Fencing Programs with Self-Invalidation and Self-Downgrade

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Stefanos Kaxiras, Carl Leonardsson, Alberto Ros, Yunyun Zhu

▶ **To cite this version:**

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Stefanos Kaxiras, Carl Leonardsson, Alberto Ros, et al.. Fencing Programs with Self-Invalidation and Self-Downgrade. 36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Jun 2016, Heraklion, Greece. pp.19-35, 10.1007/978-3-319-39570-8_2 . hal-01432925

HAL Id: hal-01432925
https://inria.hal.science/hal-01432925

Submitted on 12 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Fencing Programs with Self-Invalidation and Self-Downgrade

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Stefanos Kaxiras[1], Carl Leonardsson[1], Alberto Ros[2], and Yunyun Zhu[1]

[1] Uppsala University, Sweden
[2] Universidad de Murcia, Spain

Abstract. Cache coherence protocols using self-invalidation and self-downgrade have recently seen increased popularity due to their simplicity, potential performance efficiency, and low energy consumption. However, such protocols result in memory instruction reordering, thus causing extra program behaviors that are often not intended by the programmer. We propose a novel formal model that captures the semantics of programs running under such protocols, and employs a set of fences that interact with the coherence layer. Using the model, we perfform a reachability analysis that can check whether a program satisfies a given safety property with the current set of fences. Based on an algorithm in [19], we describe a method for insertion of *optimal* sets of fences that ensure correctness of the program under such protocols. The method relies on a *counter-example* guided fence insertion procedure. One feature of our method is that it can handle a variety of fences (with different costs). This diversity makes optimization more difficult since one has to optimize the total cost of the inserted fences, rather than just their number. To demonstrate the strength of our approach, we have implemented a prototype and run it on a wide range of examples and benchmarks. We have also, using simulation, evaluated the performance of the resulting fenced programs.

1 Introduction

Background. Many traditional cache coherence protocols such as MESI or MOESI are *transparent* to the programmer in the sense that there is no effect on memory ordering due to the coherence protocol. On the other hand, there is an ever larger demand on hardware designers to increase *efficiency* both in performance and power consumption. The quest to increase performance while maintaining transparency has led to complex coherence protocols with many states and relying on directories, invalidations, broadcasts, etc, often at the price of high verification cost, area (hardware cost) and increased energy consumption. Therefore, many researchers have recently proposed ways to simplify coherence without compromising performance but at the price of relaxing the memory consistency model [18, 7, 13, 24, 32, 14, 12, 23, 31, 25, 8, 15]. Principal techniques among these proposals are Self-Invalidation (SI) and Self-Downgrade (SD).

In traditional cache coherence protocols, when a write is performed on a cache line, the copies in other cores are invalidated (discarded). Thus, the protocol needs to track sharers of a cache line in a directory structure. A protocol with Self-Invalidation allows old copies to be kept, without invalidation at each store by another core. This eliminates the need for tracking readers [18]. In an SI protocol, invalidation is caused by synchronization instructions which occur in the code of the same thread. For instance, when a core executes a fence, it informs its own L1 cache that it has to self-invalidate.

Correspondingly, in traditional protocols, when a read operation is performed on a cache line, the last writer of the line is downgraded (or copied to the shared cache). In a protocol with Self-Downgrade (SD), downgrades are not caused by read operations in other cores. SD eliminates the need to track the last writer of a cache line [24]. Like invalidations, in an SD protocol, downgrades can be caused by fence instructions.

A protocol with both self-invalidation and self-downgrade (SISD) does not need a directory, thus removing a main source of complexity and scalability constraints in traditional cache coherence protocols [24]. But this comes at a price: SISD protocols induce *weak memory semantics* that reorder memory instructions. The behavior of a program may now deviate from its behavior under the standard *Sequentially Consistent (SC)* semantics, leading to subtle errors that are hard to detect and correct.

In the context of weak memory, hardware designers provide memory *fence* instructions to help the programmer eliminate the undesired behaviors. A fence instruction, executed by a process, limits the allowed reorderings between instructions issued before and after the fence instruction. To enforce consistency under SISD, fences should also be made visible to caches, such that necessary invalidations or downgrades may be performed. In this paper, we consider different types of fences. The different types eliminate different kinds of non-SC behaviors, and may have different impact on the program performance. In fact, unnecessary fences may significantly downgrade program performance. This is particularly true for the fences considered here, since they both incur latency, and affect the performance of the cache coherence subsystem as a whole. These fences cause the invalidation of the contents of the cache. Hence the more fences the less caching and the higher traffic we have. Thus, it is desirable to find the *optimal* set of fences, which guarantee correctness at minimal performance cost.

Challenge. One possibility to make SISD transparent to the program is to require the programmer to ensure that the program does not contain any data races. In fact, data race freedom is often required by designers of SISD protocols in order to guarantee correct program behavior [7, 13]. However, this approach would unnecessarily disqualify large sets of programs, since many data races are in reality not harmful. Examples of correct programs with races include lock-free data structures (e.g., the Chase-Lev Work-stealing queue algorithm [6]), transactional memories (e.g., the TL2 algorithm [9]), and synchronization library primitives (e.g. `pthread_spin_lock` in `glibc`). In this paper, we consider a different approach where fences are inserted to retrieve correctness. This means that we may

insert sufficiently many fences to achieve program correctness without needing to eliminate all its races or non-SC behaviors. The challenge then is to find sets of fences that guarantee program correctness without compromising efficiency. Manual fence placement is time-consuming and error-prone due to the complex behaviors of multithreaded programs [11]. Thus, we would like to provide the programmer with a tool for *automatic* fence placement. There are several requirements to be met in the design of fence insertion algorithms. First, a set of fences should be *sound*, i.e., it should have enough fences to enforce a sufficiently ordered behavior for the program to be correct. Second, the set should be *optimal*, in the sense that it has a lowest total cost among all sound sets of fences. In general, there may exist several different optimal sets of fences for the same program. Our experiments (Section 4) show that different choices of sound fence sets may impact performance and network traffic. To carry out fence insertion we need to be able to perform *program verification*, i.e., to check correctness of the program with a given set of fences. This is necessary in order to be able to decide whether the set of fences is sound, or whether additional fences are needed to ensure correctness. A critical task in the design of formal verification algorithms, is to define the program semantics under the given memory model.

Our Approach. We present a method for automatic fence insertion in programs running in the presence of SiSd. The method is applicable to a large class of self-invalidation and self-downgrade protocols such as the ones in [18, 7, 13, 24, 32, 14, 12, 23, 31, 25, 8, 15]. Our goal is to eliminate incorrect behaviors that occur due to the memory model induced by SiSd. We will not concern ourselves with other sources of consistency relaxation, such as compiler optimizations. We formulate the correctness of programs as *safety properties*. A safety property is an assertion that some specified "erroneous", or "bad", program states can never occur during execution. Such bad states may include e.g., states where a programmer specified assert statement fails, or where uninitialized data is read. To check a safety property, we check the reachability of the set of "bad" states.

We provide an algorithm for checking the reachability of a set of bad states for a given program running under SiSd. In the case that such states are reachable, our algorithm provides a counter-example (i.e., an execution of the program that leads to one of the bad states). This counter-example is used by our fence insertion procedure to add fences in order to remove the counter-examples introduced by SiSd semantics. Thus, we get a counter-example guided procedure for inferring the optimal sets of fences. The termination of the obtained procedure is guaranteed under the assumption that each call to the reachability algorithm terminates. As a special case, our tool detects when a program behaves incorrectly already under SC. Notice that in such a case, the program cannot be corrected by inserting any set of fences.

Contributions. We make the following main contributions: (i) A novel formal model that captures the semantics of programs running under SiSd, and employs a set of fences that interact with the coherence layer. The semantics support the essential features of typical assembly code. (ii) A tool, Memorax, available

at `https://github.com/memorax/memorax`, that we have run successfully on a wide range of examples under SiSd and under Si. Notably, our tool detects for the first time four bugs in programs in the Splash-2 benchmark suite [33], which have been fixed in a recent Splash-3 release [27]. Two of these are present even under SC, while the other two arise under SiSd. We employ the tool to infer fences of different kinds and evaluate the relative performance of the fence-augmented programs by simulation in GEMS.

We augment the semantics with a reachability analysis algorithm that can check whether a program satisfies a given safety property with the current set of fences. Inspired by an algorithm in [19] (which uses dynamic analysis instead of verification as backend), we describe a counter-example guided fence insertion procedure that automatically infers the optimal sets of fences necessary for the correctness of the program. The procedure relies on the counter-examples provided by the reachability algorithm in order to refine the set of fences. One feature of our method is that it can handle different types of fences with different costs. This diversity makes optimization more difficult since one has to optimize the total cost of the inserted fences, rather than just their number. Upon termination, the procedure will return all optimal sets of fences.

Related Work. Adve and Hill proposed SC-for-DRF as a contract between software and hardware: If the software is data race free, the hardware behaves as sequentially consistent [2]. Dynamic self-invalidation (for DRF programs) was first proposed by Lebeck and Wood [18]. Several recent works employ self-invalidation to simplify coherence, including SARC coherence [13], DeNovo [7, 32, 31], and VIPS-M [24, 14, 23, 25, 15].

A number of techniques for automatic fence insertion have been proposed, for different memory models and with different approaches. However, to our knowledge, we propose the first counter-example guided fence insertion procedure in the presence of a variety of fences (with different costs). In our previous work [1], we propose counter-example guided fence insertion for programs under TSO with respect to safety properties (also implemented in Memorax). Considering the SiSd model makes the problem significantly more difficult. TSO offers only one fence, whereas the SiSd model offers a variety of fences with different costs. This diversity makes the optimization more difficult since one has to minimize the total cost of the fences rather than just their number.

The work presented in [16] proposes an insertion procedure for different memory models w.r.t. safety properties. This procedure computes the set of needed fences in order to not reach each state in the transition graph. Furthermore, this procedure assigns a unique cost for all fences. The procedure is not counter-example based, and requires some modification to the reachability procedure.

In [4], the tool Trencher is introduced, which inserts fences under TSO to enforce robustness (formalised by Shasha and Snir in [30]), also using an exact, model-checking based technique. Musketeer [3] uses static analysis to efficiently overapproximate the fences necessary to enforce robustness under several different memory models. In contrast to our work, the fence insertion procedures

in [4] and [3] first enumerate all solutions and then use linear programming to find the optimal set of fences.

The program semantics under SiSd is different from those under other weak memory models (e.g. TSO and POWER). Hence existing techniques cannot be directly applied. To our knowledge, it is the first work that defines the SiSd model, proposes a reachability analysis and describes a fence insertion procedure under SiSd.

There exist works on the verification of cache coherence protocols. This paper is orthogonal to these works since we are concerned with verification of *programs* running on such architectures and not the protocols themselves.

2 Programs – Syntax and Semantics

In this section, we formalize SiSd and Si protocols, by introducing a simple assembly-like programming language, and defining its syntax and semantics.

2.1 Syntax

The syntax of programs is given by the grammar in Figure 1. A program has a finite set of processes which share a number of variables (memory locations) $\mathcal{M}$. A variable $x \in \mathcal{M}$ should be interpreted as one machine word at a particular memory address. For simplicity, we assume that all the variables and process registers assume their values from a common finite domain $\mathcal{V}$ of values. Each process contains a sequence of instructions, each consisting of a program label and a statement. To simplify the presentation, we assume that all instructions (in all processes) have unique labels. For a label λ, we apply three functions: $\mathsf{Proc}(\lambda)$ returns the process p in which the label occurs. $\mathsf{Stmt}(\lambda)$ returns the statement whose label id is λ. $\mathsf{Next}(\lambda)$ returns the label of the next statement in the process code, or `end` if there is no next statement.

$$
\begin{aligned}
\langle pgm\rangle &::= \texttt{data}\ \langle vdecl\rangle^{+}\langle proc\rangle^{+}\\
\langle vdecl\rangle &::= \langle var\rangle\ \texttt{'='}\ (\texttt{'*'} \mid \langle val\rangle)\\
\langle proc\rangle &::= \texttt{process}\ \langle pid\rangle\ \texttt{registers}\ \langle reg\rangle^{*}\ \langle stmts\rangle\\
\langle stmts\rangle &::= \texttt{begin}\ (\langle label\rangle\ \texttt{':'}\ \langle stmt\rangle\ \texttt{';'})^{+}\ \texttt{end}\\
\langle stmt\rangle &::= \langle var\rangle\ \texttt{':='}\ \langle expr\rangle \mid \langle reg\rangle\ \texttt{':='}\ \langle var\rangle \mid\\
&\quad \langle reg\rangle\ \texttt{':='}\ \langle expr\rangle \mid \texttt{llfence} \mid \texttt{fence} \mid\\
&\quad \texttt{cas}\ \texttt{'('}\ \langle var\rangle\ \texttt{','}\ \langle expr\rangle\ \texttt{','}\ \langle expr\rangle\ \texttt{')'} \mid\\
&\quad \texttt{syncwr}\ \texttt{':'}\ \langle var\rangle\ \texttt{':='}\ \langle expr\rangle \mid \texttt{ssfence} \mid\\
&\quad \texttt{cbranch}\ \texttt{'('}\ \langle bexpr\rangle\ \texttt{')'}\ \langle label\rangle
\end{aligned}
$$

Fig. 1: The grammar of concurrent programs.

2.2 Configurations

A *local configuration* of a process p is a triple $(\lambda, \mathsf{RVal}, \mathsf{L1})$, where λ is the label of the next statement to execute in p, RVal defines the values of the local registers, and $\mathsf{L1}$ defines the state of the L1 cache of p. In turn, $\mathsf{L1}$ is

a triple $(\mathsf{Valid}, \mathsf{LStatus}, \mathsf{LVal})$. Here $\mathsf{Valid} \subseteq \mathcal{M}$ defines the set of shared variables that are currently in the valid state, and $\mathsf{LStatus}$ is a function from Valid to the set $\{\texttt{dirty}, \texttt{clean}\}$ that defines, for each $x \in \mathsf{Valid}$, whether x is dirty or clean, and LVal is a function from Valid to $\mathcal{V}$ that defines for each $x \in \mathsf{Valid}$ its current value in the L1 cache of p. The *shared part* of a configuration is given by a function LLC that defines for each variable $x \in \mathcal{M}$ its value $\mathsf{LLC}(x)$ in the LLC. A configuration c then is a pair $(\mathsf{LConf}, \mathsf{LLC})$ where LConf is a function that returns, for each process p, the local configuration of p.

2.3 Semantics

In the formal definition below, our semantics allows system events to occur nondeterministically. This means that we model not only instructions from the program code itself, but also events that are caused by unpredictable things as hardware prefetching, software prefetching, program preemption, false sharing, multiple threads of the same program being scheduled on the same core, etc. A transition t is either performed by a given process when it executes an instruction, or is a system event. In the former case, t will be of the form λ, i.e., t models the effect of a process p performing the statement labeled with λ. In the latter case, t will be equal to ω for some system event ω. For a function f, we use $f\,[a \leftarrow b]$, to denote the function f' such that $f'(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$. We write $f(a) = \bot$ to denote that f is undefined for a.

Instruction Semantics

$$\frac{\sigma=(\$r{:=}x)\ ,\ x{\in}\mathsf{Valid}}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal}[\$r{\leftarrow}\mathsf{LVal}(x)],\mathsf{L1})],\mathsf{LLC})}$$

$$\frac{\sigma=(x{:=}e)\ ,\ x{\in}\mathsf{Valid}\ ,\ \mathsf{S}'{=}\mathsf{LStatus}[x{\leftarrow}\texttt{dirty}]}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},(\mathsf{Valid},\mathsf{S}',\mathsf{LVal}[x{\leftarrow}\mathsf{RVal}(e)]))],\mathsf{LLC})}$$

$$\frac{\sigma=\texttt{fence}\ ,\ \mathsf{Valid}{=}\varnothing}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},\mathsf{L1})],\mathsf{LLC})}$$

$$\frac{\sigma=\texttt{ssfence}\ ,\ \forall x{\in}\mathcal{M}.\ (x{\in}\mathsf{Valid}{\Rightarrow}\mathsf{LStatus}(x){=}\texttt{clean})}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},\mathsf{L1})],\mathsf{LLC})}$$

$$\frac{\sigma=\texttt{llfence}\ ,\ \forall x{\in}\mathcal{M}.\ (x{\in}\mathsf{Valid}{\Rightarrow}\mathsf{LStatus}(x){=}\texttt{dirty})}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},\mathsf{L1})],\mathsf{LLC})}$$

$$\frac{\sigma=(\texttt{syncwr}{:}x := e)\ ,\ x{\notin}\mathsf{Valid}}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},\mathsf{L1})],\mathsf{LLC}[x{\leftarrow}\mathsf{RVal}(e)])}$$

$$\frac{\sigma=\texttt{cas}(x,e_0,e_1)\ ,\ x{\notin}\mathsf{Valid}\ ,\ \mathsf{LLC}(x){=}\mathsf{RVal}(e_0)}{c\xrightarrow{\lambda}(\mathsf{LConf}[p{\leftarrow}(\mathsf{Next}(\lambda),\mathsf{RVal},\mathsf{L1})],\mathsf{LLC}[x{\leftarrow}\mathsf{RVal}(e_1)])}$$

System Event Semantics

$$\frac{\omega=(\texttt{fetch}(p,x))\ ,\ x{\notin}\mathsf{Valid}\ ,\ \mathsf{S}'{=}\mathsf{LStatus}[x{\leftarrow}\texttt{clean}]}{c\xrightarrow{\omega}(\mathsf{LConf}[p{\leftarrow}(\lambda,\mathsf{RVal},(\mathsf{Valid}{\cup}\{x\},\mathsf{S}',\mathsf{LVal}[x{\leftarrow}\mathsf{LLC}(x)]))],\mathsf{LLC})}$$

$$\frac{\omega=(\texttt{wrllc}(p,x))\ ,\ x{\in}\mathsf{Valid}\ ,\ \mathsf{LStatus}(x){=}\texttt{dirty}\ ,\ \mathsf{S}'{=}\mathsf{LStatus}[x{\leftarrow}\texttt{clean}]}{c\xrightarrow{\omega}(\mathsf{LConf}[p{\leftarrow}(\lambda,\mathsf{RVal},(\mathsf{Valid},\mathsf{S}',\mathsf{LVal}))],\mathsf{LLC}[x{\leftarrow}\mathsf{LVal}(x)])}$$

$$\frac{\omega=(\texttt{evict}(p,x))\ ,\ x{\in}\mathsf{Valid}\ ,\ \mathsf{LStatus}(x){=}\texttt{clean}}{c\xrightarrow{\omega}(\mathsf{LConf}[p{\leftarrow}(\lambda,\mathsf{RVal},(\mathsf{Valid}\backslash\{x\},\mathsf{LStatus}[x{\leftarrow}\bot],\mathsf{LVal}[x{\leftarrow}\bot]))],\mathsf{LLC})}$$

Fig. 2: Semantics of programs running under SiSd.

Below, we give an intuitive explanation of each transition. The formal definition can be found in Figure 2 where we assume $c = (\mathsf{LConf}, \mathsf{LLC})$, and

$\mathsf{LConf}(p) = (\lambda, \mathsf{RVal}, \mathsf{L1})$, and $\mathsf{L1} = (\mathsf{Valid}, \mathsf{LStatus}, \mathsf{LVal})$, $\mathsf{Proc}(\lambda) = p$, and $\mathsf{Stmt}(\lambda) = \sigma$. We leave out the definitions for local instructions, since they have standard semantics.

Instruction Semantics. Let p be one of the processes in the program, and let λ be the label of an instruction in p whose statement is σ. We will define a *transition relation* $\xrightarrow{\lambda}$, induced by λ, on the set of configurations. The relation is defined in terms of the type of operation performed by the given statement σ. In all the cases only the local state of p and LLC will be changed. The local states of the rest of the processes will not be affected. This mirrors the principle in SiSd that L1 cache controllers will communicate with the LLC, but never directly with other L1 caches.

Read ($\$r := x$)**:** Process p reads the value of x from L1 into the register $\$r$. The L1 and the LLC will not change. The transition is only enabled if x is valid in the L1 cache of p. This means that if x is not in L1, then a system event `fetch` must occur before p is able to execute the read operation.

Write ($x := e$)**:** An expression e contains only registers and constants. The value of x in L1 is updated with the evaluation of e where registers have values as indicated by RVal, and x becomes dirty. The write is only enabled if x is valid for p.

Fence (`fence`)**:** A full fence transition is only enabled when the L1 of p is empty. This means that before the fence can be executed, all entries in its L1 must be evicted (and written to the LLC if dirty). So p must stall until the necessary system events (`wrllc` and `evict`) have occurred. Executing the fence has no further effect on the caches.

SS-Fence (`ssfence`)**:** Similarly, an `ssfence` transition is only enabled when there are no dirty entries in the L1 cache of p. So p must stall until all dirty entries have been written to the LLC by `wrllc` system events. In contrast to a full fence, an `ssfence` permits clean entries to remain in the L1.

LL-Fence (`llfence`)**:** This is the dual of an SS-Fence. An `llfence` transition is only enabled when there are no clean entries in the L1 cache of p. In other words, the read instructions before and after an `llfence` cannot be reordered.

Synchronized write (`syncwr` : $x := e$)**:** A synchronized write is like an ordinary write, but acts directly on the LLC instead of the L1 cache. For a `syncwr` transition to be enabled, x may not be in the L1. (I.e., the cache must invalidate x before executing the `syncwr`.) When it is executed, the value of x in the LLC is updated with the evaluation of the expression e under the register valuation RVal of p. The L1 cache is not changed.

CAS ($\mathtt{cas}(x, e_0, e_1)$)**:** A compare and swap transition acts directly on the LLC. The `cas` is only enabled when x is not in the L1 cache of p, and the value of x in the LLC equals e_0 (under RVal). When the instruction is executed, it atomically writes the value of e_1 directly to the LLC in the same way as a synchronized write would.

System Event Semantics. The system may non-deterministically (i.e., at any time) perform a *system event.* A system event is not a program instruction, and so will not change the program counter (label) of a process. We will define a *transition relation* $\xrightarrow{\omega}$, induced by the system event ω. There are three types of system events as follows.

Eviction ($\texttt{evict}(p, x)$)**:** An $\texttt{evict}(p, x)$ system event may occur when x is valid and clean in the L1 of process p. When the event occurs, x is removed from the L1 of p.

Write-LLC ($\texttt{wrllc}(p, x)$)**:** If the entry of x is dirty in the L1 of p, then a $\texttt{wrllc}(p, x)$ event may occur. The value of x in the LLC is then updated with the value of x in the L1 of p. The entry of x in the L1 of p becomes clean.

Fetch ($\texttt{fetch}(p, x)$)**:** If x does not have an entry in the L1 of p, then p may fetch the value of x from the LLC, and create a new, clean entry with that value for x in its L1.

2.4 Program Semantics under an Si Protocol

In a self-invalidation protocol without self-downgrade, a writing process will be downgraded and forced to communicate its dirty data when another process accesses that location in the LLC. This behavior can be modelled by a semantics where writes take effect atomically with respect to the LLC. Hence, to modify the semantics given in Section 2.3 such that it models a program under an Si protocol, it suffices to interpret all write instructions as the corresponding `syncwr` instructions.

2.5 Transition Graph and the Reachability Algorithm

Our semantics allows to construct, for a given program $\mathcal{P}$, a finite *transition graph*, where each node in the graph is a configuration in $\mathcal{P}$, and each edge is a transition. A *run* is a sequence $c_0 \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \cdots \xrightarrow{t_n} c_n$, which is a path in the transition graph, where $t_i (0 \leq i \leq n)$ is either a label λ or a system event ω.

Together with the program, the user provides a *safety property* ϕ that describes a set *Bad* of configurations that are considered to be errors. Checking ϕ for a program $\mathcal{P}$ amounts to checking whether there is a run leading from the initial configuration to a configuration in *Bad*. To do that, the input program under SiSd is translated to the code recognized by the reachability analysis tool chosen by the user. The translated code simulates all the behaviors which are allowed in the SiSd semantics. Also, there is instrumentation added to simulate the caches. Verifying the input program amounts to verifying the translated code which is analyzed under SC. If a bad configuration is encountered, a witness run is returned by the tool. Otherwise, the program is declared to be correct.

3 Fence Insertion

In this section we describe our fence insertion procedure, which is closely related to the algorithm described in [19]. Given a program $\mathcal{P}$, a cost function κ and a safety property ϕ, the procedure finds *all* the sets of fences that are optimal for $\mathcal{P}$ w.r.t. ϕ and κ.

In this section we take *fence constraint* (or *fence* for short) to mean a pair (λ, f) where λ is a statement label and f is a fence instruction. A fence constraint (λ, f) should be interpreted as the notion of inserting the fence instruction f into a program, between the statement labeled λ and the next statement (labeled by $\mathsf{Next}\,(\lambda)$)[3]. For a program $\mathcal{P}$ and a set F of fence constraints, we define $\mathcal{P} \oplus F$ to mean the program $\mathcal{P}$ where all fence constraints in F have been inserted. To avoid ambiguities in the case when F contains multiple fence constraints with the same statement label (e.g $(\lambda, \texttt{llfence})$ and $(\lambda, \texttt{ssfence})$), we assume that fences are always inserted in some fixed order.

Definition 1 (Soundness of Fence Sets). *For a program $\mathcal{P}$, safety property ϕ, and set F of fence constraints, the set F is sound for $\mathcal{P}$ w.r.t. ϕ if $\mathcal{P} \oplus F$ satisfies ϕ under* SiSd.

A *cost function* κ is a function from fence constraints to positive integer costs. We extend the notion of a cost function to sets of fence constraints in the natural way: For a cost function κ and a set F of fence constraints, we define $\kappa(F) = \sum_{c \in F} \kappa(c)$.

Definition 2 (Optimality of Fence Sets). *For a program $\mathcal{P}$, safety property ϕ, cost function κ, and set F of fence constraints, F is optimal for $\mathcal{P}$ w.r.t. ϕ and κ if F is sound for $\mathcal{P}$ w.r.t. ϕ, and there is no sound fence set G for $\mathcal{P}$ w.r.t. ϕ where $\kappa(G) < \kappa(F)$.*

In order to introduce our algorithm, we define the notion of a *hitting set*.

Definition 3 (Hitting Set). *For a set $S = \{S_0, \cdots, S_n\}$ of sets $S_0, \cdots, S_n$, and a set T, we say that T is a hitting set of S if $T \cap S_i \neq \varnothing$ for all $0 \leq i \leq n$.*

For example $\{a, d\}$ is a hitting set of $\{\{a, b, c\}, \{d\}, \{a, e\}\}$. For a set S of sets, hitting sets of S can be computed using various search techniques, such as e.g. constraint programming. We will assume that we are given a function $\mathbf{hits}(S, \kappa)$ which computes all hitting sets for S which are cheapest w.r.t. κ. I.e., for a set S of finite sets, and a cost function κ, the call $\mathbf{hits}(S, \kappa)$ returns the set of all sets T with $T \subseteq \bigcup_{S_i \in S} S_i$ such that i) T is a hitting set of S, and ii) there is no hitting set T' of S such that $\kappa(T') < \kappa(T)$.

We present our fence insertion algorithm in Figure 3. The algorithm keeps two variables `opt` and `req`. Both are sets of fence constraint sets, but are intuitively interpreted in different ways. The set `opt` contains all the optimal fence

[3] This definition can be generalized. Our prototype tool does indeed support a more general definition of fence positions, which is left out of the article for simplicity.

constraint sets for $\mathcal{P}$ w.r.t. ϕ and κ that have been found thus far. The set `req` is used to keep track of the requirements that have been discovered for which fences are necessary for soundness of $\mathcal{P}$. We maintain the following invariant for `req`: Any fence constraint set F which is sound for $\mathcal{P}$ w.r.t. ϕ is a hitting set of `req`. As the algorithm learns more about $\mathcal{P}$, the requirements in `req` will grow, and hence give more information about what a sound fence set may look like. Notice that the invariant holds trivially in the beginning, when $\texttt{req} = \varnothing$.

Fencins($\mathcal{P}$,ϕ,κ)

```
 1: opt := ∅; // Optimal fence sets
 2: req := ∅; // Known requirements
 3: while(∃F ∈ hits(req, κ) \ opt){
 4:   π := reachable(P ⊕ F, φ);
 5:   if(π = ⊥){
        // The fence set F is sound
        //  (and optimal)!
 6:     opt := opt ∪ {F};
 7:   }else{ // π is a witness run.
 8:     C := analyze_witness(P ⊕ F, π);
        // C is the set of fences
        //  that can prevent π.
 9:     if(C = ∅){ // error under SC!
10:       return ∅;
11:     }
12:     req := req ∪ {C};
13:   }
14: }
15: return opt;
```

Fig. 3: The fence insertion algorithm.

In the loop on lines 3-14 we repeatedly compute a candidate fence set `F` (line 3), insert it into $\mathcal{P}$, and call the reachability analysis to check if `F` is sound (line 4). We assume that the call **reachable**($\mathcal{P} \oplus \texttt{F}, \phi$) returns $\bot$ if ϕ is unreachable in $\mathcal{P} \oplus \texttt{F}$, and a witness run otherwise. If $\mathcal{P} \oplus \texttt{F}$ satisfies the safety property ϕ, then `F` is sound. Furthermore, since `F` is chosen as one of the cheapest (w.r.t. κ) hitting sets of `req`, and all sound fence sets are hitting sets of `req`, it must also be the case that `F` is optimal. Therefore, we add `F` to `opt` on line 6.

If $\mathcal{P} \oplus \texttt{F}$ does not satisfy the safety property ϕ, then we proceed to analyze the witness run π. The witness analysis procedure is outlined in Section 3.1. The analysis will return a set `C` of fence constraints such that any fence set which is restrictive enough to prevent the erroneous run π must contain at least one fence constraint from `C`. Since every sound fence set must prevent π, this means that every sound fence set must have a non-empty intersection with `C`. Therefore we add `C` to `req` on line 12, so that `req` will better guide our choice of fence set candidates in the future.

Note that in the beginning, **hits**(`req`, κ) will return a singleton set of the empty set, namely $\{\varnothing\}$. Then `F` is chosen as the empty set $\varnothing$ and the algorithm continues. A special case occurs when the run π contains no memory access reorderings. This means that $\mathcal{P}$ can reach the bad states even under the SC memory model. Hence it is impossible to correct $\mathcal{P}$ by only inserting fences. The call **analyze_witness**($\mathcal{P} \oplus \texttt{F}, \pi$) will in this case return the empty set. The main algorithm then terminates, also returning the empty set, indicating that there are no optimal fence sets for the given problem.

3.1 Witness Analysis

The **analyze_witness** function takes as input a program $\mathcal{P}$ (which may already contain some fences inserted by the fence insertion algorithm), and a counter-example run π generated by the reachability analysis. The goal is to find a set G of fences such that i) all sound fence sets have at least one fence in common with G and ii) G contains no fence which is already in $\mathcal{P}$. It is desirable to keep G as small as possible, in order to quickly converge on sound fence sets.

There are several ways to implement **analyze_witness** to satisfy the above requirements. One simple way builds on the following insight: Any sound fence set must prevent the current witness run. The only way to do that, is to have fences preventing some access reordering that occurs in the witness. So a set G which contains all fences preventing some reordering in the current witness satisfies both requirements listed above.

Program fragment	Witness run
	...
	1.fetch(P0,x)
process P0	2.L0: x := 1
...	3.fetch(P0,y)
L0: x := 1;	4.L1: $\$r_0$:= y
L1: $\$r_0$:= y;	5.fetch(P0,z)
L2: $\$r_1$:= z;	6.L2: $\$r_1$:= z
...	...
	7.wrllc(P0,x)
	...

Fig. 4: Left: Part of a program $\mathcal{P}$, containing three instructions of the thread P0. Right: A part of a counter-example run π of $\mathcal{P}$.

As an example, consider Figure 4. On the left, we show part of a program $\mathcal{P}$ where the thread P0 performs three memory accesses L0, L1 and L2. On the right, we show the corresponding part of a counter-example run π. We see that the store L0 becomes globally visible at line 7, while the loads L1 and L2 access the LLC at respectively lines 3 and 5. Hence the order between the instructions L0 and L1 and the order between L0 and L2 in the program code, is opposite to the order in which they take effect w.r.t. the LLC in π. We say that L0 is *reordered* with L1 and L2. The loads are not reordered with each other. Let us assume that π does not contain any other memory access reordering. The reordering is caused by the late wrllc on line 7. Hence, this particular error run can be prevented by the following four fence constraints: $c_0 = (\texttt{L0}, \texttt{ssfence})$, $c_1 = (\texttt{L1}, \texttt{ssfence})$, $c_2 = (\texttt{L0}, \texttt{fence})$, and $c_3 = (\texttt{L1}, \texttt{fence})$. The fence set returned by **analyze_witness**$(\mathcal{P}, \pi)$ is $G = \{c_0, c_1, c_2, c_3\}$. Notice that G satisfies both of the requirements for **analyze_witness**.

4 Experimental Results

We have implemented our fence insertion algorithm together with a reachability analysis for SiSd in the tool Memorax. It is publicly available at https://github.com/memorax/memorax. We apply the tool to a number of benchmarks (Section 4.1). Using simulation, we show the positive impact of

using different types of fences, compared to using only the full fence, on performance and network traffic (Section 4.2).

4.1 Fence Insertion Results

We evaluate the automatic fence insertion procedure by running our tool on a number of different benchmarks containing racy code. For each example, the tool gives us all optimal sets of fences. We run our tool on the same benchmarks both for SiSd and for the Si protocol.[4] The results for SiSd are given in Table 1. We give the benchmark sizes in lines of code. All benchmarks have 2 or 3 processes. The fence insertion procedure was run single-threadedly on a 3.07 GHz Intel i7 CPU with 6 GB RAM.

The first set of benchmarks are classical examples from the context of lock-free synchronization. They contain mutual exclusion algorithms: a simple CAS lock –*cas*–, a test & TAS lock –*tatas*– [29], Lamport's bakery algorithm –*bakery*– [17], the MCS queue lock –*mcsqueue*– [22], the CLH queue lock –clh– [20], and Dekker's algorithm –*dekker*– [10]. They also contain a work scheduling algorithm –*postgresql*–[5], and an idiom for double-checked locking –*dclocking*– [28], as well as two process barriers –*srbarrier*– [29] and –*treebarrier*– [22]. The second set of benchmarks are based on the Splash-2 benchmark suite [33]. We use the race detection tool Fast&Furious [26] to detect racy parts in the Splash-2 code. We then manually extract models capturing the core of those parts.

In four cases the tool detects bugs in the original Splash-2 code. The *barnes* benchmark is an n-body simulation, where the bodies are kept in a shared tree structure. We detect two bugs under SiSd: When bodies are inserted (*barnes 2*), some bodies may be lost. When the center of mass is computed for each node (*barnes 1*), some nodes may neglect entirely the weight of some of their children. Our tool inserts fences that prevent these bugs. The *radiosity* model describes a work-stealing queue that appears in the Splash-2 *radiosity* benchmark. Our tool detects that it is possible for all workers but one to terminate prematurely, leaving one worker to do all remaining work. The *volrend* model is also a work-stealing queue. Our tool detects that it is possible for some tasks to be performed twice. The bugs in *radiosity* and *volrend* can occur even under SC. Hence the code cannot be fixed only by adding fences. Instead we manually correct it.

For each benchmark, we apply the fence insertion procedure in two different modes. In the first one ("Only full fence"), we use only full fences. In the table, we give the total time for computing all optimal sets, the number of such sets, and the number of fences to insert into each process. For treebarrier, one process (the root process) requires only one fence, while the others require two. Notice also that if a benchmark has one solution with zero fence, that means that the benchmark is correct without the need to insert any fences.

In the second set of experiments ("Mixed fences"), we allow all four types of fences, using a cost function assigning a cost of ten units for a full fence, five units

[4] Our methods could also run under a plain Sd protocol. However, to our knowledge, no cache coherence protocol employs only Sd without Si.

[5] http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php

for an `ssfence` or an `llfence`, and one unit for a synchronized write. These cost assignments are reasonable in light of our empirical evaluation of synchronization cost in Section 4.2. We list the number of inserted fences of each kind. In `barnes 1`, the processes in the model run different codes. One process requires an `llfence`, the other an `ssfence`.

In addition to running our tool for SiSd, we have also run the same benchmarks for Si. As expected, `ssfence` and `syncwr` are no longer necessary, and `fence` may be downgraded to `llfence`. Otherwise, the inferred fence sets are the same as for SiSd. Since Si allows fewer behaviors than SiSd, the inference for Si is mostly faster. Each benchmark is fenced under Si within 71 seconds.

		Only full fence			Mixed fences		
Benchmark	Size	Time	#solutions	#fences	Time	#solutions	Fences / proc
bakery	45 LOC	17.3 s	4	5	108.1 s	16	2xsw,4xll,1xss
cas	32 LOC	<0.1 s	1	2	<0.1 s	1	1xll,1xss
clh	37 LOC	4.4 s	4	4	3.7 s	1	3xsw,2xll,1xss
dekker	48 LOC	2.0 s	16	3	2.9 s	16	1xsw,2xll,1xss
mcslock	67 LOC	15.6 s	4	2	33.0 s	4	1xll,1xss
testtas	38 LOC	<0.1 s	1	2	<0.1 s	1	1xll,1xss
srbarrier	60 LOC	0.3 s	9	3	0.4 s	4	2xll,1xss
treebarrier	56 LOC	33.2 s	12	1 / 2	769.9 s	132	1xll,1xss
dclocking	44 LOC	0.8 s	16	4	0.9 s	16	1xsw,2xll,1xss
postgresql	32 LOC	<0.1 s	4	2	0.1 s	4	1xll,1xss
barnes 1	30 LOC	0.2 s	1	1	0.5 s	1	1xll / 1xss
barnes 2	96 LOC	16.3 s	16	1	16.1 s	16	1xss
cholesky	98 LOC	1.6 s	1	0	1.6 s	1	0
radiosity	196 LOC	25.1 s	1	0	24.6 s	1	0
raytrace	101 LOC	69.3 s	1	0	70.1 s	1	0
volrend	87 LOC	376.2 s	1	0	376.9 s	1	0

Table 1: Automatic fence insertion for SiSd.

4.2 Simulation Results

Here we show the impact of different choices of fences when executing programs. In particular we show that an optimal fence set w.r.t. the "Mixed fences" cost function yields a better performance and network traffic compared to an optimal fence set using the "Only full fence" cost function. We evaluate the micro-benchmarks analyzed in the previous section and the Splash-2 benchmarks suite [33]. All programs are fenced according to the optimal fence sets produced by our tool as described above.

Simulation Environment: We use the Wisconsin GEMS simulator [21]. We model an in-order processor that with the Ruby cycle-accurate memory simulator (provided by GEMS) offers a detailed timing model. The simulated system is a 64-

core chip multiprocessor with a SiSd architecture and 32KB, 4-way private L1 caches and a logically shared but physically distributed L2, with 64 banks of 256KB, 16-way each.

The DoI State: When an `llfence` is executed, eviction of all clean data in the L1 cache is forced. This should take a single cycle. However, when a cache line contains multiple words, with a per-word dirty bit, it may contain both dirty and clean words. To evict the clean words, we would have to write the dirty data to the LLC and evict the whole line. That would harm performance and enforce a stronger access ordering than is intended by an `llfence`. For this reason, when we implemented the SiSd protocol in GEMS, we introduced a new L1 cache state: DoI (Dirty or Invalid). A cache line in this state contains words that are either dirty or invalid, as indicated by the dirty bit. This allows an efficient, one-cycle implementation of `llfence`, where cache lines in a mixed state transition to DoI, thus invalidating precisely the clean words. It also allows the `llfence` not to cause any downgrade of dirty blocks, thus improving its performance.

Cost of Fences: Our tool employs different weights in order to insert fences. Here, we calculate the weights based on an approximate cost of fences obtained by our simulations. The effect of fences on performance is twofold. First, there is a cost to execute the fence instructions (fence latency); the more fences and the more dirty blocks to self-downgrade the higher the penalty. Second, fences affect cache miss ratio (due to self-invalidation) and network traffic (due to extra fetches caused by self-invalidations and write-throughs caused by self-downgrades). The combined effect on cache misses and network traffic also affects performance. We calculate the cost of fences in time as follows: $time_{fence} = lat_{fence} + misses_{si} * lat_{miss}$ where lat_{fence} is the latency of the fence, $misses_{si}$ is the number of misses caused by self-invalidation, and lat_{miss} is the average latency of such misses. According to this equation, the average cost in time of each type of fence when running the Splash2 benchmarks, normalized with respect to a full fence is the following: the cost of an `llfence` is 0.68, the cost of an `ssfence` is 0.23, and the cost of a `syncwr` is 0.14. The cost of the fences in traffic is calculated as $traffic_{fence} = sd * traffic_{wt} + misses_{si} * traffic_{miss}$ where sd is the number of self-downgrades, $traffic_{wt}$ is the traffic caused by a write-through, and $traffic_{miss}$ is the traffic caused by a cache miss. Normalized to a full fence, the cost in traffic is 0.43 for an `llfence`, 0.51 for an `ssfence`, and 0.10 for a `syncwr`. Thus, the weights assigned to fences in our tool seem reasonable.

Execution Time: Figure 5 (top) shows simulated execution time for both the micro-benchmarks (top) and the Splash2 benchmarks (bottom). The use of mixed fences improves the execution time compared to using full fences by 10.4% for the micro-benchmarks and by 1.0% for the Splash2 benchmarks. The DoI-mixed column shows the execution time results for the same mixed fence sets as the mixed column. But in DoI case, `llfence`s are implemented in GEMS using an extra L1 cache line state (the Dirty-or-Invalid state). This feature is an architectural optimization of the SiSd protocol. Implementing the DoI state

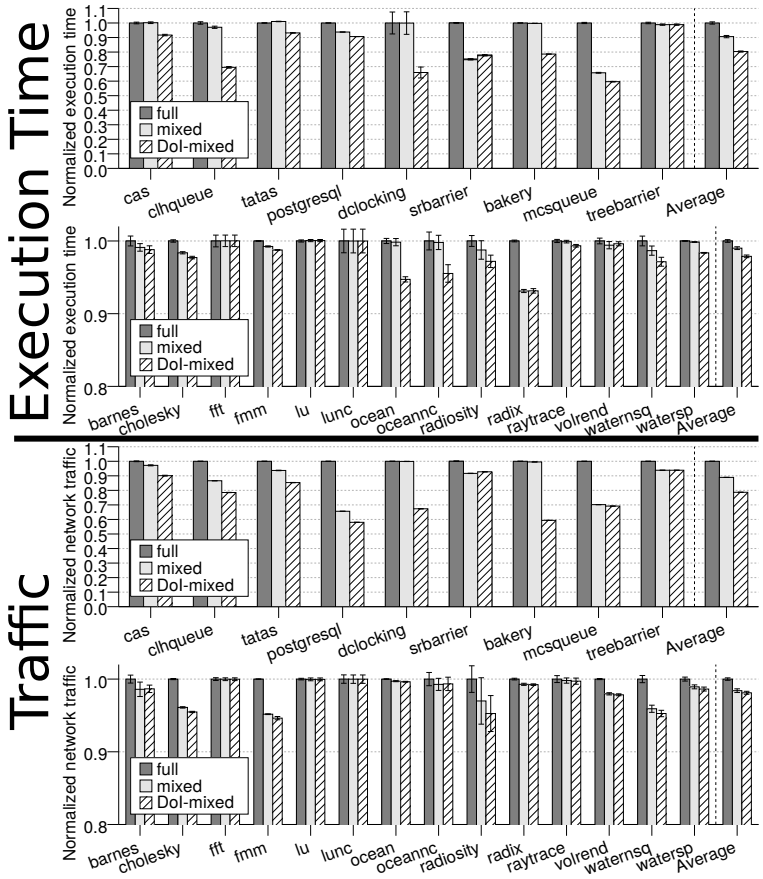

Fig. 5: Execution time and network traffic under different fence sets.

further improves the performance of the mixed fences, by 20.0% for the micro-benchmarks and 2.1% for the Splash2, on average, compared to using of full fences. Mixed fences are useful for applications with more synchronization.

Traffic: Figure 5 (bottom) shows the traffic in the on-chip network generated by these applications. The use of `llfence`, `ssfence`, `syncwr` is able to reduce the traffic requirements by 11.1% for the micro-benchmarks and 1.6% for the Splash2 applications, on average, compared to using full fences. Additionally, when employing the DoI state, this reduction reaches 21.3% and 1.9%, on average, for the micro-benchmarks and the Splash2, respectively. Again, the more synchronization is required by the applications, the more traffic can be saved by employing mixed fences.

5 Conclusions and Future Work

We have presented a uniform framework for automatic fence insertion in programs that run on architectures that provide self-invalidation and self-downgrade. We have implemented a tool and applied it on a wide range of benchmarks. There are several interesting directions for future work. One is to instantiate our framework in the context of abstract interpretation and stateless model checking. While this will compromise the optimality criterion, it will allow more scalability and application to real program code. Another direction is to consider *robustness* properties [5]. In our framework this would mean that we consider program traces (in the sense of Shasha and Snir [30]), and show that the program will not exhibit more behaviors under SiSd than under SC. While this may cause over-fencing, it frees the user from providing correctness specifications such as safety properties. Also, the optimality of fence insertion can be evaluated with the number of the times that each fence is executed. This measurement will provide more accuracy when, for instance, fences with different weighs are inserted in a loop computation in a branching program.

Acknowledgment This work was supported by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Swedish Board of Science project, "Rethinking the Memory System", the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under the project "Jóvenes Líderes en Investigación" and European Commission FEDER funds.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS, pp. 204–219 (2012)
2. Adve, S.V., Hill, M.D.: Weak ordering – a new definition. In: ISCA. pp. 2–14 (1990)
3. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence - A static analysis approach to automatic fence insertion. In: CAV. pp. 508–524 (2014)
4. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Programming Languages and Systems, pp. 533–553. Springer (2013)
5. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP (2). LNCS, vol. 6756, pp. 428–440. Springer (2011)
6. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA. pp. 21–28 (2005)
7. Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S.V., Adve, V.S., Carter, N.P., Chou, C.T.: DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In: PACT. pp. 155–166 (2011)
8. Davari, M., Ros, A., Hagersten, E., Kaxiras, S.: An efficient, self-contained, on-chip, directory: DIR_1-SISD. In: PACT. pp. 317–330 (2015)
9. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: DISC. Lecture Notes in Computer Science, vol. 4167, pp. 194–208 (2006)
10. Dijkstra, E.W.: Cooperating sequential processes (2002)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)

12. Hower, D.R., Hechtman, B.A., Beckmann, B.M., Gaster, B.R., Hill, M.D., Reinhardt, S.K., Wood, D.A.: Heterogeneous-race-free memory models. In: ASPLOS. pp. 427–440 (2014)
13. Kaxiras, S., Keramidas, G.: SARC coherence: Scaling directory cache coherence in performance and power. IEEE Micro 30(5), 54–65 (Sep 2011)
14. Kaxiras, S., Ros, A.: A new perspective for efficient virtual-cache coherence. In: ISCA. pp. 535–547 (2013)
15. Koukos, K., Ros, A., Hagersten, E., Kaxiras, S.: Building heterogeneous unified virtual memories (uvms) without the overhead. ACM TACO 13(1), 1:1–1:22 (2016)
16. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD. pp. 111–119. IEEE (2010)
17. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Communications of the ACM 17 (Aug 1974)
18. Lebeck, A.R., Wood, D.A.: Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In: ISCA. pp. 48–59 (1995)
19. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440 (2012)
20. Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Parallel Processing Symposium, 1994. Proceedings., Eighth International. pp. 165–171. IEEE (1994)
21. Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News 33(4), 92–99 (Sep 2005)
22. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) 9 (Feb 1991)
23. Ros, A., Davari, M., Kaxiras, S.: Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies. In: HPCA. pp. 186–197 (2015)
24. Ros, A., Kaxiras, S.: Complexity-effective multicore coherence. In: PACT. pp. 241–252 (2012)
25. Ros, A., Kaxiras, S.: Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In: ISCA. pp. 427–438 (2015)
26. Ros, A., Kaxiras, S.: Fast&furious: A tool for detecting covert racing. In: PARMA and DITAM. pp. 1–6 (2015)
27. Sakalis, C., Leonardsson, C., Kaxiras, S., Ros, A.: Splash-3: A properly synchronized benchmark suite for contemporary research. In: ISPASS (2016)
28. Schmidt, D.C., Harrison, T.: Double-checked locking - an optimization pattern for efficiently initializing and accessing thread-safe objects. In: PLoP (1996)
29. Scott, M.L.: Shared-Memory Synchronization. Morgan & Claypool (2013)
30. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems (TOPLAS) 10(2), 282–312 (1988)
31. Sung, H., Adve, S.V.: DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations. In: ASPLOS. pp. 545–559 (2015)
32. Sung, H., Komuravelli, R., Adve, S.V.: DeNovoND: Efficient hardware support for disciplined non-determinism. In: ASPLOS. pp. 13–26 (2013)
33. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: ISCA. pp. 24–36 (1995)