` loop is taken for the first time, we record the old *PC* so that we can recover the *PC* status when we finish executing the loop. We then define the *Loop.Taken* rule for cases where a loop is taken after the first time, and the *Loop.Untaken* rule for cases where a loop is finished after being executed at least once. Note that applying any of these four rules has an effect on the explicit control dependency set *PC* and future store dependency *FD* similar to that of the conditional branch rules.

(7) Function calls: Since we only have function calls in which the functions are pre-defined and have a definite function name, a function call can be viewed as an inlined block of statements with extra data dependencies from the actual parameters to the formal parameters and from the return value to the actual function call result. Hence, to simplify the presentation and focus on the core problem, we omit the dependency rule involving function calls in the operational semantics shown above.
However, this is not sufficient for real-world programming languages, which can have function pointers (e.g., C/C++) or virtual dispatch mechanisms (e.g., object-oriented programming languages). The essential problem is that when the address of a function call depends on some load l, we should conservatively assume that the function call could potentially have stores that write to any possible memory location and thus must ensure that any future store from the point of the function call also depends on load l. We refer to this type of dependency as a *function dependency*. It is important to note that the implementation of our dependency-preserving compiler shown in Section 3 effectively enforces function dependencies.

2.3 A Load-Store-Order-Preserving Memory Model

An alternative approach to eliminating out-of-thin-air behaviors is to strengthen the existing C/C++ memory model by requiring *sequence-before* ∪ *reads-from* to be acyclic. This well-known C/C++ memory model variant has been proposed by researchers [Batty et al. 2013; Boehm and Demsky 2014; Vafeiadis and Narayan 2013] as one of the possible approaches to forbidding out-of-thin-air behaviors. Note that this is not the "perfect" fix to the problem since it forbids not just the problematic out-of-thin-air executions (e.g., Figure 2) but also some legitimate executions such as the load buffering example shown in Figure 1. This approach is less likely to be acceptable for Java-like memory models that must describe the behavior of all loads because it is likely to incur a much higher cost.

3 DEPENDENCY-PRESERVING COMPILER

This section describes the design and implementation of our approach to preserving our extended memory model in the LLVM compiler infrastructure [Lattner and Adve 2004]. We target the LLVM compiler in this paper for two reasons: (1) LLVM is widely supported and considered by many as the state-of-the-art compiler framework; and (2) LLVM is not just adopted as a C/C++ compiler but is also adopted in the context of a commercial JVM, e.g., Azul's Falcon compiler [Azul 2017].

3.1 Design

The LLVM compiler infrastructure is designed to compile source code and generate optimized library or executable files in a modular and reusable fashion. The standard LLVM compilation pipeline is shown in Figure 7, and we illustrate the workflow as follows:

(1) Given C/C++ source code files, the *Clang* front end translates them into a type of target-independent intermediate representation (IR), i.e., the LLVM Bitcode or the LLVM IR. The LLVM IR generated in this step has not been optimized yet, and hence it preserves all the trivial computations and control flows except very obvious constant folding, etc. For example, the statement "x = 2 * 2" in the source code will be translated into "store 4, x" in the unoptimized IR; however, the statement "x = r1 * 0" will be preserved. It is important to note that such trivial constant folding does not break dependencies.
(2) The unoptimized LLVM IR generated in step 1 will then go through the LLVM IR optimizer, which performs a list of target-independent LLVM IR transformation passes to generate optimized LLVM IR. In the LLVM tool chain, the LLVM optimizer — *opt* — can perform these

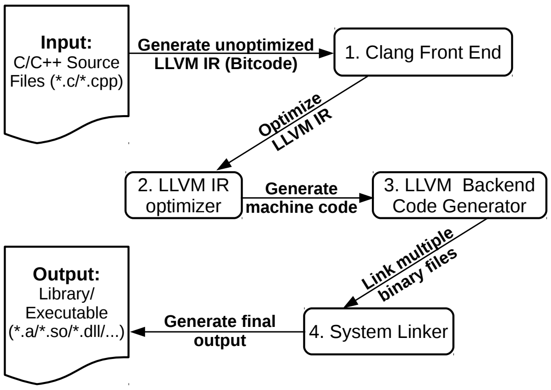
Input:
C/C++ Source Files (*.c/*.cpp)
Generate unoptimized LLVM IR (Bitcode)
1. Clang Front End
Optimize LLVM IR
2. LLVM IR optimizer
Generate machine code
3. LLVM Backend Code Generator
Link multiple binary files
4. System Linker
Generate final output
Output:
Library/ Executable (*.a/*.so/*.dll/...)

Fig. 7. The standard LLVM compilation workflow for C/C++

optimizations. Note that these transformation passes can potentially change the IR and break dependencies.

(3) The optimized LLVM IR generated in step 2 will then be passed to the LLVM backend code generator, which performs instruction selection, register allocation, peephole optimizations, etc., and finally generates optimized binary object files. In the LLVM tool chain, the LLVM static compiler — *llc* — implements this step. Note that the backend transformations are implemented as a list of code generation passes (LLVM machine passes performed in some machine code representations), and these passes can also potentially break dependencies. In general, the LLVM infrastructure supports multiple backend code generators for different architectures. In our work, we use the AArch64 (ARM's 64-bit architecture) backend as a case study and thus will focus on the code generation passes for AArch64 hardware. Our case study explore ARM 64 because it is the most relevant mainstream processor that implements a memory model that does not preserve load-store ordering. Intel processors implement the TSO memory model which by default preserves load-store ordering, and thus are expected to incur significantly less overhead to implement our memory model.

(4) Finally, the system linker will link the optimized object files and generate optimized library or executable files. Note that on a Linux machine, it is a common case to use the GNU linker as the default system linker.

As shown in the above pipeline, in order to ensure that dependencies are preserved down to the generated binary code, we must make sure the transformations in step 2 and 3 preserve the dependencies. Ideally, we should review each transformation pass. If it can potentially break dependencies, we should modify the optimization to retain most of the benefits of the optimization while preserving dependencies; otherwise, we should leave the pass unchanged. However, for our research project, this incurs too much manual effort. There are more than 50 LLVM transformation passes in step 2. Fortunately, a preliminary result shows that if we run only a select set of 35 IR-to-IR passes alone in step 2, the performance loss over running the standard set of IR-to-IR passes (as called in "opt -O3") is only 1.8%. As a result, we disable all other IR-to-IR passes and focus our efforts on the select set of passes. We show the set of IR passes that we enable in Appendix B.

3.2 Implementation

Several important compiler optimization passes are inherently dependency-preserving and require either no changes or only minor changes. We first discuss those that require no changes and then discuss how we modify/disable the remaining passes to preserve dependencies.

3.2.1 *Unmodified Passes.*

Function inlining: This optimization expands specific function call sites in the body of the caller functions, potentially reducing function call overhead and introducing more opportunities for later optimizations. Our preliminary result shows that turning off inlining incurs an overhead of 26% on AArch64 targets. Fortunately, no change is required for this pass as long as we conservatively preserve the syntactic dependencies in all functions. For example, Figure 8 shows a function `foo` that internally calls another function `bar`. The third column shows that as long as we preserve the dependency between the argument `arg` and the return statement "`return arg * 0`", the `foo` function after inlining still preserves the dependency between the load from `x` and the store to `y`.

Original `foo()`	Original `bar()`	Inlined `foo()`

`void foo() {`
`  r1 = x;`
`  y = bar(r1);`
`}`
	int bar(int arg) {
	`  return arg * 0;`
	`}`
		`void foo() {`
		`  r1 = x;`
		`  y = r1 * 0;`
		`}`

Fig. 8. Function inlining does not break the dependency between the load from `x` and the store to `y` as long as function `bar` preserves its internal dependencies, as shown in the third column.

Common subexpression elimination: Common subexpression elimination (CSE) replaces a redundant expression with the value of a pre-computed common expression. For example, it will transform the instructions "`z=x*y; t=x*y`" to "`r1=x*y; z=r1; t=r1`". We can see that the dependency from `x` and `y` to `t` is preserved because it is carried by the intermediate value `r1`. In LLVM, the global value numbering (*gvn*) pass can perform redundant load elimination that has similar effect to CSE.

Dead code elimination: Dead code elimination in general eliminates the instructions that are unreachable or have no visible effects to the program, and does not break dependencies. In LLVM, this corresponds to *adce* (aggressive dead code elimination) and *dce* (dead code elimination).

3.2.2 *Modified Passes.* To preserve a simple notion of syntactic dependency, we must consider both data dependencies and control dependencies. For example, as shown in Figure 9, the original LLVM optimizations (`-O3`) recognize that the expression "`r1*0`" will always generate the value `0` and will transform the store instruction to "`y=0`", which no longer depends on the load from `x`. In the other example shown in Figure 10, the original LLVM optimizations (`-O3`) determine that no matter which branch the program takes, it will execute the same store instruction, so it merges the two stores to `y` and later eliminates the empty control blocks. Hence, this breaks the dependency from "`y=1`" to the load from `x`.

Unoptimized code	Optimized code

`r1 = x;`
`y = r1 * 0;`
	`r1 = x;`
	`y = 0;`

Fig. 9. LLVM optimizations (`-O3`) can break data dependencies.

Unoptimized code	Optimized code

`if (x > 0) y = 1;`
`else y = 1;`
	`y = 1;`

Fig. 10. LLVM optimizations (`-O3`) can break control dependencies.

We next outline the important optimization passes that we have modified to preserve dependencies:

Combining redundant instructions (*instcombine*): This pass combines instructions to fewer and simpler ones and does not modify the control flow graph. For example, it performs simple constant folding, dead code elimination, algebraic simplification, and reordering of operands to expose more common subexpression elimination opportunities, etc. To preserve dependencies, we modify this pass to disable the transformations that can potentially break dependencies. Figure 11 shows examples of our modification to the instruction simplification optimization in order to preserve dependency. More specifically, Figure 11(a) shows an example in which we prevent it from simplifying the condition "`r2=(r1==r1)`" to "`r2=true`". At the same time, we still allow those dependency-preserving transformations and also perform a limited form of strength reduction on algebraic instructions when the original simplification would break dependencies. For example, Figure 11(b) shows that although we cannot completely simplify the three `AND` instructions to the value `0`, we can still perform partial simplification and eliminate two redundant `AND` instructions; Figure 11(c) shows that while we cannot transform "`r1*0`" to the value `0`, we can transform it to a potentially less expensive `AND` instruction.

	Unoptimized code		Dependency-preserving code
(a)	`r2 = (r1 == r1);`	⇒	`r2 = (r1 == r1);`
	`if (r2)...`		`if (r2)...`
(b)	`r2 = r1 & 0xffff;`	⇒	`r4 = r1 & 0;`
	`r3 = r1 & 0xffff0000;`		
	`r4 = r2 & r3;`		
(c)	`r2 = r1 * 0;`	⇒	`r2 = r1 & 0;`

Fig. 11. Examples of how the dependency-preserving *instcombine* pass transforms the code.

Simplify the CFG (*simplifycfg*): This pass simplifies control flows, which includes a form of dead code elimination with respect to control flows (e.g., removing unreachable basic blocks and basic blocks that contain only an unconditional branch), basic block merging, and hoisting common code outside of control flow blocks. Since we disable dependency-breaking algebraic simplifications in the *instcombine* pass, the *simplifycfg* pass cannot eliminate control flows by statically calculating the value of conditions. For example, in Figure 12 (a), since we disable simplifying the condition "`r1 == r1`" to the value `true`, the transformation shown in the middle column cannot happen, and thus the control flow dependency is preserved.

However, transformations that involve moving stores out of the control flow blocks are generally problematic and should be prohibited. Figure 10 shows such an example in which the pass first hoists the common stores "`y=1`" out of the `if/else` branch and then eliminates the `if/else` blocks, which breaks the dependency of "`y=1`" on the load from `x`. Figure 12 (b) shows an even more problematic example. Before the transformation, the store "`y=1`" depends on the load "`r1=x`", and the store "`*addr=2`" also depends on "`r1=x`" because of the conditional store "`y=1`" that is sequenced-before it (i.e., implicit control dependency). After the original transformation, although the new unconditional store "`y=r1?1:0`" still depends on "`r1=x`", the later store "`*addr=2`" no longer depends on "`r1=x`". Our dependency-preserving transformation preserves this missing dependency by adding redundant computations that require the value of the condition (i.e., "`(&y)|(r1&0)`") to compute the address of the new unconditional store (i.e., "`*r2=r1?1:0`") so that all later stores still depend on the old condition `r1`.

We also disable the elimination of control flow blocks in some cases even when there is no store within the control flow blocks. For example, in Figure 12 (c), we can see in the unoptimized code that the store to `z` syntactically depends on the load from `x` even though `r1` is a local variable. However, before LLVM runs the *simplifycfg* pass, it runs a pass that transforms the LLVM IR to static single assignment (SSA) form, which simplifies the instruction "`z=r1`" to "`z=1`" and makes the `if/else` blocks empty. This step alone preserves the control dependency because it does not modify the control flow. However, after that, the *simplifycfg* pass will eliminate the empty `if/else` blocks and transform it to the code shown in the middle, in which "z=1" does not depend on the load from x anymore. Our dependency-preserving transformation on the right keeps the empty conditional branch.

This shows that multiple dependency-preserving passes combined together can break dependencies. The fundamental problem that causes this is the poorly defined notion of dependency in the IR. In the example in Figure 12 (c), the later *simplifycfg* pass has no information about whether the empty conditional branch carries a dependency to later stores, thus eliminating it can potentially break a dependency, as shown in this case. Ideally, if the IR was augmented with extra dependency edges between statements, we could use that information to ensure that a specific transformation does not break existing dependency edges. In practice, to make our approach simpler to implement in the existing LLVM framework without requiring large changes to LLVM's IR, we adopt a coarse-grained extension to the IR such that basic blocks contain extra information indicating whether some other statements may or may not depend on them. In this case, when we construct the SSA form and encounter a PHI node that has the same value from multiple basic blocks, we conservatively mark the incoming blocks as unremovable to preserve such control flow blocks even if they are empty.

	Unoptimized	Original transformation	Dependency-preserving transformation

(a)	`r1 = x;`
	`if (r1 == r1) y = 1;`
	`else y = 2;`
		`r1 = x;`
		`y = 1;`
			`r1 = x;`
			`if (r1 == r1) y = 1;`
			`else y = 2;`

(b)	`r1 = x;`
	`y = 0;`
	`if (r1) y = 1;`
	`*addr = 2;`
		`r1 = x;`
		`y = r1 ? 1 : 0;`
		`*addr = 2;`
			`r1 = x;`
			`r2 = (&y) | (r1 & 0);`
			`*r2 = r1 ? 1 : 0;`
			`*addr= 2;`

(c)	`r1 = 0;`
	`if (x > 0) r1 = 1;`
	`else r1 = 1;`
	`z = r1;`
		`z = 1;`	`// Keep empty blocks`
			`if (x > 0) ;`
			`else ;`
			`z = 1;`

Fig. 12. Examples of how the *simplifycfg* pass can potentially break dependencies.

Passes that potentially reorder stores: According to our dependency notion, reordering an earlier store s_1 and a later store s_2 can potentially break address dependencies if the address of s_1 depends on some load that s_2 does not depend on. We list the passes that can reorder stores and our corresponding strategies as follows:

(1) Dead store elimination pass (*dse*): This pass looks for stores that have no visible side effects and eliminates them. Figure 13 (a) is an example in which the first store in the unoptimized code "arr[r1]=0" is a dead store since there are no loads after it until the last store "arr[r1]=1". In addition, store "y=1" depends on load "r1=x" because the address of "arr[r1]=0" depends on "r1=x" (i.e., address dependency). After the original transformation, this pass eliminates the store "arr[r1]=0", which breaks the dependency from store "y=1" to load "r1=x". Our solution is to add redundant computations involving the address arr[r1] to the store to y, which ensures that the dependency on "r1=x" is passed to the store "*r2=1".

(2) Loop invariant code motion (*licm*): This pass optimizes loops by moving loop invariant code outside of the loop. In general, it hoists load instructions out of the loop body and sinks store instructions to the end of the loop. Figure 13 (b) shows an example that illustrates why

this can be problematic. In the unoptimized code, all stores “`arr[i++]=1`” depend on the load from `x`, but the original transformation breaks this dependency by sinking the store “`addr[x]=0`”. We disable such transformations.
(3) SLP (superword-level parallelization) vectorization (*slp-vectorizer*): This pass can combine similar independent instructions into vector instructions. Figure 13 (c) shows how it can effectively reorder stores by combining adjacent stores. We modify this pass to prohibit the original transformation shown in Figure 13 (c).
(4) Memory copy optimization (*memcpyopt*): This pass performs optimization related to memset, memcpy, and memmove calls, and we disable this pass.

	Unoptimized	Original transformation	Dependency-preserving transformation
(a)	`r1 = x;`	`r1 = x;`	`r1 = x;`
	`arr[r1] = 0;`	`y = 1;`	`r2 =(&y)|((arr+r1)&0);`
	`y = 1;`	`arr[r1] = 1;`	`*r2 = 1; // y = 1`
	`arr[r1] = 1;`		`arr[r1] = 1;`
(b)	`do {`	`do {`	`do {`
	`  addr[x] = 0;`	`  arr[i++] = 1;`	`  addr[x] = 0;`
	`  arr[i++] = 1;`	`} while (i < 100);`	`  arr[i++] = 1;`
	`} while (i < 100);`	`addr[x] = 0;`	`} while (i < 100);`
(c)	`arr[x&0] = 0;`	`y = 1;`	`arr[x&0] = 0;`
	`y = 1;`	`arr[0..3] = {0..3};`	`y = 1;`
	`arr[1] = 1;`		`arr[1] = 1;`
	`arr[2] = 2;`		`arr[2] = 2;`
	`arr[3] = 3;`		`arr[3] = 3;`

Fig. 13. Examples of how reordering stores can potentially break dependencies.

Loop unrolling This pass performs loop unrolling, which expands the loop body across multiple iterations, reducing the overhead of checking the loop condition and updating the trip count, and expose further optimization opportunities (e.g., vectorization). Similar to `if/else` control dependencies, the loop body generally depends on the loop condition, and thus a full unrolling (i.e., expanding the loop body completely) can potentially break dependencies. Hence, we modify this pass such that it does not statically reason about the trip count of a loop and fully unroll the loop when its trip count is not an explicit constant, as shown in Figure 14 (a). However, if the trip count of a loop is specified as a constant, as shown in Figure 14 (b), we allow full unrolling because the loop condition does not depend on any loads.

	Before transformation		After transformation
(a)	`for (int i = 0; i < (x*0 + 2); i++)`	⇒	`arr[0] = 0;`
	`  arr[i] = i;`		`arr[1] = 1;`
(b)	`for (int i = 0; i < 2; i++)`	⇒	`arr[0] = 0;`
	`  arr[i] = i;`		`arr[1] = 1;`

Fig. 14. Examples of loop unrolling. (a) statically computing the trip count and unrolling the loop potentially breaks control dependencies; (b) unrolling loops with explicit constant trip count does not break dependencies.

Backend passes that can break dependencies: Given a dependency-preserving LLVM IR, the LLVM backend generates object code by passing the IR through a sequence of backend passes, which can also potentially break dependencies in the following ways:

(1) Data dependencies: The major backend pass that breaks data dependencies is the SelectionDAG-based instruction selection pass. To generate machine code, the LLVM backend first builds a per basic block structure called a selection DAG, which is a directed acyclic graph that represents the order of instruction within a basic block. It then goes through several rounds of node combining, which effectively performs a form of algebraic simplification, common subexpression elimination, constant folding, and strength reduction, etc. Similar to the modifications we made to the *instcombine* pass, we disable algebraic simplifications that can break dependencies.

(2) Control dependencies: In addition to the IR-level control flow simplification, the LLVM backend can further simplify control dependencies, e.g., merging branches and eliminating empty blocks, including those conditional branches on which we potentially rely to preserve dependencies. We modify the code generation preparation (*codegenprepare*) pass and completely disable the control flow optimizer (*branchfolding*) pass to preserve control dependencies.

4 LOAD-STORE-ORDER-PRESERVING COMPILER

This section describes the implementation of our approach to preserving atomic load-store ordering in the LLVM compiler for AArch64 targets. In LLVM IR, atomic load/store operations are special load/store operations with memory ordering parameters similar to their C/C++ counterparts (e.g., `memory_order_acquire`), and atomic read-modify-write operations (e.g., `compare_exchange_strong` and `fetch_add`) are represented as `atomicrmw` or `cmpxchg`. Similar to the dependency-preserving approach, we need to ensure that both IR-level optimizations and the backend generate code that preserves load-store ordering for atomic operations.

4.1 Target-Independent Optimizations

For IR optimization passes, we only need to focus on those passes that can potentially perform load-store reordering to atomic load/store operations, which fortunately is a small subset of the IR optimization passes. For example, these passes include the loop invariant code motion pass (*licm*), the memory copy optimization pass (*memcpyopt*), the dead store elimination pass (*dse*), the SLP vectorization pass, etc. We carefully reviewed these passes and found that they do not perform load-store reordering for atomic operations by design. The reasons include: (1) the semantics of atomic operations disallow the optimization (e.g., atomic operations cannot be optimized into a memcpy/memset operation since it can potentially reorder the atomic operations and change the visible side effect) or (2) it is tricky to reason about the correctness of optimizations of atomic operations, and the optimization is not especially important in most cases. For example, the *licm* pass will optimize normal loads/store out of a loop but is conservative with atomic operations. As a result, we can enable all the original IR-level passes.

4.2 Backend Optimizations for AArch64

4.2.1 LLVM AArch64 Backend for C/C++ Atomics. Figure 15 shows how the LLVM backend compiles C++ atomics to assembly for AArch64 targets. In example (a), an atomic load/store with `memory_order_relaxed` ordering parameter is compiled to a normal load/store instruction, while an atomic load with `memory_order_acquire`[5] or store with `memory_order_release` is compiled

[5]`memory_order_consume` is not broadly supported by compilers due to challenges associated with preserving data dependencies. LLVM effectively converts it to the stronger ordering parameter `memory_order_acquire`.

to a load-acquire (`ldar`) or store-release (`stlr`), respectively, which are load and store instructions in AArch64 with implicit one-way barrier semantics. For example, the normal load and store in line 2 and 3 can be reordered by the processor at runtime, while the `ldar` in line 1 guarantees that loads/stores after it cannot be reordered before the load. Similarly, all loads/stores before the `stlr` in line 5 cannot be reordered after it. In example (b), a `fetch_add` operation is compiled to a loop that continuously attempts to atomically fetch and add one to the memory location. It takes advantage of the exclusive load/store (`ldxr`/`stxr`) pair, which has exclusive locking semantics on the load/store address to guarantee the read-modify-write is atomic. It is important to note that in the ARMv8 architecture, the success bit (`w10` in this example) of a successful store-exclusive is not supposed to introduce any dependency from the load-exclusive it is paired with [Pulte et al. 2018]. As a result, the store in line 6 does not have a dependency on the load in line 2 and thus can be reordered before it. In example (c), a compare-and-swap operation is compiled to a conditional branch that compares the load value with the expected value and then decides whether or not it should store the new value to the memory. Since there is a control dependency (line 3) from the load part (line 2) to any stores after the compare-and-swap operation, those subsequent stores cannot be reordered before the load part. The conclusions from these three examples are (1) that atomic loads that have a stronger ordering parameter than `memory_order_relaxed` and atomic compare-and-swap operations already have an ordering constraint relative to subsequent stores, and (2) that we only need to preserve the ordering from the relaxed load and those `fetch_add`-like read-modify-write operations (with ordering parameters that do not have acquire semantics) to subsequent stores.

	C++ code		AArch64 assembly
(a)	`r1 = arr[0].load(acquire);`	⇒	`1: ldar w1, [x8]`
	`r2 = arr[1].load(relaxed);`		`2: ldr w2, [x8, #4]`
	`arr[2].store(0, relaxed);`		`3: str wzr, [x8, #8]`
	`arr[3].store(0, release);`		`4: add x8, x8, #12`
			`5: stlr wzr, [x8]`
(b)	`r1 = arr[0].fetch_add(1, relaxed);`	⇒	`1:.BB_1:`
	`arr[1].store(0, relaxed);`		`2: ldxr w9, [x8]`
			`3: add w9, w9, #1`
			`4: stxr w10, w9, [x8]`
			`5: cbnz w10, .BB_1`
			`6: str wzr, [x8, #4]`
(c)	`int expected = 0;`	⇒	`1:.BB_1:`
	`r1 = arr[0].compare_exchange_weak(`		`2: ldxr w9, [x8]`
	` expected, 1, relaxed, relaxed);`		`3: cbz w9, .BB_2`
	`return;`		`4: clrex`
			`5: ret`
			`6:.BB_2:`
			`7: orr w9, wzr, #0x1`
			`8: stxr w10, w9, [x8]`
			`9: ret`

Fig. 15. Examples of how LLVM backend compiles C++ atomic operations to assembly code for AArch64 targets. In each example, variable `arr` is an array of `atomic_int`, and register `x8` contains the base address of array `arr`.

4.2.2 Forbidding Reordering of Loads and Stores in AArch64. To forbid a normal load from being reordered with subsequent stores in AArch64 targets, Boehm and Demsky [2014] propose that one could add either a fence or a bogus conditional branch after the load (i.e., adding control dependency from the load to subsequent stores). For cases in which a load is followed by a store, one could alternatively add a bogus address dependency [Maranget et al. 2012] from the load to the store to guarantee the ordering. Also, a similar strategy with respect to adding address dependency to insert (between the target relaxed load and subsequent stores) a bogus load whose address depends on the target relaxed load. To better understand the performance characteristics of these options, we use micro-benchmarks written in assembly to benchmark the performance overhead of these options on an ARM Cortex-A72 core.

The first option is to simply replace the normal load with a `ldar` load, which has implicit acquire semantics. The second option is to replace the normal store with a `stlr` store, which has implicit release semantics. The third option is to insert a "`dmb ld`" fence before a relaxed store so that it waits for previous loads to finish. The fourth option is to add a bogus conditional branch after the normal load such that the branch condition uses the result of the load[6]. The fifth is to add a bogus load whose address depends on the target relaxed load. The sixth option is to add an extra address/control dependency from the normal load to an existing subsequent store or conditional branch instruction. Figure 16 shows the performance overhead of these strategies, which is normalized to the performance of the micro-benchmarks without any load-store ordering constraints. The "Store" column represents the scenario in which a load is followed by a store, and the "Conditional Branch" column represents the scenario in which a load is followed by a conditional branch. This result shows that using release stores is the most expensive option and adding bogus conditional branches after relaxed loads is the least expensive option in either scenario for the processor we used, and that adding fences (i.e., the first three options) is more expensive than the other three alternatives. Given this preliminary result, we adopt the strategy of adding bogus conditional branch in the implementation of our load-store-order-preserving compiler. It is important to note that compared to adding bogus conditional branches, the strategies of adding address dependencies to existing stores/branches or inserting bogus dependent loads can be potential solutions for those processors that incur higher overheads from fake conditional branches.

Strategy/Subsequent Instruction	Store	Conditional Branch
Acquire Load	500.1%	267.7%
Release Store	1095.1%	382.0%
DMB LD Fence	457.1%	238.3%
Bogus Conditional Branch	28.6%	26.2%
Bogus Load	50.0%	26.4%
Extra Dependencies to Existing Store/Branch	50.0%	28.8%

Fig. 16. Performance overhead incurred by different strategies of forbidding load-store reordering for micro-benchmarks.

Figure 17 (a) and (b) show examples in which a relaxed load is followed by an existing subsequent store or conditional branch in the same basic block, respectively. In both examples, we intentionally add a bogus conditional branch that uses the result of the load, i.e., lines 2 to 4 in Figure 17 (a) and lines 2 to 4 in Figure 17 (b). This intentional control dependency forces stores after the load to be visible after the load. Note that we add an `AND` instruction (specifically `AND` with `zero`) in

[6]At the time of writing this paper, there is still some uncertainty about what the branch target should look like. In this paper, we evaluate this strategy based on Pulte et al. [2018]'s model, in which any instruction succeeding a conditional branch in program order has control dependency on the loads that the branch has data dependency on.

	C++ code		AArch64 assembly
(a)	`r1 = arr[1].load(relaxed);`	⇒	`1: ldr w1, [x8, #4]`
	`arr[0].store(0, relaxed);`		`2: and w2, w1, wzr`
			`3: cbnz w2, .BB_1`
			`4:.BB_1:`
			`5: str wzr, [x8]`
(b)	`r1 = arr[1].load(relaxed);`	⇒	`1: ldr w1, [x8, #4]`
	`if (r2)`		`2: and w9, w1, wzr`
	` arr[0].store(0, relaxed);`		`3: cbnz w9, .BB_1`
			`4:.BB_1:`
			`5: cbz w2, .BB_2`
			`6: str wzr, [x8]`
			`7:.BB_2:`
(c)	`r1 = arr[1].load(relaxed);`	⇒	`1: ldr w1, [x8, #4]`
	`if (r1)`		`2: cbz w1, .BB_1`
	` arr[0].store(0, relaxed);`		`3: str wzr, [x8]`
			`4:.BB_1:`
(d)	`r1 = arr[1].load(relaxed);`	⇒	`1: ldr w1, [x8, #4]`
	`arr[r1].store(0, relaxed);`		`2: str wzr, [x8, w1, sxtw #2]`

Fig. 17. Our approach to imposing the ordering between relaxed loads and subsequent stores. Register `x8` contains the base address of array `arr`. Bogus conditional branches are added intentionally to impose the load-store ordering in example (a) and (b), and example (c) and (d) do not require such extra ordering constraints because the ordering constraints exist in the source code inherently.

lines 2 of both examples (a) and (b) to ensure that the conditional branch consistently takes the same direction to avoid too many branch mispredictions.

Another important observation is that for some relaxed loads, there already exist reordering constraints from the load to subsequent stores. For example, in the source code in Figure 17 (c), the conditional branch after the relaxed load already depends on the result of the load, so any stores after the load in the assembly naturally have a control dependency on the load and must be visible after it without adding any redundant instructions. Similarly, Figure 17 (d) shows an example in which a subsequent store naturally has an address dependency on the load and thus we do not need to add extra reordering constraints. In order to optimize for these cases to avoid unnecessary overheads, we implement a local analysis that conservatively checks whether the address of a subsequent store or the condition of a subsequent branch depends on specific loads and use the analysis result to decide whether we need to add a bogus conditional branch after the loads.

To implement our solution, we made two modifications to the LLVM AArch64 backend:

(1) Add extra ordering constraints for relaxed loads: We modify the code generation preparation pass such that before LLVM lowers optimized IR to machine code, it collects the set of relaxed loads that need extra ordering constraints (e.g., examples shown in Figure 17 (a) and (b)). We then intentionally add bogus conditional branches after the collected relaxed loads. Note that when there are multiple relaxed loads in a sequence, we only insert one bogus conditional branch whose condition uses the result of all those loads.
(2) Preserving redundant data/control dependencies: After the above modification to the code generation preparation pass, we still need to ensure that later backend passes (e.g., the SelectionDAG-based instruction selection and control flow optimization pass) do not optimize these instructions away, e.g., eliminating “`and w2, w1, wzr`”.

5 EVALUATION

In this section, we evaluate the cost of the two approaches to avoiding out-of-thin-air behaviors in C/C++ for AArch64 targets. In our evaluation, we report execution times on a Firefly-RK3399 board, which has a six-core 64-bit CPU (two ARM Cortex-A72 cores and four ARM Cortex-A53 cores), 4 GB memory, and runs Ubuntu 16.04.2. We have made both our compiler implementations and benchmarks publicly available at http://plrg.eecs.uci.edu/oota-html. As Sullivan [2017] shows, the performance results can vary depending on the processor in question. More specifically, their results seem to suggest that dependencies and fences may exhibit less performance penalty in ARMv8 than in ARMv7 and Power architectures. Ideally, the evaluation would have been more complete if we also considered the ARMv7 and Power architectures; however, in LLVM, they use different backends (which does require review and modifications) to generate architecture-specific code, so this is a non-trivial effort. As a first-step, we believe that evaluating the approaches on a relatively new version of 64-bit ARM processor could still be a useful indicator for future processors, and decided to leave the evaluation on the ARMv7 and Power architectures as future work.

5.1 Cost of Preserving Dependencies

Although avoiding out-of-thin-air behaviors applies only to multi-threaded code, our dependency-preserving approach incurs overhead for both single-threaded and multi-threaded programs. Single-threaded code represents a worse case scenario—the memory system bandwidth is not utilized by other cores and thus the extra instructions we add have a relatively higher cost. Thus, we measure the overheads of our dependency-preserving optimizations on single-threaded code.

5.1.1 Single-Threaded Programs. We ran each C/C++ benchmark in SPEC CPU2006 [Henning 2006] under four compiler configurations. The configuration "Full Optimizations" is the stock LLVM compiler with all optimizations enabled (`-O3`); the configuration "No Optimization" is the stock LLVM compiler with all optimizations disabled (`-O0`); the configuration "Dependency-preserving" is our dependency-preserving compiler. Due to the amount of engineering work needed to review/modify each optimization pass, we only select a core set of IR-level optimization passes (35 out of 46) to carefully review and modify when necessary to implement the dependency-preserving compiler. The configuration "Partial Optimization" is the stock LLVM compiler whose IR-level optimizations only include the same core set of passes that are enabled in our dependency-preserving compiler.

Note that the "No Optimization" configuration (`-O0`) naturally preserves dependencies; however, the benchmarks under this configuration execute with an average (geometric mean) slowdown of 155.9% and a maximum slowdown of 580.9%. Figure 18 shows more detailed performance overhead of each benchmark under configurations "Partial Optimization" and "Dependency Preserving", with each normalized to the performance under "Full Optimization" (`-O3`) configuration. Under the "Partial Optimization" configuration, the benchmarks incur an average of 1.8% slowdown and a maximum of 11.6% slowdown. Our dependency-preserving compiler has an average 3.1% slowdown and a maximum of 17.6% slowdown. Given the fact that we completely turn off 11 IR-level passes in our dependency-preserving compiler, which roughly accounts for the 1.8% overhead as shown under the "Partial Optimization" configuration, it is likely that one could further reduce the overhead of preserving dependencies by analyzing those optimization passes. There also remain opportunities for further optimization of the passes that we modified for the dependency-preserving memory model.

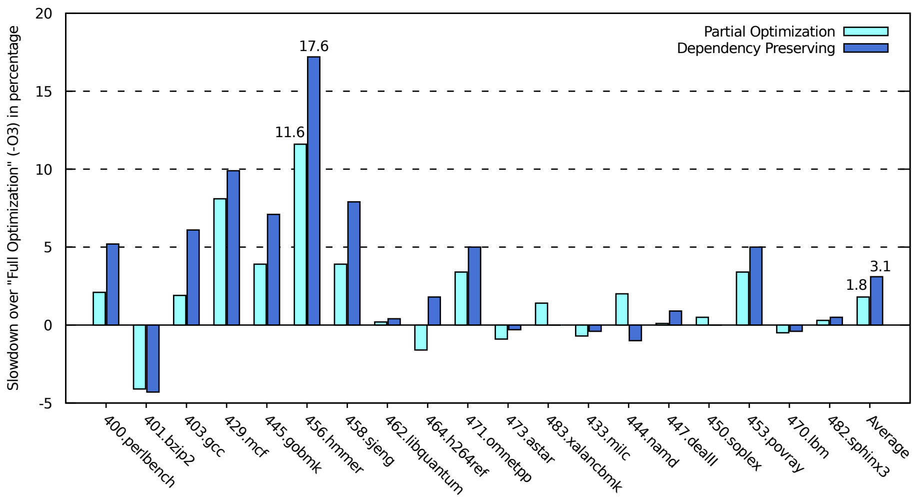

Fig. 18. Performance overhead (in percentage) introduced by different compiler configurations compared to the full optimization configuration (-O3) for C/C++ benchmarks in SPEC CPU2006.

Speedup in Single-Threaded Runs. As shown in Figure 18, we observe speedup in the single-threaded runs for some benchmarks under our dependency-preserving compiler. A possible explanation is the non-linear interaction between some optimization passes on some benchmarks. In fact, researchers have shown that different compiler options or transformation combination could have a significant impact on performance factors (e.g., cache accesses) [Cavazos et al. 2007; Pan and Eigenmann 2006]. One supportive observation in our case is that some benchmarks such as "401.bzip2" under the "Partial Optimization" configuration also have a speedup over the baseline "Full Optimization" configuration. Moreover, our dependency-preserving compiler does require modifying some backend passes such that they do not eliminate intentionally added AND instructions or conditional branches; and we have observed that simply disabling the backend control flow optimizer pass (i.e., *BranchFolding*) in the stock LLVM ("-O3") yields speedups for some single-threaded benchmarks. To give a more detailed comparison, for those benchmarks with a speedup under the "Dependency Preserving" configuration over the baseline in the single-threaded runs, we also list their overhead over the "Partial Optimization" configuration in Figure 19.

Benchmark	Overhead over "Partial Optimization" (%)
401.bzip2	-0.2
473.astar	0.6
433.milc	0.3
444.namd	-2.9
470.lbm	0.1

Fig. 19. Performance overhead of the "Dependency Preserving" configuration compared to the "Partial Optimization" configuration in the single-threaded runs for the benchmarks that have a speedup under "Dependency Preserving" configuration over baseline.

5.1.2 Multiple Copies of Single-Threaded Programs. To evaluate the performance overhead in a multi-core environment, we ran two copies of each C/C++ benchmark in SPEC CPU2006 at the same time, with each copy running on a Cortex-A72 core. We report the performance overhead of our dependency-preserving compiler compared to the stock LLVM with all optimizations enabled (-O3) in Figure 20. In this scenario, our dependency-preserving approach incurred an average slowdown of 2.6% and a maximum slowdown of 13.7%, which is smaller than that of running in a single-copy scenario in Section 5.1.1. A likely explanation is that our approach to preserving dependencies increases the number of instructions that are executed but does not significantly increase the number of memory accesses to data. As a result, when we run multiple copies of the same program simultaneously, the memory bandwidth that is accessible to each copy is reduced, and hence the cost of running the extra CPU instructions becomes relatively smaller (especially for memory-bounded programs). This experimental result indicates that in multi-core environments in which more than one core is used, the performance overhead incurred by our dependency-preserving compiler is likely to be smaller than that incurred in the single-core scenario.

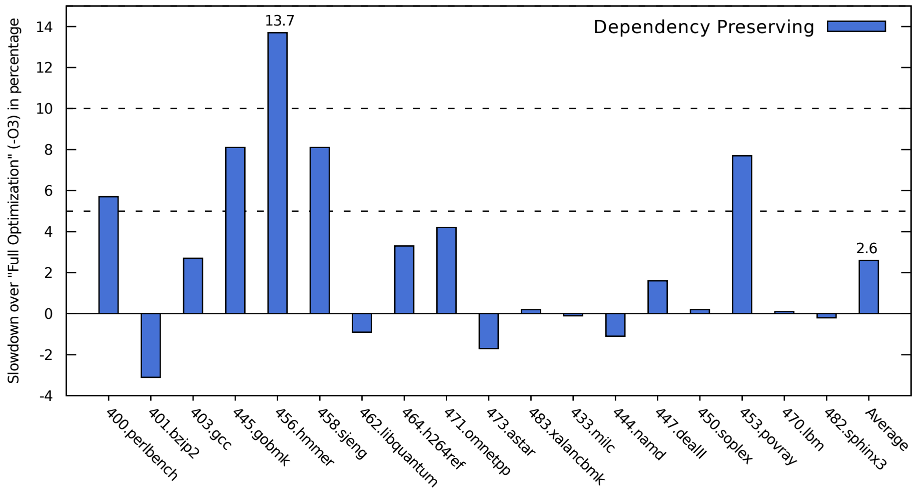

Fig. 20. Performance overhead (in percentage) introduced by our dependency-preserving compiler compared to the full optimization configuration (-O3) for C/C++ benchmarks in SPEC CPU2006 with two copies of each benchmark running at two cores simultaneously. We omit the "429.mcf" benchmark here because running two copies at the same time requires more than 4 GB memory and thus causes out-of-memory error.

5.2 Cost of Forbidding Load-Store Reordering

Unlike the dependency-preserving approach, forbidding load-store reordering to avoid out-of-thin-air behaviors only affects relaxed atomics in C/C++11. Hence, for example, the load-store-order-preserving approach should impose no overhead on the SPEC CPU2006 benchmarks because they do not use any C/C++ relaxed atomics. We believe that relaxed atomics will primarily appear in concurrent data structure code, while most other program code would not be affected since they would likely use other primitives that provide stronger semantics, e.g., locks and atomics with `memory_order_seq_cst`. Hence, we focus on evaluating the performance overhead incurred by forbidding load-store reordering for real-world concurrent data structures. The results can be roughly viewed as the upper bound of the performance overhead of this approach. The performance impact on full applications would depend on how much time those applications spend in concurrent data structure code. Ideally, we would also like to benchmark full applications; however, many existing multi-threaded applications that we have access to are not kept up-to-date with the C/C++ memory model, and porting them to use C/C++ atomics is a non-trivial effort. Hence we leave it as future work.

5.2.1 Concurrent Data Structures with Multiple Threads. In this evaluation, we gather a total set of 43 real-world concurrent data structures from several different sources, which range from basic synchronization primitive implementations, concurrent queues/stacks/deques to concurrent maps. Most of these data structures are lock-free, and all of them intensively utilize C/C++11 atomics. In more details, among these concurrent data structures, we collect 18 of them from the CDS C++ library [Khiszinsky 2017], 13 of them from the Folly library [Facebook 2018], 4 different implementations of concurrent maps from the Junction library [Preshing 2018], 2 queue implementations by Rigtorp [2017a,b], and 6 benchmarks used in CDSSpec [Ou and Demsky 2017].

We ran each benchmark using 7 compiler configurations: (1) the stock LLVM compiler with all optimizations enabled (`-O3`), i.e., *Full Optimization*; (2) our load-store-order-preserving compiler which adds bogus conditional branches after relaxed loads (*Bogus Conditional Branch*); (3) a variant of configuration 2 which adds address dependencies to existing stores rather than bogus conditional branches if there is a subsequent store after a relaxed load (*Address Dependency to Store*); (4) a modified compiler which adds address dependencies from relaxed loads to a subsequent load, which can be an existing load if any or an intentionally inserted bogus load otherwise (*Bogus Load*). Note that for a target relaxed load, we insert a bogus load whose address is the same as the relaxed load to avoid cache misses; (5) a modified compiler which treats relaxed loads as acquire loads (*Acquire Load*); (6) a modified compiler which treats relaxed stores as release stores (*Release Store*); and (7) a modified compiler which inserts “`dmb ld`” fences before relaxed stores (*DMB Fence*).

Since the Firefly-RK3399 board has two faster ARM Cortex-A72 cores and four slower ARM Cortex-A53 cores, it can potentially increase the noise in our performance evaluation if we run the benchmarks with multiple threads across the two different types of cores. Hence, we ran each of our benchmark with two threads, and each thread exclusively runs on a Cortex-A72 core. We ran each benchmark test case for 5 times and use the average (arithmetic mean) of those 5 runs as the execution time for each benchmark test case. For a benchmark in which there exist multiple variants, we use the geometric mean of the execution time of all the variants as the execution time of that benchmark. Although we ran the benchmarks with only two threads in this experiment, it is important to note that the extra overhead (i.e., extra dependencies or fences) that we introduce in this approach is local to each core and thus should not result in extra communication between cores, and hence one would not expect scaling issues; moreover, if the processor has limited memory bandwidth, as the number of cores utilized increases, the relative overhead of these extra dependencies or fences should become smaller.

	Multiple Threads		Single Thread	
Configurations/Overheads	Average	Maximum	Average	Maximum
Bogus Conditional Branch	-0.3%	6.3%	-0.0%	5.2%
Address Dependencies to Store	1.3%	23.2%	0.5%	8.7%
Bogus Load	2.6%	42.9%	2.8%	14.7%
Acquire Load	0.4%	27.5%	2.1%	42.7%
Release Store	3.6%	82.6%	6.8%	38.9%
DMB Fence	-0.1%	32.0%	3.2%	25.9%

Fig. 21. Performance overheads (over full optimizations) incurred by different strategies of forbidding load-store reordering for concurrent data structure benchmarks. The “Multiple Threads” columns show results for benchmarks running with two threads, and the “Single Thread” columns show results for benchmarks running with a single thread.

The performance overheads of different strategies to preserve load-store ordering (running with two threads) over the performance under full optimizations are shown in the “Multiple Threads” columns in Figure 21. The “Average” column shows the geometric mean of the execution time of our benchmarks, and the “Maximum” column shows the maximum overhead incurred by the corresponding strategy. We can see that the “*Bogus Conditional Branch*”, “*Acquire Load*” and “*DMB*

Fence" strategies incur an average overhead of less than 0.5% across the 43 benchmarks on average. Notably, the "*Bogus Conditional Branch*" strategy does not incur an overhead on average and only incurs a maximum of 6.3% overhead. All other strategies have higher maximum overhead than the "*Bogus Conditional Branch*" strategy, indicating that they are less desirable approaches to preserving load-store ordering in our experimental setting. We also show the performance results of each benchmark for the *Bogus Conditional Branch* strategy in Appendix E and the optimizations for the load-store-order-preserving approach in Appendix D.

Contention. Under this experimental setting, we found that some of our benchmarks have a faster execution time under our load-store-order-preserving compilers over full optimizations (-O3), such as the Folly UnorderedAtomicInsertMap implementation [Facebook 2018]. A possible explanation is that there exists contention in the data structure, and adding extra instructions to implement ordering constraints alleviates this contention. To better compare the performance of our approach, we also run our benchmark in a single-threaded (contention-free) setting in Section 5.2.2.

5.2.2 Concurrent Data Structures with a Single Thread. We run our benchmarks in a single thread on an ARM Cortex-A72 core in order to study the performance overhead of our approach without the contention issue under the 7 compiler configurations described in Section 5.2.1. For example, for a concurrent queue, we ran the queue with a single thread, which executes a certain number of enqueue method calls and then a certain number of dequeue method calls. The results are shown in the "Single Thread" columns in Figure 21. We can see that without contention, the "*Bogus Conditional Branch*" strategy does not incur an overhead over full optimizations on average and only incurs a maximum of 5.2% overhead. It also shows that the "*Bogus Load*", "*Acquire Load*", "*Release Store*" and "*DMB Fence*" strategies are more expensive on average and in worst case, which agrees with our micro-benchmarking results shown in Figure 16. Notably, even though the "*Bogus Load*" strategy only adds address dependencies to existing or bogus loads (which may seem inexpensive), it is still not desirable relative to the "*Bogus Conditional Branch*" strategy. A possible explanation is that the added address dependencies can halt the execution of all future memory operations.

Thus, among the six strategies we implemented, when we consider both the multi-threaded and single-threaded experiment results, the "*Bogus Conditional Branch*" strategy is the most desirable under the processor we use because it has the lowest average overhead and worst-case overhead in both the multi-threaded and single-threaded runs compared to all other strategies. Also, if we consider the single-threaded runs, the "*Address Dependencies to Store*" strategy is only slightly less desirable than the "*Bogus Conditional Branch*" strategy, and it may serve as a potential approach for processors that incur higher overheads from fake conditional branches.

6 RELATED WORK

In spite of much research on high-performance concurrent programming languages, we still do not have a definitive solution to the out-of-thin-air problem.

The C/C++11 memory model [Batty et al. 2011; Becker 2011; Boehm and Adve 2008; JTC 2011] does not forbid out-of-thin-air executions; and the C++14 memory model [?] does not clearly define out-of-thin-air behaviors and only vaguely states that implementations should ensure that out-of-thin-air values that circularly depend on their computations should be disallowed. In our dependency-preserving approach, we formally define a notion of dependency and evaluate a prototype implementation on widely deployed commercial hardware.

Boehm and Demsky [2014] propose the approach of ensuring that $sb \cup rf$ is acyclic for relaxed atomics, but the actual overhead was unclear. Our work complements this by providing an initial evaluation on the overhead of the approach for execution of real-world concurrent data structure code on a mainstream processor.

The Java memory model [Manson et al. 2005; Shipilëv 2016a,b] disallows the canonical out-of-thin-air example shown in Figure 2 by establishing a notion of causality and showing that there cannot exist a justifying execution for `r1=r2=42`. However, the Java memory model was later found unsound with respect to some common compiler optimizations [Cenciarelli et al. 2007; Ševčík and Aspinall 2008] such as redundant read elimination. The fact that our dependency-preserving approach supports normal memory accesses rather than just C/C++ atomics indicates that it is a promising direction to explore for the Java memory model.

Dolan et al. [2018] have recently proposed a memory model that provides a property called *local data race freedom*, which guarantees that all data-race-free portions of a program still have sequential consistency semantics. They show that to implement their memory model, one would need to preserve the ordering between loads and stores. They implement it on OCaml with similar strategies that we use in our load-store-order-preserving compiler and show that the average overhead over stock OCaml compiler for ARMv8 architecture with sequential programs is ~0.6%. Although both their results and ours suggest that the overhead of preserving load-store ordering is relatively low on ARMv8 architecture, it is important to note the differences: (1) the primary goal of their model is to provide the local DRF property but not to prohibit OOTA behavior, although their model effectively disallows OOTA behavior; and (2) their results are based on OCaml, which has a Java-like memory model, and thus they need to preserve load-store ordering for all normal (non-atomic) accesses; while our approach targets the C/C++ memory model, which only affects the C/C++ atomics.

Sullivan [2017] has proposed an approach called *Relaxed Memory Calculus* (RMC) that is fundamentally different from the C/C++ and Java memory models. In the RMC approach, programmers essentially reason about the relative ordering of memory accesses in concurrent programs (in a fashion close to hardware memory models) and explicitly specify the constraints on the execution order and visibility of writes. Unfortunately, the RMC approach also suffers from OOTA behavior and needs further fixes. It is important to note that Sullivan [2017] demonstrates that ARMv8 seems to have a smaller overhead on dependencies/fences than ARMv7 and Power; more notably, SC atomics perform nearly as well as C11 atomics on ARMv8. Hence, this encourages future work to extend our evaluation to ARMv7 and Power architecture.

Researchers have also proposed memory models whose goal is to disallow out-of-thin-air behaviors while embracing compiler optimizations. Jeffrey and Riely [2016]; Pichon-Pharabod and Sewell [2016] have proposed weak memory models based on event structures, which are sets of memory access events with some causal order and conflict relationship. Instead of focusing on a candidate execution, these approaches capture how different candidate executions relate to each other and diverge. Kang et al. [2017] propose a memory model that forbids out-of-thin-air behaviors based on operational semantics with timestamps and promises. Their approach introduces special reduction steps to allow a thread to perform a write with a promise that can be locally certified. Jagadeesan et al. [2010] propose an operational memory model for Java based on a notion of speculation to forbid OOTA executions. Our approach takes a different direction to provide an initial study of the runtime overhead of restricting compiler optimizations to eliminate out-of-thin-air behaviors.

Researchers [Marino et al. 2011; Singh et al. 2012] have suggested stronger memory models, e.g., sequential consistency, in which out-of-thin-air behaviors are prohibited. They show that the cost is low when they implement such memory models on specialized hardware. Our approaches also explores stronger memory models than the existing C/C++ memory model; however, the constraints we impose in general are much weaker than the sequential consistency memory model. In addition, both of our approaches directly target existing widely-deployed commercial processor designs that implemented a relaxed memory model. Liu et al. [2017] have proposed a stronger

Java memory model, which by default has sequential consistency semantics. They show that the overhead is arguably acceptable for server-side applications running on Intel x86 architectures.

Zhang and Feng [2016] propose an operational memory model that is based on a replay mechanism to simulate speculation. Their model forbids some but not all OOTA behaviors. Demange et al. [2013]; Ševčík et al. [2013] propose TSO for C and Java, which is strictly stronger than our approach of preserving load-store ordering.

There is also work that benchmarks the performance of weak memory models. Ritson and Owens [2016] focus on investigating the cost of prohibiting out-of-thin-air behaviors on the Linux kernel. They inject identifiable assembly sequences into the compiler output and use binary rewriting techniques to test different instruction sequences that may prevent out-of-thin-air behaviors. Our work focuses on a more general-purpose approach which involves modifying existing compiler code generation process and comparing the result with the original compiler.

Hardware memory models generally do not allow out-of-thin-air behaviors since they respect a syntactic data and control dependencies, while traditional compiler optimizations could potentially introduce such behaviors [Boehm and Demsky 2014]. Our dependency-preserving approach defines a dependency notion that is close to that of the hardware and enforces the compiler to generate code that respects such dependencies.

7 CONCLUSION

Restricting compiler optimizations is a promising solution to eliminate out-of-thin-air behaviors. Our results show that on an ARMv8 processor the dependency-preserving approach has an average overhead of 3.1% and a maximum overhead of 17.6% on the SPEC CPU2006 C/C++ benchmarks, and that the load-store-order-preserving approach has no overhead on average and a maximum overhead of 6.3% on 43 concurrent data structures, which indicates that the approach deserves further consideration. There remain opportunities to further reduce overheads by implementing more sophisticated optimizations and by carefully auditing the compiler optimization passes we omitted.

ACKNOWLEDGMENTS

We thank Doug Lea, Viktor Vafeiadis, Paul McKenney, Olivier Giroux, Alan Stern, JF Bastien, Will Deacon, Stephen Dolan, and Hans Boehm for their helpful comments or discussions. We would also like to thank the anonymous reviewers for their helpful comments. This project was partly supported by a Google Research Award and the National Science Foundation under grants CCF-1319786, OAC-1740210 and CNS-1703598.

REFERENCES

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.*

Jade Alglave, Luc Maranget, Paul E McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems.*

Azul. 2017. https://www.azul.com/press_release/falcon-jit-compiler/. (May 2017).

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages.*

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015a. The Problem of Programming Language Concurrency Semantics. In *Proceedings of the 2015 European Symposium on Programming.*

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015b. The Problem of Programming Language Concurrency Semantics. In *Proceedings of the 24th European Symposium on Programming.*

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages.*

Pete Becker. 2011. ISO/IEC 14882:2011, Information Technology – Programming Languages – C++. (2011).

Hans Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness.*

Hans J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Matko Botinčan, Paola Glavan, and Davor Runje. 2010. Verification of Causality Requirements in Java Memory Model is Undecidable. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics.*

John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the 5th Annual IEEE/ACM International Symposium on Code Generation and Optimization.*

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Proceedings of the 2007 European Symposium on Programming.*

Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. 2013. Plan B: A Buffered Memory Model for Java. In *Proceedings of the Symposium on Principles of Programming Languages.*

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 242–255.

Facebook. 2018. https://github.com/facebook/folly. (Mar 2018).

John L Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

Radha Jagadeesan, Corin Pitcher, and James Riely. 2010. Generative Operational Semantics for Relaxed Memory Models. In *Proceedings of the 2010 European Symposium on Programming.*

Alan Jeffrey and James Riely. 2016. On Thin Air Reads towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science.*

ISO JTC. 2011. ISO/IEC 9899:2011, Information Technology – Programming Languages – C. (2011).

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages.*

Max Khiszinsky. 2017. https://github.com/khizmax/libcds. (Dec 2017).

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages.*

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++ 11. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 618–632.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization.*

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A Volatile-by-Default JVM for Server Applications. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 49.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the Symposium on Principles of Programming Languages.*

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to The ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf. (2012).

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

Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. Out-of-Thin-Air Execution is Vacuous. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html. (Jul 2016).

Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. Pattern-based Synthesis of Synchronization for the C++ Memory Model. In *Formal Methods in Computer-Aided Design.*

Brian Norris and Brian Demsky. 2013. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

Peizhao Ou and Brian Demsky. 2015. AutoMO: Automatic Inference of Memory Order Parameters for C/C++11. In *Proceeding of the 30th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

Peizhao Ou and Brian Demsky. 2017. Checking Concurrent Data Structures Under the C/C++11 Memory Model. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the 4th Annual IEEE/ACM International Symposium on Code Generation and Optimization.*

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the Symposium on Principles of Programming Languages*.

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *Proceedings of the 31st European Conference on Object-Oriented Programming*.

Jeff Preshing. 2018. https://github.com/preshing/junction. (Feb 2018).

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the Symposium on Principles of Programming Languages*.

David P Reed and Rajendra K Kanodia. 1979. Synchronization with eventcounts and sequencers. *Commun. ACM* 22, 2 (1979), 115–123.

Erik Rigtorp. 2017a. https://github.com/rigtorp/SPSCQueue. (Feb 2017).

Erik Rigtorp. 2017b. https://github.com/rigtorp/MPMCQueue. (Aug 2017).

Carl G Ritson and Scott Owens. 2016. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the Symposium on Principles of Programming Languages*.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-memory Concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 22.

Aleksey Shipilëv. 2016a. Java Memory Model Pragmatics. https://shipilev.net/blog/2014/jmm-pragmatics/. (Sep 2016).

Aleksey Shipilëv. 2016b. Java Memory Model Pragmatics. https://shipilev.net/. (Oct 2016).

Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-end Sequential Consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*.

Michael J Sullivan. 2017. *Low-level Concurrent Programming Using the Relaxed Memory Calculus*. Ph.D. Dissertation. Carnegie Mellon University.

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A Program Logic for C11 Concurrency. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Momchil Velikov. 2012. http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics. (Oct 2012).

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22th European Conference on Object-Oriented Programming (ECOOP '08)*.

Yang Zhang and Xinyu Feng. 2016. An Operational Happens-before Memory Model. *Frontiers of Computer Science* 10, 1 (2016), 54–81.

A A POTENTIAL OOTA EXAMPLE WITH ADDRESS DEPENDENCIES

Figure 22 presents a dependency cycle example involving address dependencies. In this example, global variables `x` and `y` are `0` and each element in the global array `z` is `0`. For the problematic execution in which `r1=r2=1`, the reason why `r1=x` can return value 1 is a chain of justifications that start with the assumption that `r1=x` can return value 1. As a result, the load in line 3 of Thread 1 does not read from the store in line 2 of Thread 1. Note that the dependence is transmitted through the absence of reading from the store in line 2. Rather than explicitly model the dependence from the store to the load, our semantics leverages the fact that all dependency chains end with a store and simply adds a dependency on the load "`r1=x`" to all stores after the store "`z[r1]=1`", which in our example is the store "`y=1`". Our dependency-preserving memory model forbids this execution because there exists a cycle in $dep \cup rf$.

```
int x=y=0; // Initially 0
int z[2]; // Initially 0
-----------------------------
// Thread 1     | // Thread 2
1:r1 = x;       | 1:r2 = y;
2:z[r1] = 1;    | 2:x = r2;
3:if (!z[0])    |
4:  y = 1;      |
```

Fig. 22. If `x=y=0` and each element in array `z` is `0` initially, can `r1=r2=1`? Note that according to our dependency notion, store "`y=1`" has an address dependency on load "`r1=x`" because the address of store "`z[r1]=1`" depend on "`r1=x`".

B THE SELECT SET OF IR-LEVEL PASSES ENABLED IN OUR DEPENDENCY-PRESERVING COMPILER

Figure 23 presents the select set of IR-level passes that we enable in our dependency-preserving compiler. Note that compared to the full set of IR-level passes enabled under full optimizations (`-O3`), this is a relatively small set, which means that there remain opportunities to further reduce the overhead by reviewing the disabled passes.

Pass Name	Is the Pass Modified
simplifycfg	Modified
instcombine	
dse	
licm	
slp-vectorizer	
loop-unroll	
gvn	Unmodified
loop-rotate	
mem2reg	
globalopt	
functionattrs	
tailcallelim	
lower-expect	
sroa	
inline	
forceattrs	
inferattrs	
prune-eh	
adce	
rpo-functionattrs	
elim-avail-extern	
float2int	
strip-dead-prototypes	
globaldce	
constmerge	
deadargelim	
argpromotion	
early-cse	
correlated-propagation	
loop-unswitch	
indvars	
loop-idiom	
loop-deletion	
barrier	
alignment-from-assumptions	

Fig. 23. The select set of IR-level transformation passes that we enable in our dependency-preserving compiler. Note that we globally modify the *InstructionSimplify* analysis to preserve data dependencies and to avoid phi nodes merging, which can affect some of the unmodified passes that rely on it, e.g., the *gvn* pass.

C OUT-OF-THIN-AIR PROPERTIES OF THE DEPENDENCY-BASED MEMORY MODEL

Since there is no agreed upon definition of out-of-thin-air executions, we provide a proof sketch for a property about the causality of executions in our memory model.

Definition C.1. *(Value independent semantics). A memory model has value-independent semantics iff the semantics of the memory model do not depend on the value loaded or stored with the exception of CAS. The C/C++ and Java memory models are both value independent.*

Definition C.2. *(Load available semantics). A memory model has load available semantics if a load can always read from some value that will not affect which values later loads can read from. For the C/C++ memory model, this is the earliest store in the modification order that is visible to the load.*

Theorem C.1 (Dependency Theorem). *For a memory model that has value-independent and load available semantics and that ensures that $dep \cup rf$ is acyclic, then if a store s is not reachable from a load l in the graph $dep \cup rf$ for an execution e, then for any value v that the load l returns there exist an execution e' with an equivalent load which returns value v such that either: (1) e' has an error or (2) e' has a store s' that writes the same value to the same address as s.*

Proof Sketch.

Define A to be the part of the execution that can reach l in the $dep \cup rf$ graph. Define B to be the part of the execution that l can reach in the $dep \cup rf$ graph. Define C to the part of the execution that can reach s in the $dep \cup rf$ graph.

Then:

(1) Load dependencies for the address of a store s_a or the condition of a branch with an untaken store in B is not sb before anything in C. This is true by the definition of dependency.
(2) Load dependencies for the address of a store s_a or the condition of a branch with an untaken store in B is not sb before anything in A. This is true by the definition of dependency and by the assumption that $dep \cup rf$ is acyclic.
(3) There is no load in C that reads from any store in B. This is true by the definition of dependency.
(4) Load dependencies for the address of a store s_a or the condition of a branch with an untaken store that are sb before A are in A. This is true by the definition of dependency.
(5) Load dependencies for the address of a store s_a or the condition of a branch with an untaken store that are sb before C are in C. This is true by the definition of dependency.

For any value v that load l returns, we can construct an execution e' in which (1) every store that is sb before $A \cup C$ in the execution e' has an equivalent store in e that writes to the same address, (2) every store in $A \cup C$ in e is in e' and writes to the same address, and (3) every load that is sb before $A \cup C$ reads from the same store as it did in e (note that some loads than are sb, but not in $A \cup C$ may be missing) since all of the load dependencies for the conditional branches or addresses of stores are in $A \cup C$. The stores in execution e' that are sb before $A \cup C$ are a subset of stores in execution e and they write to the same addresses so this is possible (loads in execution e' who are missing their corresponding store are not in $A \cup C$ and can simply be made to read from some store without affecting other loads since we assume that the memory model has load available semantics). Note that the stores may not write the same values, but the memory semantics are value-independent and thus admit the same rf relation. Note that we may have new loads appear that are sb before $A \cup C$, but such loads can always read from a value by the assumption that the memory model has load available semantics.

The execution e' may throw an error in which case we trivially prove the property. Thus assume that execution e' does not throw an error. Then by induction on $dep \cup rf$ and the definition of dep, the store s' must store the same value as store s.

Theorem C.1 implies that executions with causality cycles or satisfaction cycles in which a store s cyclically justifies the value it stores are not possible if $dep \cup rf$ is acyclic. Any load l that reads from s cannot reach s in the $dep \cup rf$ graph since it is acyclic and the load l reads from s. Thus by the theorem, the load l can return any value and store s will still store the same value.

D OPTIMIZATIONS FOR THE LOAD-STORE-ORDER-PRESERVING APPROACH

There are two core ideas behind our optimizations to alleviate the performance overhead of enforcing the load-store ordering. One is to take advantage of existing ordering constraints that are intrinsic to the source code to avoid adding unnecessary extra ordering constraints, and the other one is to move the added ordering constraints out of the critical sections when possible. We discuss them in more details as follows.

Avoid Unnecessary Ordering Constraints. As shown in Figure 17 (c) and (d), a relaxed load can automatically have ordering constraints to subsequent stores because of existing control or address dependencies, e.g., when the result of the load is used to compute the condition of an immediately following conditional branch. Another scenario to optimize is when a relaxed load is followed by `fetch_add` like atomic operations with acquire and release semantics and there does not exist any atomic store in between. One notable real example is the synchronizing barrier implementation [Velikov 2012], with an interesting code snippet shown in Figure 24. The `fetch_add` operation that immediately follows the relaxed load on variable `step_` has the `memory_order_acq_rel` memory order, and the LLVM backend will transform this `fetch_add` operation to acquire load-exclusive and release store-exclusive instructions. As a result, the `fetch_add` operation effectively acts as a fence that forbids the relaxed load and any subsequent stores to be reordered across it. Another such pattern is a relaxed load immediately followed by a `CAS` operation. In the implementation of our load-store-order-preserving compiler, we have an analysis to identify these patterns for relaxed loads to avoid adding unnecessary ordering constraints.

```
unsigned step = step_.load(relaxed);
if (nwait_.fetch_add(1, memory_order_acq_rel) == n_ - 1) {
 // Subsequent stores...
}
// Other subsequent stores...
```

Fig. 24. A relaxed load followed by a `fetch_add` like read-modify-write operation (no other atomic store in between) is naturally guaranteed to be ordered before subsequent stores after the read-modify-write operation.

Move Added Ordering Constraints out of Critical Sections. If a relaxed load in the critical path requires adding extra ordering constraints, we can potentially reduce the penalty if we can safely move the intentionally added ordering constraints out of the critical path. Figure 25 shows the code for the `unlock` method of the Ticket Lock [Reed and Kanodia 1979] implementation. This lock data structure maintains a `turn` variable to indicate whose turn it is to take the lock. For a thread that holds the lock, the `unlock` method simply increments the `turn` variable to allow the next waiting thread to acquire the lock. Since the thread holding the lock has exclusive access to the `turn` variable, it does not use an atomic `fetch_add` operation, which is potentially more costly than a plain load and store. In this case, to ensure that the relaxed load is ordered before all subsequent stores, a straightforward approach is to add a bogus conditional branch right after the relaxed load and before the release store; however, this intentionally added control dependency is in the critical section because the `turn` variable has not been incremented yet (so the waiting thread potentially needs to wait for a longer time). One observation is that the store in this case has release semantics, meaning that the load cannot be reordered across it. Hence, we can instead add the bogus conditional branch after the release store. As a result, the critical section does not contain any added control dependency that delays the process of releasing the lock. In our implementation, for a relaxed load, we try to find the latest release store in the same basic block of that load and add the bogus conditional branch after that last release store.

```
void unlock() {
 unsigned my_turn = turn.load(std::memory_order_relaxed);
 // Still in the critical section
 turn.store(my_turn + 1, std::memory_order_release);
 // Not in the critical section anymore
}
```

Fig. 25. Example of a relaxed load followed by a release store. Since the relaxed load cannot be reordered across the release store, we can safely delay adding a bogus conditional branch till after the release store rather than before the release store.

E RESULTS FOR ADDING BOGUS CONDITIONAL BRANCHES

E.1 Running with a Single Thread

We first present detailed results for single-threaded execution for preserving load-store ordering using bogus conditional branches. These results best capture the actual overhead that our compiler adds to the code to preserve load-store ordering. The plots show percentage slowdown relative to -O3 compilation. Positive numbers mean that the load-store order preserving version is slower than the -O3 version while negative numbers mean that the load-store order preserving version is faster than the -O3 version.

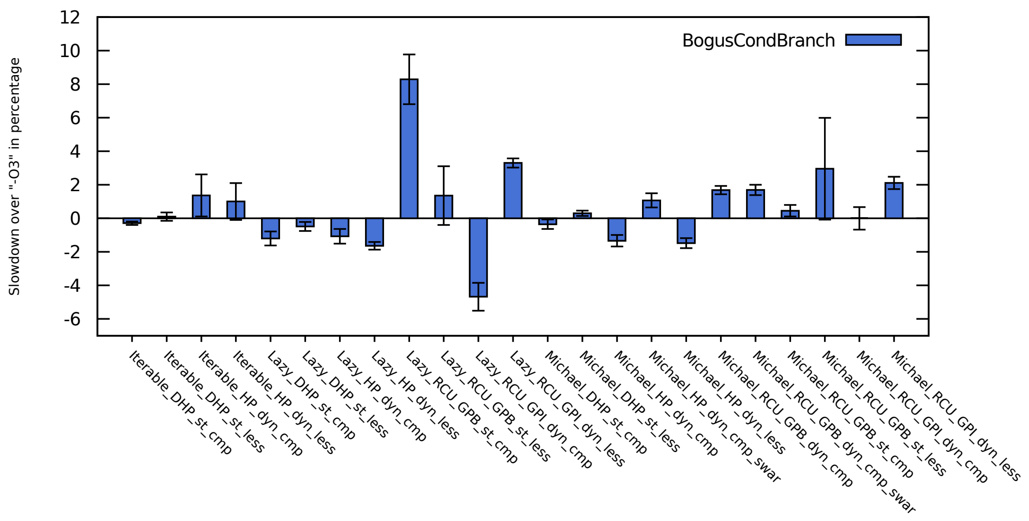

Fig. 26. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different split-ordered list variants from the CDS Library with a single thread.

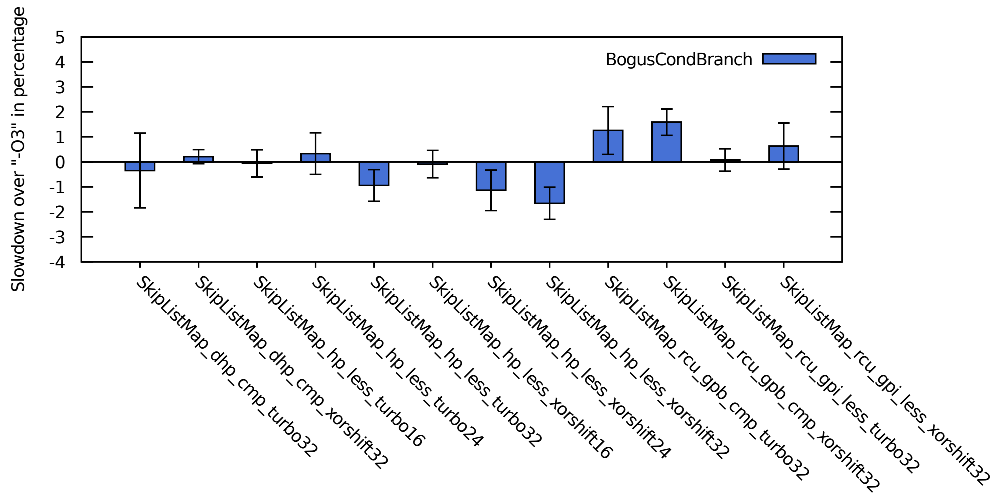

Fig. 27. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different skip list map variants from the CDS Library with a single thread.

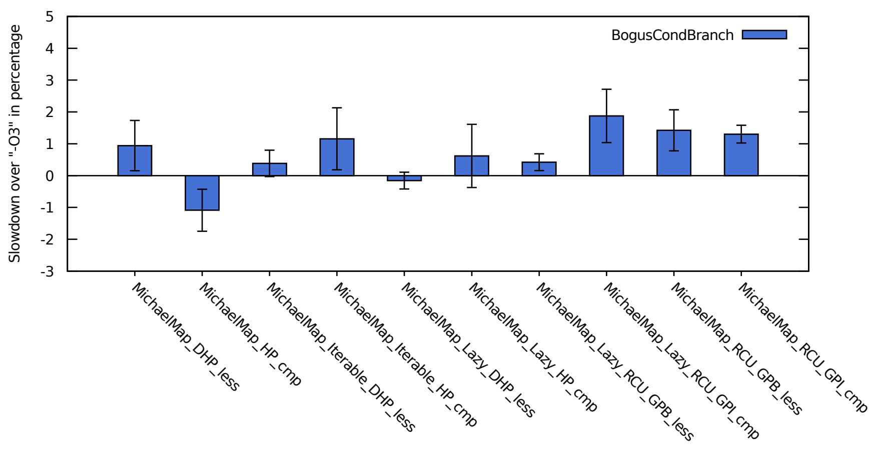

Fig. 28. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different Michael map variants from the CDS Library with a single thread.

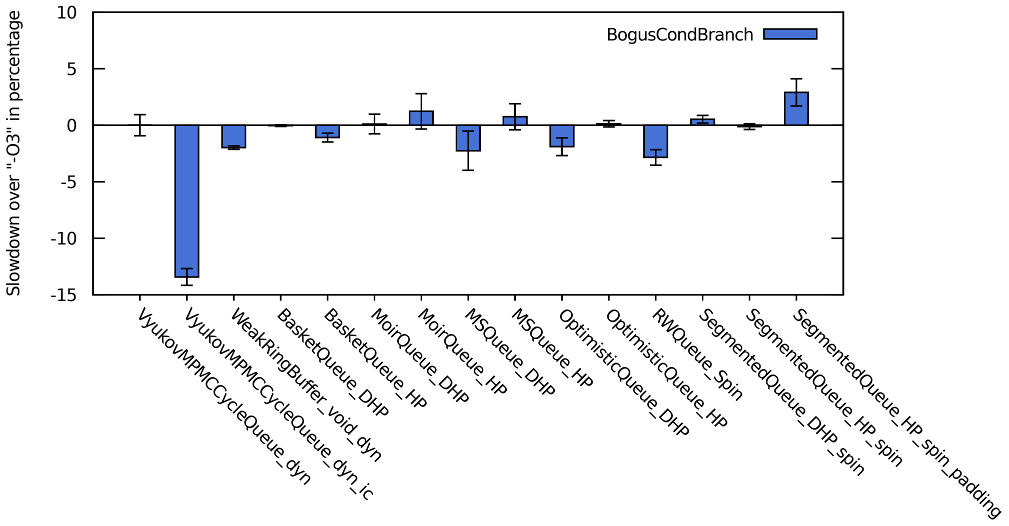

Fig. 29. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different queue benchmarks/variants from the CDS Library with a single thread.

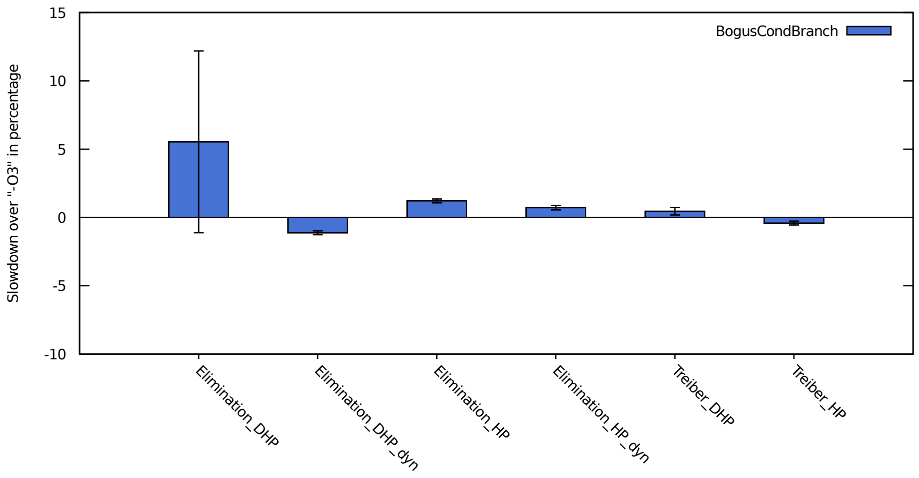

Fig. 30. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for the Treiber stack and elimination-backoff stack variants from the CDS Library with a single thread.

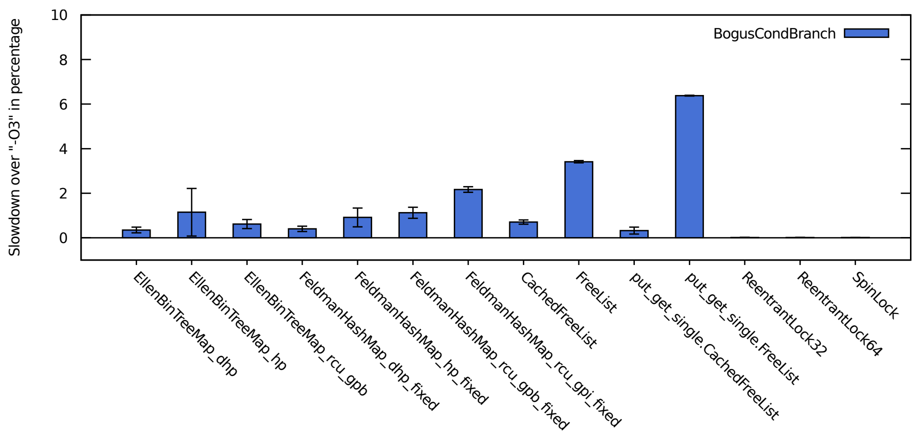

Fig. 31. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for other benchmarks/variants from the CDS Library with a single thread.

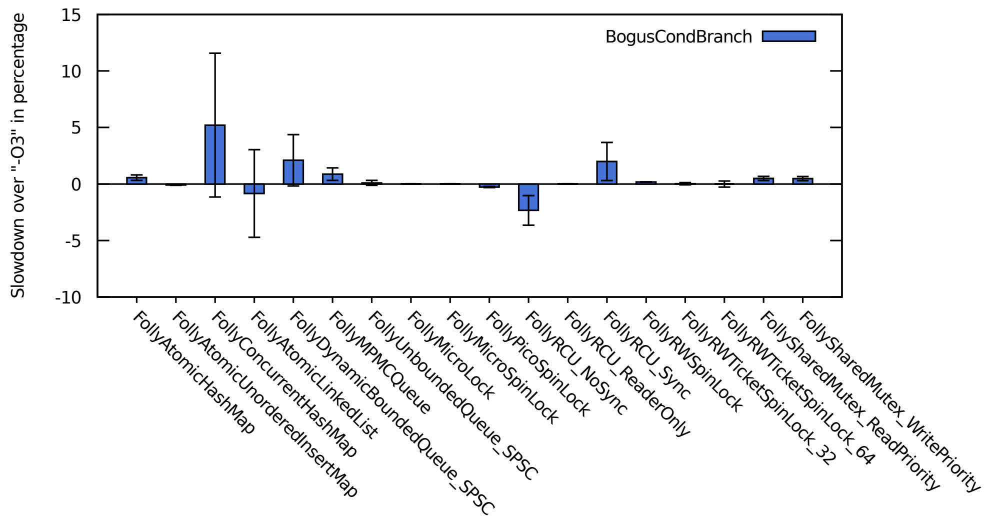

Fig. 32. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks/variants from the Folly Library with a single thread.

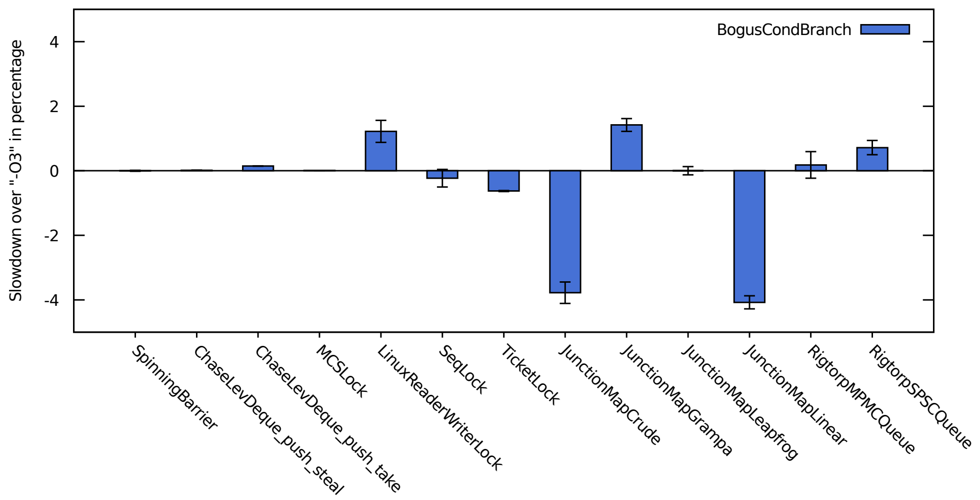

Fig. 33. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks from CDSSpec, Rigtorp's SPSC & MPMC Queues and the Junction Library with a single thread.

E.2 Running with Multiple Threads

We next present detailed results for multiple-threaded execution for preserving load-store ordering using bogus conditional branches for completeness. These results are significantly more challenging to interpret as theoretically more efficient code can result in worse performance due to extra contention. The results are also noisy — small differences in timing can result in large performance differences. The plots show percentage slowdown relative to -O3 compilation. Positive numbers mean that the load-store order preserving version is slower than the -O3 version while negative numbers mean that the load-store order preserving version is faster than the -O3 version.

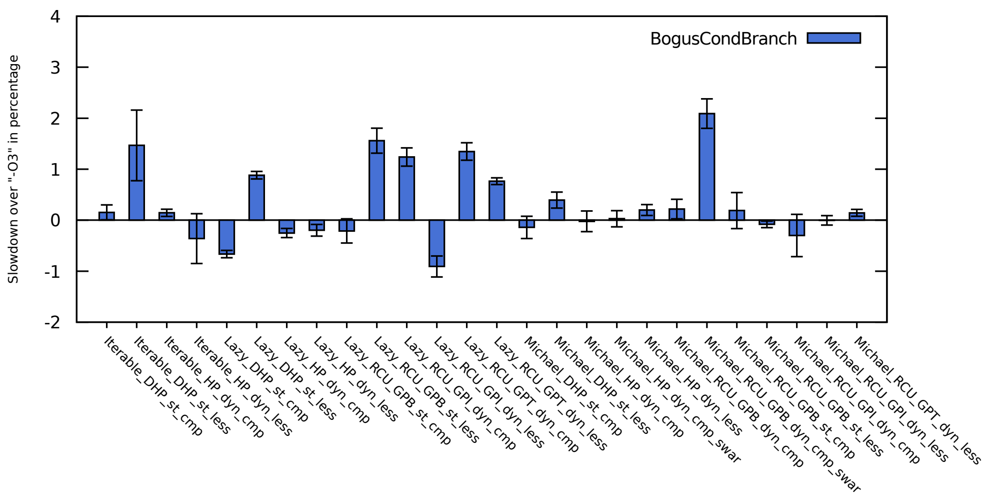

Fig. 34. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different split-ordered list variants from the CDS Library with multiple threads.

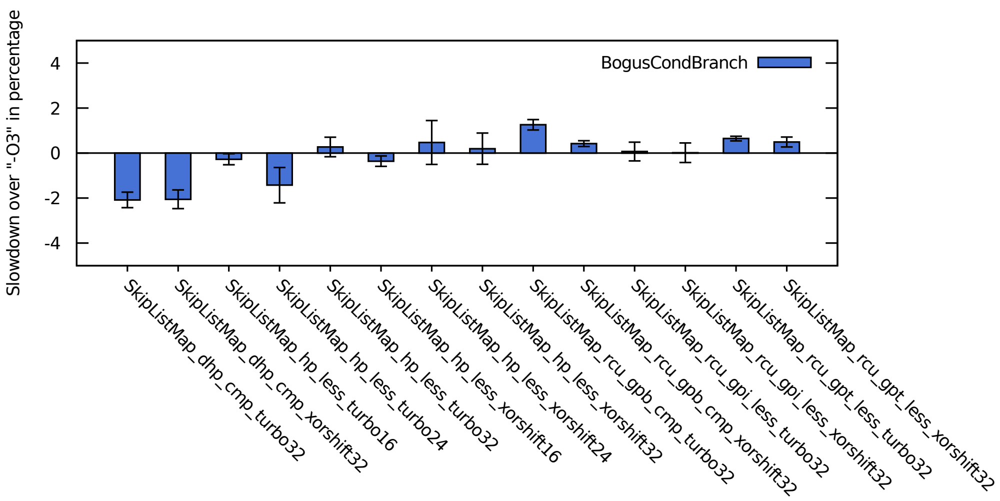

Fig. 35. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different skip list map variants from the CDS Library with multiple threads.

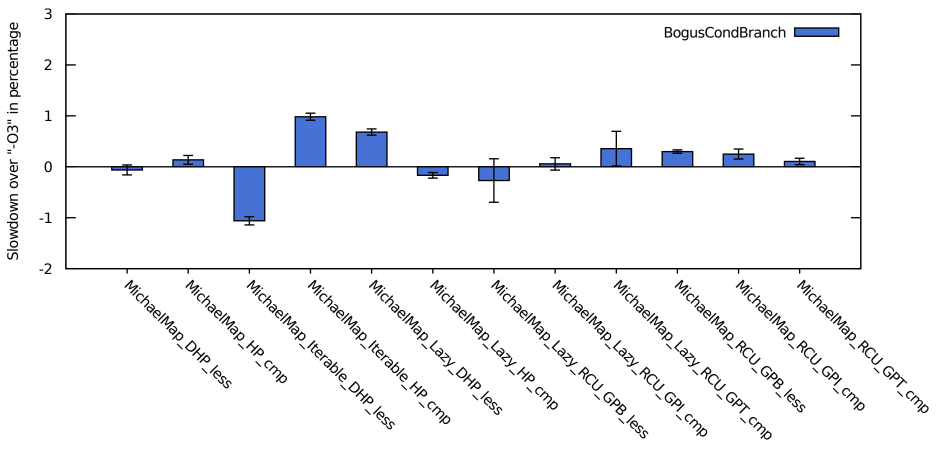

Fig. 36. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different Michael map variants from the CDS Library with multiple threads.

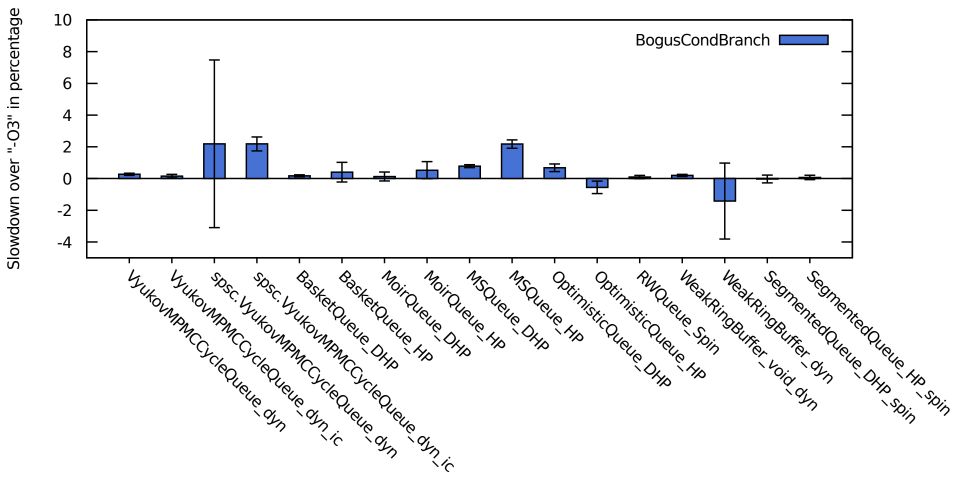

Fig. 37. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different queue benchmarks/variants from the CDS Library with multiple threads.

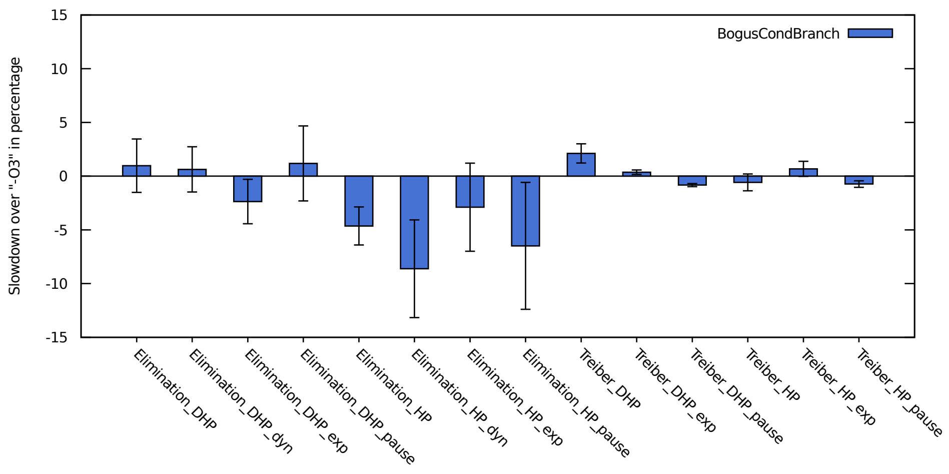

Fig. 38. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for the Treiber stack and elimination-backoff stack variants from the CDS Library with multiple threads.

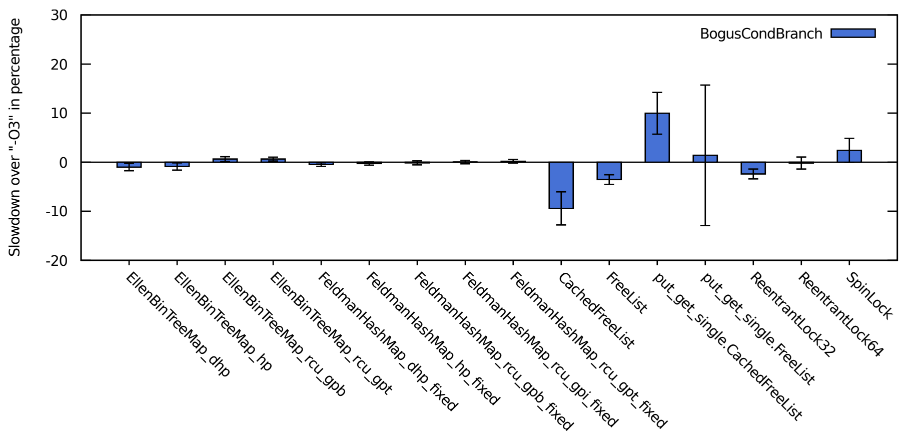

Fig. 39. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for other benchmarks/variants from the CDS Library with multiple threads.

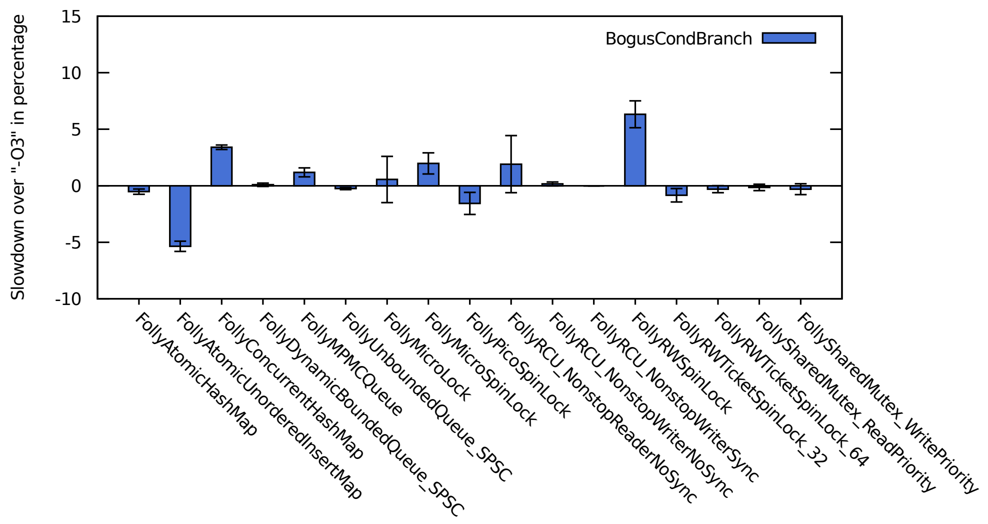

Fig. 40. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks/variants from the Folly Library with multiple threads.

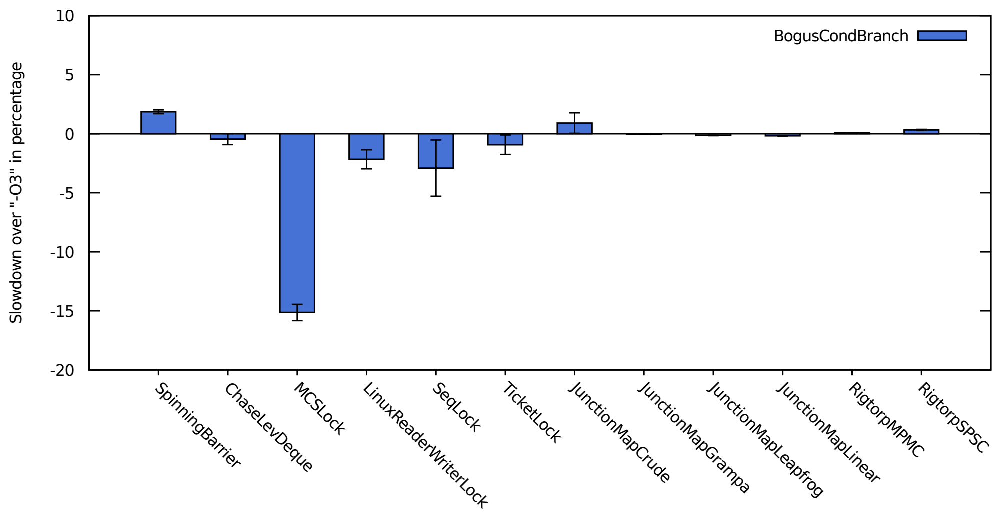

Fig. 41. Performance overhead (in percentage) of the *Bogus Conditional Branch* strategy over *Full Optimizations* for different benchmarks from CDSSpec, Rigtorp's SPSC & MPMC Queues and the Junction Library with multiple threads.