MENDING FENCES WITH SELF-INVALIDATION AND SELF-DOWNGRADE [*]

PAROSH AZIZ ABDULLA [a], MOHAMED FAOUZI ATIG [a], STEFANOS KAXIRAS [a],
CARL LEONARDSSON [a], ALBERTO ROS [b], AND YUNYUN ZHU [a]

[a] Uppsala University, Sweden
  e-mail address: parosh.abdulla@it.uu.se
  e-mail address: mohamed_faouzi.atig@it.uu.se
  e-mail address: stefanos.kaxiras@it.uu.se
  e-mail address: carl.leonardsson@it.uu.se
  e-mail address: yunyun.zhu@it.uu.se

[b] Universidad de Murcia, Spain
  e-mail address: aros@ditec.um.es

ABSTRACT. Cache coherence protocols based on 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 programmers. We propose a novel formal model that captures the semantics of programs running under such protocols, and features a set of fences that interact with the coherence layer. Using the model, we design an algorithm to analyze the reachability and check whether a program satisfies a given safety property with the current set of fences. 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. Traditional cache coherence protocols, either directory-based or snooping-based, are *transparent* to the programmer in the sense that they respect the memory consistency model of the system, and hence 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 [LW95, CKS+11, KK11, RK12, SKA13, KR13, HHB+14, RDK15, SA15, RK15a, DRHK15, KRHK16, RK16]. Principal techniques among these proposals are Self-Invalidation (Si) and Self-Downgrade (Sd).

A protocol with Self-Invalidation (Si) allows old copies of the data to be kept, without invalidation on each store operation by another core. This eliminates the need for tracking readers [LW95]. In an Si protocol, invalidation of data from a cache is caused by synchronization instructions executed by the core local to the cache.

Correspondingly, in a protocol with Self-Downgrade (Sd), downgrades are not caused by read operations in other cores, but again by synchronization instructions. Sd eliminates the need to track the last writer of a cache line [RK12].

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 [RK12]. But this comes at a price: SiSd protocols induce *weak memory semantics* that allow reordering or memory instructions. The behavior of a program may now deviate from its behavior under the standard *Sequentially Consistent (SC)* semantics [Lam79], sometimes 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 to eliminate the undesired behaviors. A fence instruction, executed by a thread, 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. Each type eliminates a different kind of non-SC behavior, and may have different impact on the program performance. In fact, unnecessary fences may significantly jeopardize program performance. This is particularly true for the fences considered in this work, 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. There are multiple ways of defining optimality. The one we adopt is to calculate the number of occurrences of fences in the source program, with the costs of different fences taken into account.

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 [CKS+11, KK11]. 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 [CL05]), transactional memories (e.g., the TL2 algorithm [DSS06]), and synchronization library primitives (e.g. `pthread_spin_lock` in `glibc`). In this paper, we consider a different approach where fences are inserted to restore 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 [HS08]. 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. We define the cost to be the number of occurrences of fences in the source program, with the costs of different fences taken into account. In general, there may exist several different optimal sets of fences for the same program. Our experiments (Section 7) 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 [LW95, CKS+11, KK11, RK12, SKA13, KR13, HHB+14, RDK15, SA15, RK15a, DRHK15, KRHK16]. 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:

- We define 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.
- We develop 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 [WOT+95], which have been fixed in a recent Splash-3 release [SLKR16]. 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 [LNP+12] (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 [AH90]. Dynamic self-invalidation (for DRF programs) was first proposed by Lebeck and Wood [LW95]. Several recent works employ self-invalidation to simplify coherence, including SARC coherence [KK11], DeNovo [CKS+11, SKA13, SA15], and VIPS-M [RK12, KR13, RDK15, RK15a, KRHK16].

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 [AAC+12], 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 [KVY10] 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 [BDM13], the tool Trencher is introduced, which inserts fences under TSO to enforce robustness (formalised by Shasha and Snir in [SS88]), also using an exact, model-checking based technique. Musketeer [AKNP14] 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 [BDM13] and [AKNP14] first enumerate all solutions and then use linear programming to find the optimal set of fences.

In [GHS11] and [GHS09], the authors discuss fence insertion and verification of satisfiability modulo theory (SMT) of relaxed memory models under transactional memories. Our work, in comparison, addresses the reachability problem.

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, ours 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. Self-Invalidation, Self-Downgrade, and their Fences

In this section, we recall the notions of self-invalidation and self-downgrade, and describe the main features of the system architecture and the protocol we consider. We also introduce two fences that are defined under the protocol.

2.1. **Self-Invalidation and Self-Downgrade.** Self-invalidation eliminates the need to track sharers of a cache line in a directory structure [LW95]. We consider that invalidation of shared data in caches is caused by fences inserted in the programs and not as a result of writes from other cores.

Correspondingly, self-downgrade eliminates the need to track the last writer (i.e., the owner, in a Moesi-like protocol) of a cache line [RK12]. This is because downgrades are also not performed as a consequence of read operations, but by means of fence instructions inserted in the programs.

A protocol that implements self-invalidation together with self-downgrade does not need a directory, thus removing one of the main sources of complexity and scalability constraints in traditional cache coherence protocols [RK12].

We first set the stage for the architecture and the coherence protocol we study in this work, by discussing some of their details: i.e., how memory accesses are resolved, and how the self-invalidation and the self-downgrade are performed when a fence is encountered.

System architecture. We assume a standard multicore architecture with a number of cores, each with a private L1 cache. The proposals and algorithms described in this paper are more widely applicable to systems with several levels of private caches. The last level cache (LLC) of the system is logically shared among all the cores.

2.2. **Cache coherence protocol.** We also assume a very simple version of a self-invalidation/self-downgrade protocol with only three stable states in the L1 cache (invalid –I–, clean –C–, and dirty –D–) and only two stable states in the LLC (invalid –I– and valid –V–). There are no invalidations or downgrades, which means that there are no transient states to account for the arrival of such coherence actions. There are no requests other than from the L1s to the LLC (and from LLC to memory). There is no distinction of data into private or shared as in [RK12], as this would distract from our discussion. Such optimizations are straightforward extensions in our approach.

Basic actions: To connect with the formal specification of the system behavior that follows in Section 4 we present here some necessary —if somewhat mundane— details of the basic actions in our assumed system.

- A *read* request that misses in the L1 cache issues a request to the LLC. If it hits in the LLC, a reply containing the data is sent. In case of a miss, main memory is accessed to get the data block. When the data arrives to the L1 cache, the miss is resolved and the data can be accessed. The block is stored in an L1 cache line in clean state.
- A write request is always resolved immediately, even if the block is not present in the L1 (in this case, the miss status handling register –MSHR– can temporarily hold the new data). This is because writes are assumed to be data-race-free, i.e, they are always ordered with respect to conflicting reads [AH90]. In this case, writes do not require "write permission".
- After writing in an L1 cache line (e.g., one word), if the data block is missing it is fetched from the LLC. The block is merged with the written word. Before merging the data, the cache line is in a transient state and once merged transitions to dirty.
- An atomic read-modify-write (RMW) request (e.g., test-and-set –TAS–) needs to reach the LLC, get the data block and send it back to the L1 cache. During this operation, the corresponding cache line in the LLC is blocked, so no other RMW request can proceed. When the data arrives at the L1 cache it is read and possibly modified. If modified, the data is written-back in the LLC, unblocking the LLC line at the same time. Otherwise the LLC line will be simply unblocked. This blocking operation —common in other protocols for directory operations that generate new messages (indirection, invalidations, etc.)— is only necessary in this protocol for RMW requests. Once the transaction finishes, the data block remains in the L1 in clean state.
- Evictions of clean cache lines only require a change of state to invalid. However, evictions of dirty cache lines need to write back to the LLC the data that have been modified locally. This is necessary to avoid overwriting unrelated data in the LLC cache line (a different part of the LLC line may have been modified independently without a conflict). When modified data are written-back an acknowledgement message is sent to the L1 to signal the completion of the corresponding writes.
- To keep track of the locally modified data in an L1 cache line, it is necessary to keep information in the form of a dirty bit per word (byte), either with the L1 cache lines [CKS+11], or in the write-buffer or MSHRs [RK12].

2.3. **Self-Invalidation and Self-Downgrade fences.** Since the described protocol has neither invalidation on writes nor downgrades on reads, we need to ensure that a read operation sees the latest value written, when this is intended by the program. Typically, the program contains synchronization to enforce an order between conflicting writes and reads.

In this context, to ensure that a read gets the latest value of a corresponding write, two things need to happen: first, the data in the writer's cache must be self-downgraded and put back in the LLC sometime after the write but before the read; second, if there is a (stale) copy in the reader's cache, it must be self-invalidated sometime before the read. The self-downgrade and self-invalidation also need to be ordered the same as the write and read are ordered by synchronization.

Prior proposals [LW95, KK11, CKS+11, SKA13, ADC11, RK12, KR13] invariably offer SC for data-race-free (DRF) programs [AH90]. In general, such proposals can be thought of

as employing a single fence causing the self-invalidation and self-downgrade of cached data, *on every synchronization in the DRF program* (e.g., [RK12].)

Our approach is fundamentally different. We make no assumption as to what constitutes synchronization (perhaps ordinary accesses relying on SC semantics, or algorithms involving atomic RMW operations). We insert *fences* in a program to cause self invalidation and self downgrade in such a way as to produce the desired behavior.

With only a single fence, ensuring that a read sees the latest value of a write causes the self-invalidation *and* self-downgrade of *both* the reader's and the writer's cache. In many cases, this is unnecessary.

One of the contributions of our work is to propose two separate fences, which we call "load-load fence" (`llfence`) and "store-store fence" (`ssfence`) to address the above problem. An `llfence` self-invalidates only *clean* data in the cache (at word level), while an `ssfence` writes back only *dirty* data to the LLC (again at word level), so all the data in the L1 cache of the process are clean.
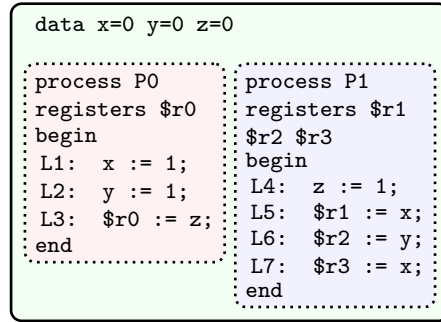
The separation of self-invalidation and self-downgrade into two fences affects performance in two ways: first, we reduce the fence latency when we do not have to self-downgrade; second, we eliminate extraneous misses (that cost in performance) when we only need to self-downgrade.

2.4. **Improving self-invalidation of partially dirty cache lines: the DoI state.** The `llfence` defined above operates efficiently for cache lines that are entirely clean, or entirely dirty. In particular, for clean data they take a single cycle, while for dirty data they do not perform any action. However, cache lines that contain both clean and dirty words are not self-invalidated efficiently. Consider, for example, a cache line with one clean word and one dirty word (its dirty bit is set). The `llfence` must invalidate the clean word (if it were to be accessed afterwards), without affecting the dirty word. If we invalidate the whole cache line we also have to write the dirty data to the LLC. This would have the same impact (for this cache line) as a single full fence, and would offer no advantage from using the `llfence` instead.

In order to improve the efficiency of `llfence` operations we propose that they operate at word granularity, being able to self-invalidate the clean words in a single cycle and leaving untouched the dirty words. Thus, we introduce a new state for L1 cache lines, called *DoI* (dirty or invalid), for exactly this purpose. A cache line in this state contains words that are either dirty (with the dirty bit set) or invalid (with the dirt bit unset). An `llfence` transitions any partially dirty cache line to the *DoI* state. No write back is performed for its dirty words. This allows an efficient, one-cycle implementation of `llfence`, since now the only necessary action for a `llfence` is to change the cache-line state from dirty to *DoI*.

3. OVERVIEW

In this section, we give an informal overview of the main concepts in our framework. We describe the semantics (configurations and runs) of programs running under SiSd, the notion of safety properties, the weak memory model induced by SiSd, the roles of fences, and optimal sets of fences. This will be formalized in later sections.

```
data x=0 y=0 z=0

process P0        process P1
registers $r0     registers $r1
begin             $r2 $r3
L1:   x := 1;     begin
L2:   y := 1;     L4:   z := 1;
L3:   $r0 := z;   L5:   $r1 := x;
end               L6:   $r2 := y;
                  L7:   $r3 := x;
                  end
```

FIGURE 1. A simple program $\mathcal{P}$.

Example. We will use the toy program $\mathcal{P}$ in Figure 1 as a running example. The program is written in a simple assembly-like programming language. The syntax and semantics of the language are formally defined in Section 4. $\mathcal{P}$ consists of two processes P0 and P1 that share three variables x, y, and z. Process P0 has one register $r0, and process P1 has three registers $r1, $r2, and $r3. Process P0 has three instructions labeled with L1, L2, L3, and process P1 has four instructions labeled with L4, L5, L6, L7.

To simplify the presentation, we assume that each cache line holds only one variable. We also assume that the underlying protocol contains both SI and SD. It is straightforward to extend our framework to the case where a cache line may hold several variables, and to the case where the protocol only contains one of SI and SD. In $\mathcal{P}$, all the instructions have unique labels. Therefore, to simplify the presentation, we identify each label with the corresponding instruction, e.g., L1 and x := 1 in P0.
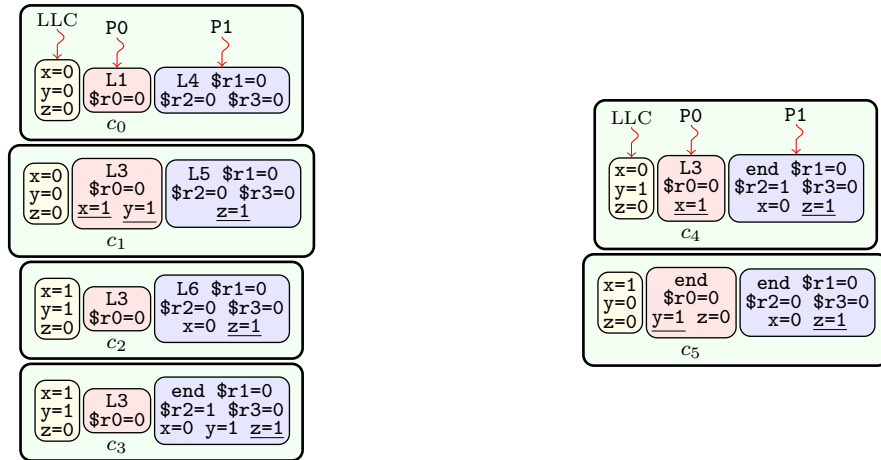
FIGURE 2. Configurations

Configurations. A *configuration* is a snapshot of the global state of the system, and consists of two parts, namely the *local* and *shared* parts. The local part defines the local states of the processes, i.e., it defines for each process: (i) its next instruction to be executed, (ii) the values stored in its registers, and (iii) the variables (memory locations) that are currently cached in its L1, together with their status: *invalid*, *clean*, or *dirty*; and the current value of

the variable in case it is valid. The shared part defines, for each variable, its value in the LLC. Figure 2 shows different configurations of $\mathcal{P}$. Each configuration is depicted as three fields, representing the LLC, P0, and P1 respectively. $\mathcal{P}$ starts its execution from the *initial configuration* c_0, where the values of all variables are 0 in the LLC. P0 and P1 are about to execute the instructions labeled L1 and L4, respectively. The values of all registers are 0. None of the variables is valid in the L1 of the processes. In contrast, in c_4, the value of y is 1 in the LLC. P0 is about to execute the instruction L3, while P1 has ended its execution. The value of the register $r2 is 1. The variables x, and z are valid in the L1 of P1, with values 0 and 1 respectively. The variable z is dirty in the L1 of P1 (marked by underlining z=1), while x is clean (not underlined). Finally, there is a dirty copy of x with value 1 in P0.

Safety Properties. Suppose that, together with the program $\mathcal{P}$, we are given a *safety property* ϕ which states that a certain set *Bad* of configurations will not occur during any execution of $\mathcal{P}$. For the example, we assume that *Bad* is the set of configurations where (i) P1 has ended its execution, and (ii) the registers $r2 and $r3 have values 1 and 0 respectively. For instance, c_3 and c_4 are members of *Bad*. We are interested in checking whether $\mathcal{P}$ satisfies ϕ under the SISD semantics. Note that the set *Bad* is not reachable in $\mathcal{P}$ under SC semantics, which also means that $\mathcal{P}$ satisfies ϕ under SC.

Runs. The semantics of a program boils down to defining a *transition relation* on the set of configurations. The execution of the program can be viewed as a *run*, consisting of a sequence of *transitions*, i.e., events that take the program from one configuration to another by changing the local states of the processes and the shared parts. Such a transition will either be performed by a given process when it executes an instruction, or it occurs due to a system event. We consider three kinds of system events: fetch, evict, and wrllc. They model respectively fetching a value from LLC to L1, invalidating an L1 entry, and writing a dirty L1 entry through to the LLC. The system events are decoupled from program instructions and execute independently.
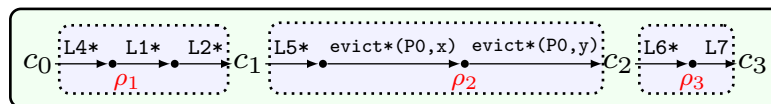

FIGURE 3. The run π_1.

In Figure 3 we show one example run π_1 of $\mathcal{P}$. It consists of three sequences ρ_1, ρ_2, ρ_3 of transitions, and takes us from c_0 through c_1 and c_2 to c_3. ρ_1: Starting from c_0, P1 executes L4. Since z is invalid in the L1 of P1, it is fetched from the LLC. In Figure 3, the star in L4* is to simplify the notation, and it indicates that the instruction L4 is preceded by fetch event[1] of the process (here P1) on the relevant variable (here z). Consequently, a dirty copy of z with value 1 is stored in the L1 of P1. Next, P0 executes L1 and L2, putting dirty copies (with values 1) of x and y in its L1, reaching the configuration c_1. ρ_2: P1 executes L5, fetching x from the LLC, and storing a clean copy with value 0. P0 evicts the variable x. An evict event may only be performed on clean variables. To simplify the notation, we augment the evict event in Figure 3 by a star. This indicates that it is preceded by an

---

[1]In the examples of this section, fetch events always precede read or write events. In general, fetch events may occur anywhere along the run.

wrllc event on x. The latter updates the value of x to 1 in the LLC, and makes x clean in the L1 of P0. Next, P0 evicts y in a similar manner, thus reaching c_2. ρ_3: P1 executes L6. Since y is invalid in the L1 of P1, it is fetched from the LLC, and stored with value 1 as clean. The register $r2 will be assigned the value 1. Finally, P1 executes L7. Since x is valid, it need not be fetched from the memory (L7 is therefore not starred in Figure 3), and hence $r3 is assigned the value 0. Thus we reach c_3, which is in the set *Bad*. And so $\mathcal{P}$ violates the safety property ϕ under the SISD semantics.

Weak Memory Model. Although the configuration c_3 is not reachable from c_0 under SC semantics, we demonstrated above that it is reachable under SISD semantics. The reason is that SISD introduces a weak memory semantics in the form of *reorderings* of events. In the case of π_1, we have a *read-read* reordering. More precisely, the read event L7 overtakes the read event L6, in the sense that L6 is issued before L7, but the value assigned to $r2 in L6 (coming from the write on y in L2) is more recent than the value assigned to $r3 in L7 (which is the initial value of x). To prevent event reorderings, we use *fences*. In this paper, we use four types of fences, namely llfence, ssfence, fence, and syncwr. In this section, we only describe the first three types.


```
data x=0 y=0 z=0

process P0        process P1
registers $r0     registers $r1
begin             $r2 $r3
L1:   x := 1;     begin
L2:   y := 1;     L4:   z := 1;
L3:   $r0 := z;   L5:   $r1 := x;
end               L6:   $r2 := y;
                  L8:   llfence;
                  L7:   $r3 := x;
                  end
```

FIGURE 4. The program $\mathcal{P}_1$.
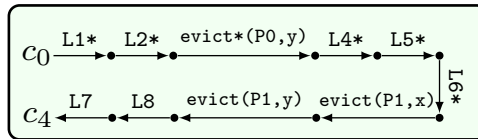
FIGURE 5. The run π_2.

LL Fences. To forbid the run π_1, we insert an llfence between L6 and L7, obtaining a new program $\mathcal{P}_1$ (Figure 4). Intuitively, an llfence (load-load-fence) blocks when there are clean entries in the L1 of the process, and hence forbids the reordering of two read (load) operations. For instance, in the above example, L8 cannot be executed before x has become invalid in the L1 of P1, and hence the new value 1 of x will be assigned to $r3 in L7. Therefore, $\mathcal{P}_1$ does not contain the run π_1 any more.

```
data x=0 y=0 z=0

process P0        process P1
registers $r0     registers $r1
begin             $r2 $r3
L1:   x := 1;     begin
L9:   ssfence;    L4:   z := 1;
L2:   y := 1;     L5:   $r1 := x;
L3:   $r0 := z;   L6:   $r2 := y;
end               L7:   $r3 := x;
                  end
```
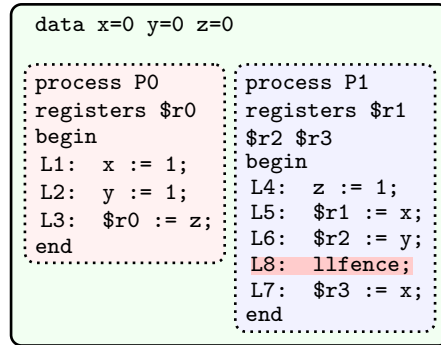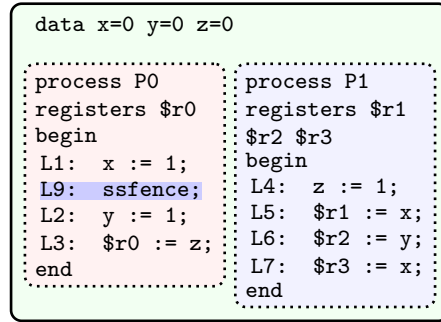
FIGURE 6. The program $\mathcal{P}_2$.
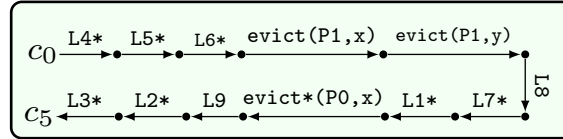
FIGURE 7. The run π_3.

SS Fences. Despite the fact that the insertion of L8 eliminates the run π_1, the program $\mathcal{P}_1$ still does not satisfy the safety property ϕ: the set Bad is still reachable from c_0 in $\mathcal{P}_1$, this time through a run π_2 (Figure 5) that leads to the configuration c_4 (which is also a member of Bad). In π_2, P0 performs L1 and L2 and then evicts only y, which means that the values of x and y in the LLC will be 0 resp. 1. Now, P1 will perform the instructions L4, L5, L6. Next P1 evicts x and then y which means that now P1 does not contain any clean variables, and hence L8 is enabled. Notice that these evict events are not followed by stars (since they concern clean copies of the variables). Finally, P1 executes L7. Since x is invalid in the L1 of P0, it is fetched from the LLC (where its value is 0 since it was never evicted by P0), and hence $r3 will be assigned the value 0. Thus, we are now in c_4.

Notice again that π_2 is not possible under SC (in the SC semantics, fences have no effect, so they are equivalent to empty statements). The reason why π_2 is possible under SISD is due to a *write-write* reordering. More precisely, the write event L1 is issued before the write event L2, but L2 takes effect (updates the LLC) before L1. To forbid the run π_2, we insert an ssfence between L1 and L2, obtaining the program $\mathcal{P}_2$ (Figure 6). An ssfence (store-store-fence) is only enabled when there are no dirty entries in the L1 of the process. Hence it forbids the reordering of two write operations. For instance, in the above example, L9 cannot be executed before x has been evicted by P0, and hence the value of x in the LLC will be updated to 1.

In fact, no configuration in Bad is reachable from c_0 in $\mathcal{P}_2$, which means that $\mathcal{P}_2$ indeed satisfies the property ϕ. Thus, we have found a sound set of fences for $\mathcal{P}$ w.r.t. ϕ. It is interesting to observe that, although $\mathcal{P}_2$ is correct w.r.t. ϕ, the program still contains runs that are impossible under SC, e.g., the run π_3 given in Figure 7.

Full Fences. Consider a safety property ϕ' defined by (unreachability of) a new set of configurations Bad'. The set Bad' contains all configurations in Bad, and also all configurations where (i) the processes P0 and P1 have both terminated, and (ii) both $r0 and $r3 have

values 0. We show that $\mathcal{P}_2$ violates ϕ', i.e., the set Bad' is reachable from c_0 in $\mathcal{P}_2$. To that end, we construct the run π_3, depicted in Figure 7. (The run can be explained similarly to π_1 and π_2.) At the end of π_3, we reach the configuration c_5 which is in Bad'.

Notice that the run π_3 is not possible under SC, while it is feasible under the SISD semantics even in the presence of the two fences at L8 and L9. The reason π_3 is possible under SISD is *write-read* reordering. More precisely, read events may overtake write events (although not the other way round). In π_3, the write event L4 is issued before the read events L5, L6, and L7, but L4 does not take effect (does not update the LLC) before the read events. There are several ways to prevent the reachability of the set Bad'. One is to replace the `llfence` at L8 and the `ssfence` at L9 by the full fence `fence`, thus obtaining the program $\mathcal{P}_3$ (Figure 8). A full fence `fence` is only enabled when the L1 of the process is empty, and hence it forbids all reorderings of events of the process. In $\mathcal{P}_3$, no configuration in Bad' is reachable from c_0. Thus we have inserted a sound set of fences in $\mathcal{P}$ w.r.t. the set Bad'.

```
data x=0 y=0 z=0

process P0          process P1
registers $r0       registers $r1
begin               $r2 $r3
L1:   x := 1;       begin
L9:   fence;        L4:   z := 1;
L2:   y := 1;       L5:   $r1 := x;
L3:   $r0 := z;     L6:   $r2 := y;
end                 L8:   fence;
                    L7:   $r3 := x;
                    end
```
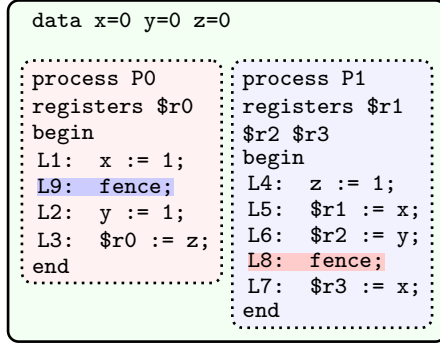
FIGURE 8. The program $\mathcal{P}_3$.

Optimal Sets of Fences. We will describe some optimal sets of fences for the program $\mathcal{P}$. As we will notice, this task is not trivial even for $\mathcal{P}$. Our framework allows to make use of different kinds of fences. We saw above three examples of fences (and we introduce another one in Section 4). The motivation is that different kinds of fences may have different costs. Using a more "light-weight" fence may both increase program performance and reduce network traffic (see Section 7). In that respect, the cost of a full fence is higher than that of an `llfence` or an `ssfence`. The cost assignment is provided by the user of our tool. Let us assume that an `llfence` or an `ssfence` costs 1 unit, and that a full fence costs 2 units. Let F_1 be the set of fences where there is an `ssfence` after L1, and an `llfence` after L6. Then, F_1 is optimal for the program $\mathcal{P}$ w.r.t. the property ϕ. First, F_1 is sound since $\mathcal{P}_2$ (which is the result of inserting the two fences in $\mathcal{P}$) satisfies ϕ, i.e., it does not reach Bad from c_0. Second, F_1 has the minimal cost that guarantees unreachability of Bad. The set F_2 which we get by replacing both the `llfence` and `ssfence` by full fences is also sound. It is also minimal w.r.t. the number of fences (which is 2). However, it is not optimal w.r.t. ϕ since it has a larger cost than F_1. On the other hand, F_2 is optimal w.r.t. the set ϕ'. In fact, there are several optimal sets of fences w.r.t. ϕ' (12 sets to be exact, as reported by our tool). One such a set is F_3 which we get by inserting an `ssfence` after L1, an `llfence` after L2, and an `ssfence` followed by an `llfence` after L6. The set F_3 is not minimal w.r.t. the

number of fences, but optimal w.r.t. the property ϕ'. Notice that there are at least 2^{15} ways to insert three types of fences in the simple program of Figure 1. (Each type may or may not be inserted in any particular position.)

4. 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.

4.1. **Syntax.** The syntax of programs is given by the grammar in Figure 9. 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 take 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.

$$
\begin{aligned}
\langle pgm \rangle &::= \texttt{data } \langle vdecl \rangle^+ \langle proc \rangle^+ \\[6pt]
\langle vdecl \rangle &::= \langle var \rangle \; \texttt{'='} \; (\texttt{'*'} \mid \langle val \rangle) \\[6pt]
\langle proc \rangle &::= \texttt{process } \langle pid \rangle \texttt{ registers } \langle reg \rangle^* \; \langle stmts \rangle \\[6pt]
\langle stmts \rangle &::= \texttt{begin } (\langle label \rangle \; \texttt{':'} \; \langle stmt \rangle \; \texttt{';'})^+ \; \texttt{end} \\[6pt]
\langle stmt \rangle &::= \langle var \rangle \; \texttt{':='} \; \langle expr \rangle \quad \mid \quad \langle reg \rangle \; \texttt{':='} \; \langle var \rangle \quad \mid \\
&\qquad \langle reg \rangle \; \texttt{':='} \; \langle expr \rangle \quad \mid \quad \texttt{llfence} \quad \mid \quad \texttt{fence} \quad \mid \\
&\qquad \texttt{cas '(' } \langle var \rangle \; \texttt{','} \; \langle expr \rangle \; \texttt{','} \; \langle expr \rangle \; \texttt{')'} \quad \mid \\
&\qquad \texttt{syncwr ':' } \langle var \rangle \; \texttt{':='} \; \langle expr \rangle \quad \mid \quad \texttt{ssfence} \quad \mid \\
&\qquad \texttt{cbranch '(' } \langle bexpr \rangle \; \texttt{')'} \; \langle label \rangle
\end{aligned}
$$

FIGURE 9. The grammar of concurrent programs.

4.2. **Configurations.** As illustrated in Figure 10, 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

$$c : (\mathsf{LConf}, \mathsf{LLC})$$

LConf : for each process p, returns the local configuration of p

$(\lambda, \mathsf{RVal}, \mathsf{L1})$: Local configuration of p

λ : the label of the next statement to execute in p

RVal : the values of the local registers in p

$\mathsf{L1}$: $(\mathsf{Valid}, \mathsf{LStatus}, \mathsf{LVal})$, the state of the L1 cache of p

$\mathsf{Valid} \subseteq \mathcal{M}$: the set of shared variables currently in the valid state

$\mathsf{LStatus} : \mathsf{Valid} \to \{\texttt{dirty}, \texttt{clean}\}$: for each $x \in \mathsf{Valid}$, whether x is dirty or clean

$\mathsf{LVal} : \mathsf{Valid} \to \mathcal{V}$: for each $x \in \mathsf{Valid}$, its current value in the L1 cache of p

$\mathsf{LLC}: \mathcal{M} \to \mathcal{V}$: shared part of c, defines for each $x \in \mathcal{M}$ its value in the LLC

FIGURE 10. The definition of a configuration c

that returns, for each process p, the local configuration of p. In the formal definition below, our semantics allows system events to occur non-deterministically. 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.

Below, we give an intuitive explanation of each transition. The formal definition can be found in Figure 11 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.

4.3. Semantics.

4.3.1. *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 (`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.

4.3.2. *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 (`evict`(p, x))**:** An `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 (`wrllc`(p, x))**:** If the entry of x is dirty in the L1 of p, then a `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 (`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.

Instruction Semantics

$$c = (\mathsf{LConf}, \mathsf{LLC}), \ \mathsf{LConf}(p) = (\lambda, \mathsf{RVal}, \mathsf{L1}),$$
$$\mathsf{L1} = (\mathsf{Valid}, \mathsf{LStatus}, \mathsf{LVal}), \ \mathsf{Proc}\,(\lambda) = p, \ \mathsf{Stmt}\,(\lambda) = \sigma$$

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

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

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

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

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

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

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

$$\frac{\sigma = (\mathtt{cbranch}(e)\ \lambda'), \ \mathsf{RVal}(e) = \mathtt{true}}{c \xrightarrow{\lambda} (\mathsf{LConf}\,[p \leftarrow (\lambda', \mathsf{RVal}, \mathsf{L1})]\,, \mathsf{LLC})}$$

$$\frac{\sigma = (\mathtt{cbranch}(e)\ \lambda'), \ \mathsf{RVal}(e) = \mathtt{false}}{c \xrightarrow{\lambda} (\mathsf{LConf}\,[p \leftarrow (\mathsf{Next}\,(\lambda)\,, \mathsf{RVal}, \mathsf{L1})]\,, \mathsf{LLC})}$$

System Event Semantics

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

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

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

FIGURE 11. Semantics of programs running under SiSd.

4.4. **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 4.3 such that it models a program under an SI protocol, it suffices to interpret all write instructions as the corresponding `syncwr` instructions.

4.5. **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. Figure 12 shows four nodes in the transition graph of the program in Figure 1. The configurations c_2 and c_3 are those depicted in Figure 2, while c_6 is the configuration we get from c_2 by adding a clean copy of y with value 1 to the L1 of P1; and c_7 is the configuration we get from c_6 by updating the label of P1 to L7, and the value of $r2 to 1. 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. Figure 12 shows the path of the run ρ_3.
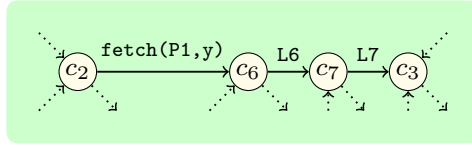

FIGURE 12. Part of the transition graph of the program in Figure 1.

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.

5. Litmus Tests and Comparison with Other Memory Models

In this section, we will first compare the behavior of SISD with other weak memory models. We do this by presenting a sequence of litmus tests that differentiate the SISD semantics from the other models. Each program consists of a number of threads that share a number of variables. Later, we will describe a number of additional litmus tests to clarify the bahavior of the SISD model.

5.1. **Comparing with Other Memory Models.**

```
Initially:  x=0, y=0

process P0      process P1
registers $r1   registers $r2
begin           begin
L1:   x := 1;   L3:  y := 1;
L2:   $r1 := y; L4:  $r2 := x;
end             end

Assertion:  $r1=0, $r2=0
```
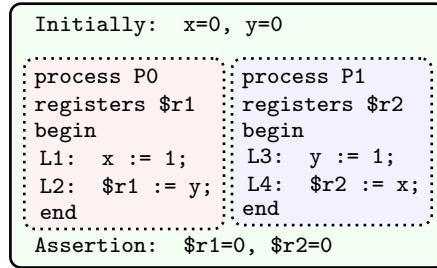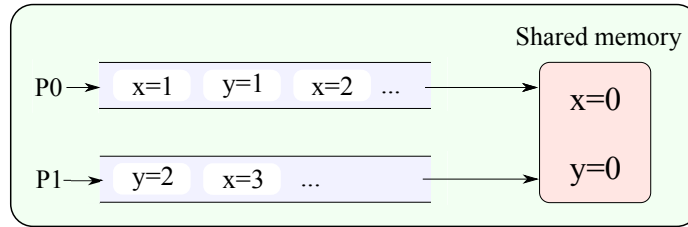
FIGURE 13. SB (Dekker's) algorithm.

FIGURE 14. TSO

Sequential Consistency (SC). The SISD model is weaker than SC. Consider the classic SB (Dekker's) algorithm shown in Figure 13. Under the SC semantics, the program does not have any runs satisfying the assertion, since the operations performed by the process $P0$ are not reordered. If $\$r_1 = 0$, then $x := 1$ must have been executed and the value 1 of x is updated to the memory. At this time, $P0$ sees the value of y as 0 in the memory, which means that $y := 1$ is not executed by $P1$ yet. When $P1$ executes $y := 1$ and $\$r_2 := x$, the value of x in the memory is already 1 and thus 1 is assigned to $\$r_2$.

Any run of a program $\mathcal{P}$ under SC can be simulated by a run of $\mathcal{P}$ under SISD as the following: 1) right after each write operation, the assigned value of the variable is updated to LLC and the variable is invalidated from the local cache immediately; 2) right before each read operation, the value of the variable is fetched from LLC and after reading the value, the variable is immediately invalidated from the local cache.

Total Store Order (TSO). The SISD model and TSO are not comparable. As illustrated in Figure 14, the TSO model inserts a store buffer between each process and the shared memory. When a process executes a write instruction, the instruction is appended to the end of the buffer of the process. At any point of the execution, the instruction at the head of the buffer may nondeterministically be removed and applied to update the memory. When a process reads the value of a variable, it fetches the value from the most recent write operation on the variable in its buffer. If such a write operation is missing, then the value is fetched from the memory.

First, we show that TSO is not weaker than SISD. Consider the program MP in Figure 15. Under the TSO semantics, the program MP does not have any runs that satisfy the assertion. The reason is that the two write operations performed by $P0$ will reach the memory in the same order as they are performed, i.e, $x := 1$ and then $y := 1$. Furthermore, the two read operations performed by $P1$ are not re-ordered according to the TSO semantics. Therefore, if $\$r_2 = 1$ then $P1$ has already seen that the value of x is equal to 1 when it performs
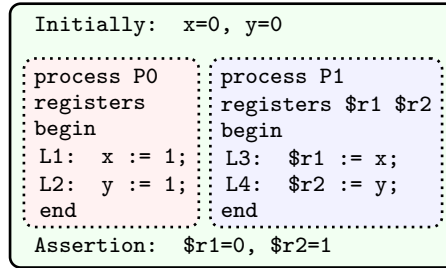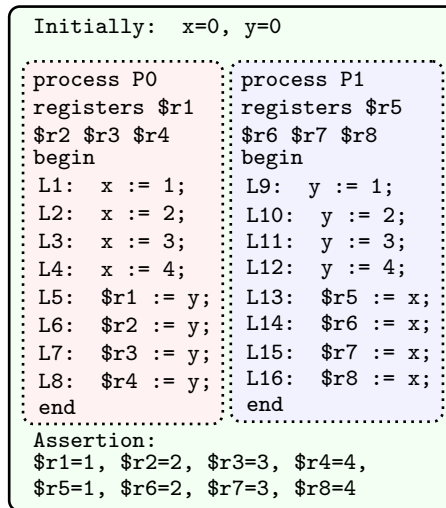
```
Initially:  x=0, y=0

process P0      process P1
registers       registers $r1 $r2
begin           begin
 L1:  x := 1;    L3:  $r1 := x;
 L2:  y := 1;    L4:  $r2 := y;
 end             end

Assertion:  $r1=0, $r2=1
```

FIGURE 15. Program MP.

```
Initially:  x=0, y=0

process P0       process P1
registers $r1    registers $r5
$r2 $r3 $r4      $r6 $r7 $r8
begin            begin
 L1:  x := 1;     L9:   y := 1;
 L2:  x := 2;     L10:  y := 2;
 L3:  x := 3;     L11:  y := 3;
 L4:  x := 4;     L12:  y := 4;
 L5:  $r1 := y;   L13:  $r5 := x;
 L6:  $r2 := y;   L14:  $r6 := x;
 L7:  $r3 := y;   L15:  $r7 := x;
 L8:  $r4 := y;   L16:  $r8 := x;
 end              end

Assertion:
$r1=1, $r2=2, $r3=3, $r4=4,
$r5=1, $r6=2, $r7=3, $r8=4
```

FIGURE 16. Program ReadSeq.

the assignment $r_1 := x$. In contrast, the program MP has the following run under SISD that satisfies the assertion. First, the process $P0$ assigns 1 to x and y respectively, but only updates the value of y to the LLC. Next, $P1$ fetches the values of x and y (0 and 1 respectively) from the LLC, and assigns them to the registers using the instructions $r_1 := x$ and $r_2 := y$, which means the assertion will be satisfied.

The program ReadSeq in Figure 16 shows that SISD is not weaker than TSO. Under TSO, the program exhibits a run that satisfies the assertion as follows. First, $P0$ executes the instructions $x := 1$, $x := 2$, $x := 3$, and $x := 4$, one by one, and the corresponding operations are put in its buffer. Then, $P1$ executes the instructions $y := 1$, $y := 2$, $y := 3$, and $y := 4$, one by one, again putting the corresponding operations in its buffer. Now, the memory is updated with $x := 1$ after which $P1$ executes $r_5 := x$, thus assigning 1 to r_5. Following this, the memory is updated with $x := 2$ after which $P1$ executes $r_6 := x$, thus assigning 2 to r_6. Finally 3 and 4 are assigned to r_7 and r_8 respectively in similar manners. A similar sequence of operations is performed assigning 1, 2, 3 and 4 to r_1, r_2, r_3, and r_4 respectively.

However, under the SISD semantics, the ReadSeq program does not have any runs that can satisfy the assertion. More specifically, since the processes do not have any store buffers, at most three different values of a variable can be kept in this example. I.e., one in the local
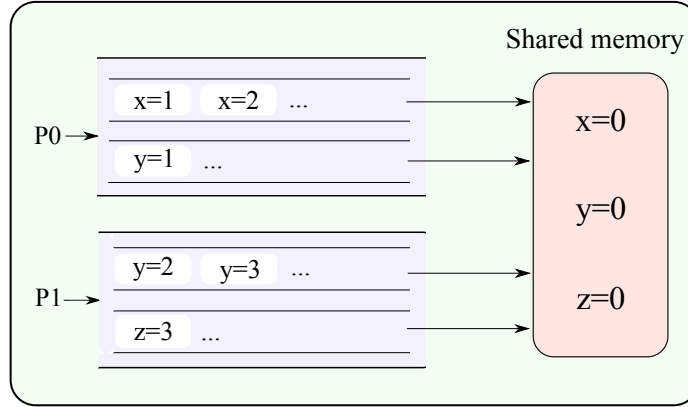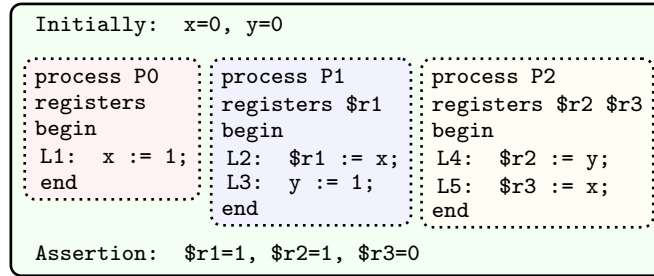
FIGURE 17. PSO

FIGURE 18. Program WRC.

cache of the process which executes the write operation of the variable, one in the local cache of the process which executes the read operation of the variable, and one in the LLC. When there are four or more values assigned to a variable in a similar manner as in ReadSeq, the assertion that each process reads all the values of the variable in the same order as it is written by the other process cannot be satisfied.

Partial Store Order (PSO). The SISD model and PSO are not comparable. In PSO, a store buffer is inserted for each variable, between each process and the shared memory as illustrated in Figure 17 .

To illustrate the difference between SISD and PSO, we consider the program WRC in Figure 18. Under the PSO semantics, the program WRC does not have any runs that satisfy the assertion. More precisely, if $\$r_1 = 1$ then the write operation $x := 1$ by $P0$ must have reached the memory before the write operation $y := 1$ has been performed by $P2$. Furthermore, if $\$r_2 = 1$ holds then the write operation $y := 1$ must have reached the memory before the instruction $\$r_3 := x$ has been performed by $P2$. Since read instructions are not re-ordered in PSO and since $x := 1$ reaches the memory before $y := 1$ it follows that the value of x is equal to 1 in the memory when $\$r_3 := x$ is performed by $P2$, and hence $\$r_3 = 1$.

The program WRC has the following run under SISD that satisfies the assertion. The process $P2$ fetches the initial value 0 of the variable x. The process $P0$ assigns 1 to x and updates the value to the LLC. The process $P1$ fetches the value 1 of x from the LLC and then executes the instruction $\$r_1 := x$ which means that the value of $\$r_1$ is equal to 1. Then,
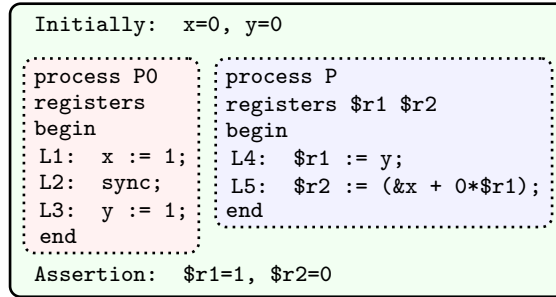
```
Initially:  x=0, y=0

process P0      process P
registers       registers $r1 $r2
begin           begin
 L1:  x := 1;    L4:  $r1 := y;
 L2:  sync;      L5:  $r2 := (&x + 0*$r1);
 L3:  y := 1;   end
 end

Assertion:  $r1=1, $r2=0
```
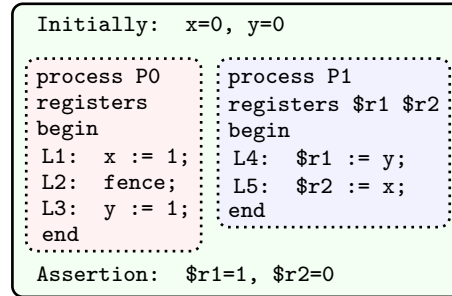
FIGURE 19. PwrEg

```
Initially:  x=0, y=0

process P0      process P1
registers       registers $r1 $r2
begin           begin
 L1:  x := 1;    L4:  $r1 := y;
 L2:  fence;     L5:  $r2 := x;
 L3:  y := 1;   end
 end

Assertion:  $r1=1, $r2=0
```

FIGURE 20. SisdEg

$P1$ assigns the value 1 to the variable y and updates the value to the LLC. The process $P2$ fetches the value of y from the LLC and then executes the instruction $\$r_2 := y$ which means that $\$r_2 = 1$. Finally, $P2$ executes the instruction $\$r_3 := x$ and we get $\$r_3 = 0$.

As PSO is weaker than TSO, we can use the same example in Figure 16 to show that the ReadSeq program also has a run under PSO satisfying the assertion, which means that SISD is not weaker than PSO.

POWER/ARM. The SISD model and POWER are not comparable. We show that the POWER memory model is not weaker than SISD with the program PwrEg in Figure 19. The program does not have any runs that satisfy the assertion under POWER, since cycles of the form write-(sync)→write-(read from)→ read-(address dependency)→read-(from read)→ are not allowed by POWER. The sync instruction maintains the order between the two write instructions and the address dependency maintains the order between the two read instructions, which make the cycle impossible.

However, under SISD the program SisdEg in Figure 20 has the run that satisfies the assertion. The process $P1$ fetches the initial value 0 of the variable x. The process $P0$ assigns 1 to x, updates the value to the LLC, assigns 1 to y, and updates the value to the LLC again. The process $P1$ fetches the value 1 of y from the LLC and then executes the instruction $\$r_1 := y$. Therefore, the value of $\$r_1$ is equal to 1. Finally, $P1$ executes the instruction $\$r_2 := x$ and we get $\$r_2 = 0$.

We can show that SISD is not weaker than POWER with the program ReadSeq in Figure 16.

We can also show that the SiSd model and ARM are not comparable with the programs in Figure 16, Figure 19 and Figure 20.

Relaxed Memory Order (RMO). We can show that SiSd is not weaker than RMO again with the program `ReadSeq` in Figure 16, since RMO allows the reorder between both 1) write and write operations and 2) read and read/write operations.
    We keep the other direction open since we have not found any examples as the proof.

5.2. **Further Litmus Tests.** We describe the behavior of the SiSd model for three more litmus tests.

Load-Buffering (LB). The `LB` program in Figure 21 does not have any runs that can satisfy the assertion under SiSd. If $\$r_1 = 1$, then process $P1$ must have executed the write instruction $x := 1$ and updated the value 1 of x to the LLC before $P0$ has executed the instruction $\$r_1 := x$. This means when $P1$ executes the read instruction $\$r_2 := y$, the value of y has not been updated by $P0$, and thus $\$r_2 = 0$.

```
Initially:  x=0, y=0

process P0        process P1
registers $r1     registers $r2
begin             begin
L1:  $r1 := x;    L3:  $r2 := y;
L2:  y := 1;      L4:  x := 1;
end               end

Assertion:  $r1=1, $r2=1
```
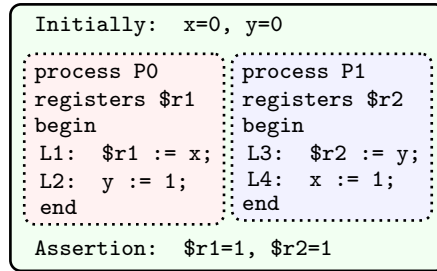
FIGURE 21. Program LB.

ISA2. The program `ISA2` in Figure 22 has the following run under SiSd that satisfies the assertion. The process $P2$ fetches the initial value 0 of the variable x. The process $P0$ assigns 1 to x and y, and updates the value to the LLC. The process $P1$ fetches the value 1 of y from the LLC and then executes the instruction $\$r_1 := y$, which means that the value of $\$r_1$ is equal to 1. Then, $P1$ assigns the value 1 to the variable z and updates the value to the LLC. The process $P2$ fetches the value of z from the LLC and then executes the instruction $\$r_2 := z$, which means that $\$r_2 = 1$. Finally, $P2$ executes the instruction $\$r_3 := x$ and we get $\$r_3 = 0$.

IRIW. The program `IRIW` in Figure 23 has the following run under SiSd that satisfies the assertion. The processes $P1$ and $P3$ fetch the initial values 0 of the variables y and x respectively. The process $P0$ assigns 1 to x and updates the values to the LLC. The process $P1$ fetches the value 1 of x from the LLC and executes the instruction $\$r_1 := x$, which means that the value of $\$r_1$ is equal to 1. Then $P1$ executes the instruction $\$r_2 := y$, which means that $\$r_2 = 0$. After that, the process $P2$ assigns the value 1 to the variable y and updates the value to the LLC. The process $P3$ fetches the value of y from the LLC and then executes the instruction $\$r_3 := y$, which means that $\$r_3 = 1$. Finally, $P3$ executes the instruction $\$r_4 := x$ and we get $\$r_4 = 0$.
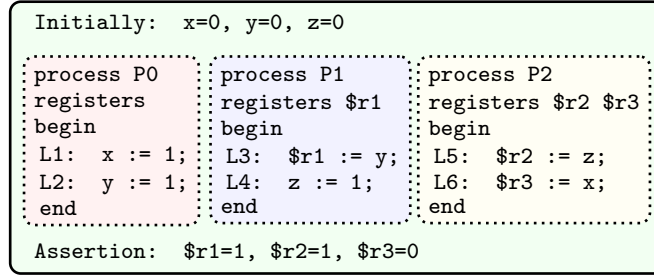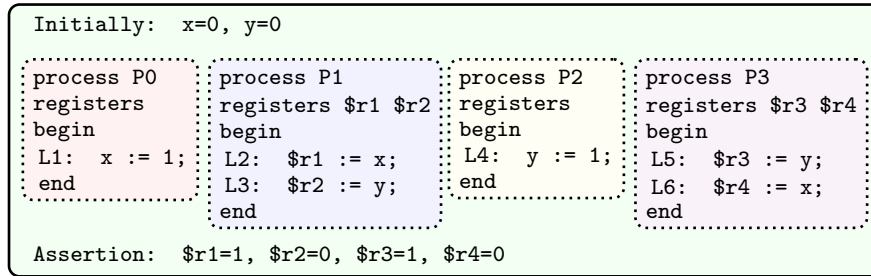
```
Initially:   x=0, y=0, z=0

process P0        process P1          process P2
registers         registers $r1       registers $r2 $r3
begin             begin               begin
 L1:   x := 1;     L3:   $r1 := y;     L5:   $r2 := z;
 L2:   y := 1;     L4:   z := 1;       L6:   $r3 := x;
 end              end                 end

Assertion:   $r1=1, $r2=1, $r3=0
```

FIGURE 22. Program `ISA2`.

```
Initially:   x=0, y=0

process P0      process P1            process P2        process P3
registers       registers $r1 $r2     registers         registers $r3 $r4
begin           begin                 begin             begin
 L1:   x := 1;   L2:   $r1 := x;       L4:   y := 1;     L5:   $r3 := y;
 end             L3:   $r2 := y;       end               L6:   $r4 := x;
                 end                                     end

Assertion:   $r1=1, $r2=0, $r3=1, $r4=0
```

FIGURE 23. Program `IRIW`.

6. Fence Insertion

In this section we describe our fence insertion procedure, which is closely related to the algorithm described in [LNP+12]. 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 $\mathsf{Next}(\lambda))$[2]. 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 $(\lambda, \mathtt{llfence})$ and $(\lambda, \mathtt{ssfence})$), we assume that fences are always inserted in some fixed order.

Definition 6.1 (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 6.2 (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 $\kappa(G) < \kappa(F)$.

---

[2]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.

Observe that the optimality is evaluated with the number of occurrences of fences in the source program and the costs of different fences.

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

Definition 6.3 (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 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

- T is a hitting set of S, and
- there is no hitting set T' of S such that $\kappa(T') < \kappa(T)$.

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;
```

FIGURE 24. The fence insertion algorithm.

We present our fence insertion algorithm in Figure 24. 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 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 $\mathtt{req} = \varnothing$.

In the loop from 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**$(\mathcal{P} \oplus F, \phi)$ returns $\bot$ if ϕ is unreachable in $\mathcal{P} \oplus F$, and a witness run otherwise. If $\mathcal{P} \oplus F$ 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 $\mathcal{P} \oplus F$ does not satisfy the safety property ϕ, then we proceed to analyze the witness run π. The witness analysis procedure is outlined in Section 6.1. 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**$(\mathcal{P} \oplus F, \pi)$ 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.

6.1. **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

• all sound fence sets have at least one fence in common with G and
• 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.

As an example, consider Figure 25. 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: $c_0 = (\text{L0}, \text{ssfence})$, $c_1 = (\text{L1}, \text{ssfence})$,

Program fragment	Witness run

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

FIGURE 25. Left: Part of a program $\mathcal{P}$, containing three instructions of the thread P0. Right: A part of a counter-example run π of $\mathcal{P}$.

$c_2 = (\texttt{L0}, \texttt{fence})$, 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**.

7. 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 7.1). 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 7.2).

7.1. **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.[3] 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–* [Sco13], Lamport's bakery algorithm *–bakery–* [Lam74], the MCS queue lock *–mcsqueue–* [MCS91], the CLH queue lock *–clh–* [MLH94], and Dekker's algorithm *–dekker–* [Dij02]. They also contain a work scheduling algorithm *–postgresql–*[4], and an idiom for double-checked locking *–dclocking–* [SH96], as well as two process barriers *–srbarrier–* [Sco13] and *–treebarrier–* [MCS91]. The second set of benchmarks are based on the Splash-2 benchmark suite [WOT+95]. We use the race detection tool Fast&Furious [RK15b] to detect racy parts in the Splash-2 code. We then manually extract models capturing the core of those parts.

---

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

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

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

TABLE 1. Automatic fence insertion for SiSd.

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 data race from it. After careful code inspection we find that it is possible for all workers but one to terminate prematurely, leaving one worker to do all remaining work, which is caused by data race. [SLKR16] 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 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 7.2. 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.

7.2. **Simulation Results.** Here we show the impact of different choices of fences when executing programs. In particular we show that an optimal fence set using 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. Here network traffic refers to the traffic in both the on-chip interconnection network and the memory bus. We account for all the traffic due to coherence messages. We evaluate the micro-benchmarks analyzed in the previous section and the whole Splash-2 benchmark suite [WOT+95], running the applications from beginnig to end, but presenting results only for the parallel phase of the applications. 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 [MSB+05]. 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 implementing the SiSd protocol described in Section 2 and 32KB, 4-way, private L1 caches and a logically shared but physically distributed L2 with 64 banks of 256KB, 16-way each.

Cost of Fences: Our automatic fence insertion tool employs different weights in order to insert the optimal amount of fences given the cost of each fence. 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.

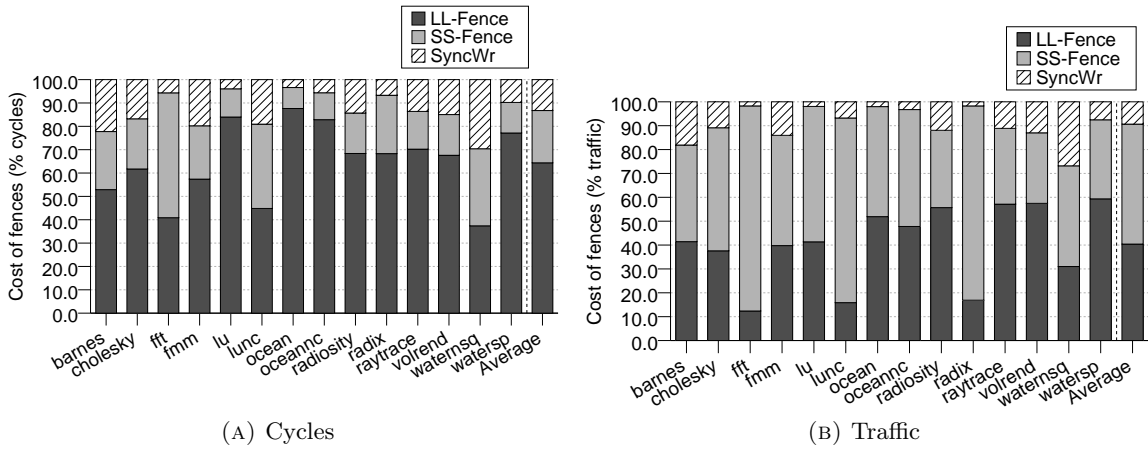(A) Cycles   (B) Traffic

FIGURE 26. Percentage of cycles and traffic that each type of fence cost

We calculate the cost of fences in terms of execution as indicated in equation 7.1, where $latency_{fence}$ is the time in cycles required by the fence, $misses_{si}$ is the number of misses

caused by self-invalidation, and $latency_{miss}$ is the average latency of such misses. According to this equation, and considering a protocol implementing the DoI state described in Section 2, the average percentage of cycles (execution time) employed by each type of fence when running the Splash2 benchmarks is the following: the cost of an `llfence` is 64.4%, the cost of an `ssfence` is 22.4%, and the cost of a `syncwr` is 13.2%, as shown in Figure 26a.

$$time_{fence} = latency_{fence} + misses_{si} \times latency_{miss} \qquad (7.1)$$

The cost of the fences in traffic is calculated as indicated in equation 7.2, 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. In percentage, the cost of the each type of fence on average in terms of traffic is 40.4% for an `llfence`, 50.3% for an `ssfence`, and 9.3% for a `syncwr`, as shown in Figure 26b. Thus, the weights assigned to fences in our tool seem reasonable.

$$traffic_{fence} = sd * traffic_{wt} + misses_{si} \times traffic_{miss} \qquad (7.2)$$

Cache Misses: As mentioned, the fences affect the cache miss rate. Figure 27 shows clearly the effect of self-invalidation and self-downgrade on misses. First we show the misses due to cold capacity and conflict misses (*Cold-cap-conf*), which, in general, are not affected by the type of fences. However, in some cases reducing the self-invalidation can give the appearance of extra capacity misses because of having a more occupied cache. The graph does not plot coherence misses since fenced programs on SISD coherence do not induce cache misses due to coherence transactions. The second kind of miss is named as *Bypass*. These misses are due to atomic operations which cannot use the data in the private cache, but need to access it from the shared cache. They are very frequent in the micro-benchmarks (Figure 27a), which are synchronization intensive, but almost unnoticeable for the Splash2 benchmarks (Figure 27b). Finally, we show the misses caused by self-invalidation *Self-inv*. These are the ones which number is reduced, when applying the mixed fences, but also when employing the DoI state, since dirty words are not invalidated.

Traffic: As already mentioned, traffic is also affected by the type of fences employed. Figure 28 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.

Execution Time: Finally, we show the impact on execution time, which is affected by the reductions in cache misses and traffic. Figure 29 shows simulated execution time for both the micro-benchmarks (Figure 29a) and the Splash2 benchmarks (Figure 29b). 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, `llfences` 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.

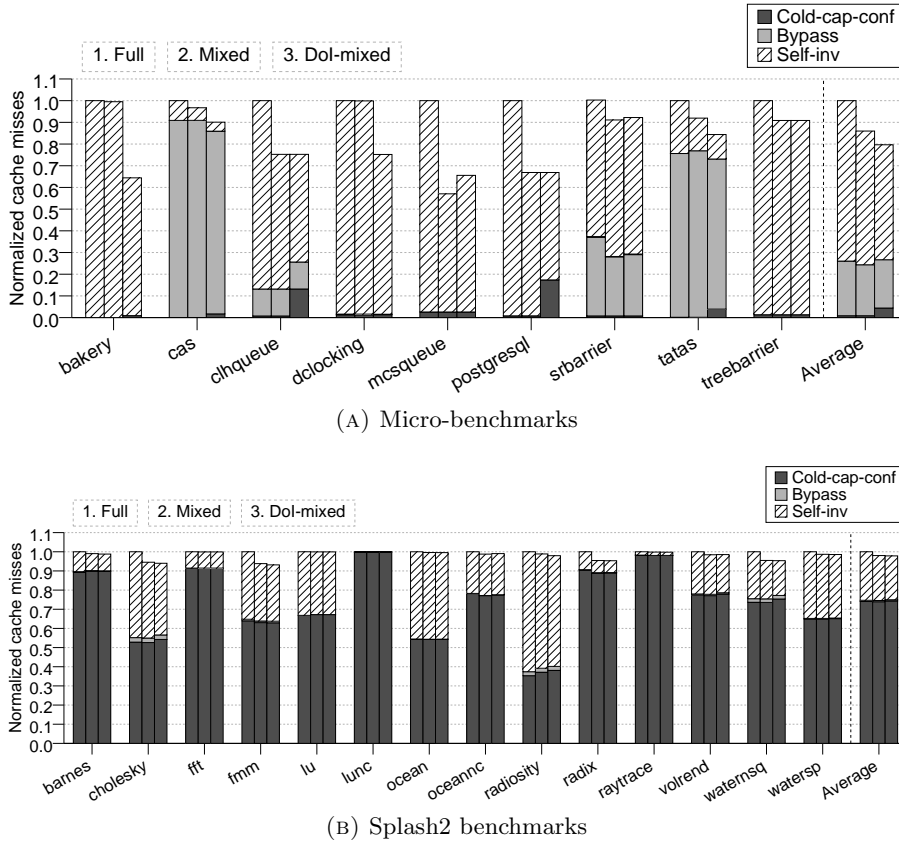(A) Micro-benchmarks

(B) Splash2 benchmarks

FIGURE 27. Normalized cache misses under different fence sets and protocol states

Implementing the DoI state 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. Applications using more synchronization would benefit to a large extent from the use of mixed fences.

8. 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 [BMM11]. In our framework, this would mean that we consider program traces (in the sense of Shasha and Snir [SS88]), 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.

(A) Micro-benchmarks
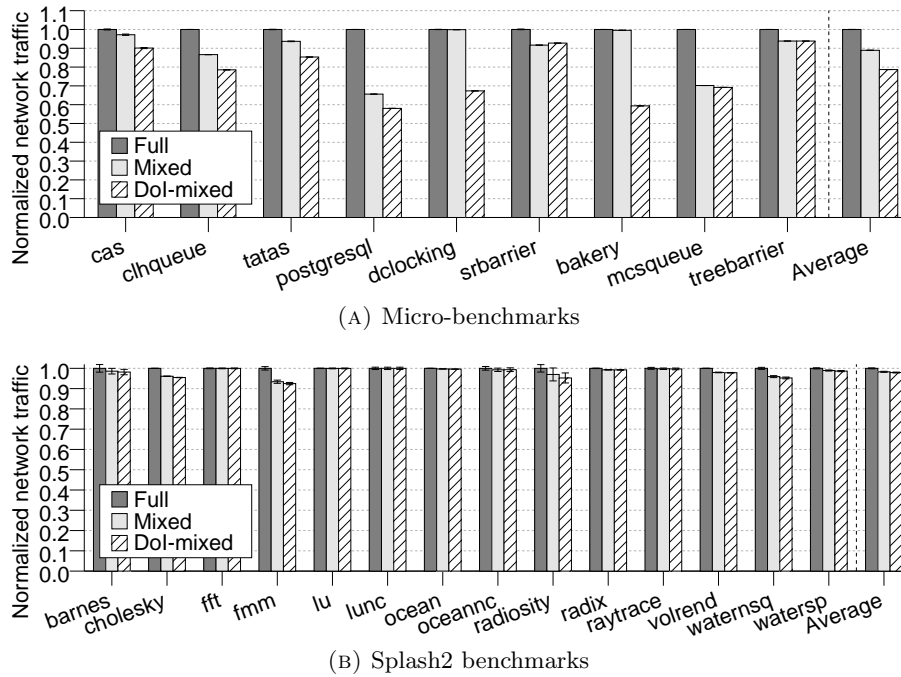

(B) Splash2 benchmarks

FIGURE 28. Normalized network traffic under different fence sets and protocol states

This measurement will provide more accuracy when, for instance, fences with different weights are inserted in a loop computation in a branching program.

REFERENCES

[AAC+12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *TACAS*, pages 204–219. Springer, 2012.
[ADC11] Thomas J. Ashby, Pedro Díaz, and Marcelo Cintra. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Transactions on Computers (TC)*, 60(4):472–483, April 2011.
[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *ISCA*, pages 2–14, 1990.
[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, pages 508–524, 2014.
[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *Programming Languages and Systems*, pages 533–553. Springer, 2013.
[BMM11] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *ICALP (2)*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.
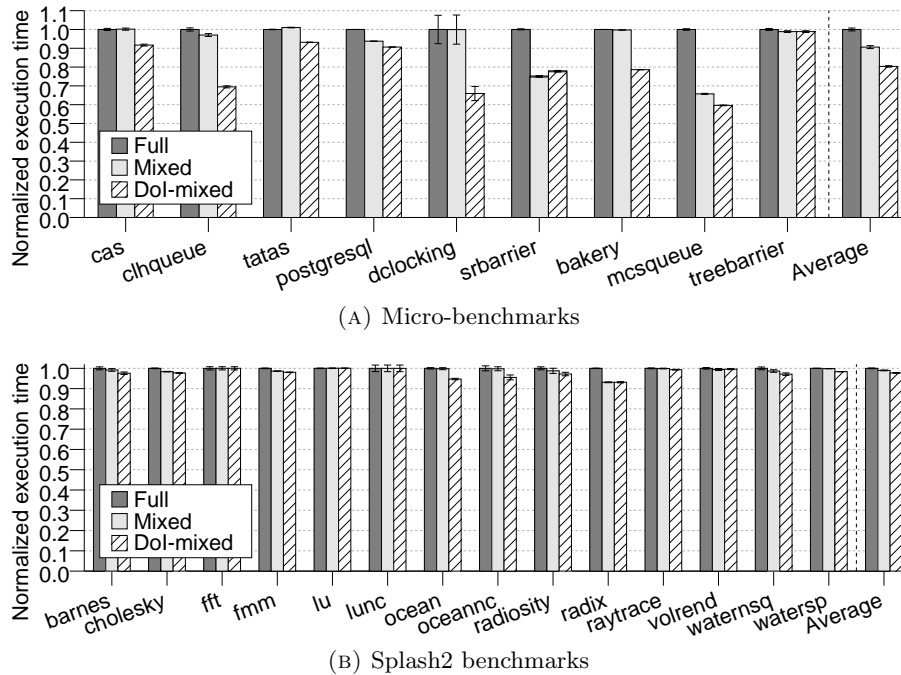
(A) Micro-benchmarks

(B) Splash2 benchmarks

FIGURE 29. Execution time under different fence sets and protocol states

[CKS⁺11]   Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *PACT*, pages 155–166, 2011.

[CL05]   David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA*, pages 21–28, 2005.

[Dij02]   E. W. Dijkstra. *Cooperating sequential processes*. 2002.

[DRHK15]   Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. An efficient, self-contained, on-chip, directory: DIR₁-SISD. In *PACT*, pages 317–330, 2015.

[DSS06]   David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208, 2006.

[GHS09]   Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 321–336, 2009.

[GHS11]   Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Verification of STM on relaxed memory models. *Formal Methods in System Design*, 39(3):297–331, 2011.

[HHB⁺14]   Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In *ASPLOS*, pages 427–440, 2014.

[HS08]   Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[KK11]   Stefanos Kaxiras and Georgios Keramidas. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, 30(5):54–65, September 2011.

[KR13]   Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *ISCA*, pages 535–547, 2013.

[KRHK16]   Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. Building heterogeneous unified virtual memories UVMS without the overhead. *ACM TACO*, 13(1):1:1–1:22, 2016.

[KVY10]   Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *FMCAD*, pages 111–119. IEEE, 2010.

[Lam74]   Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17, August 1974.

[Lam79]   Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (TC)*, 28(9):690–691, September 1979.

[LNP+12]   Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440, 2012.

[LW95]   Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *ISCA*, pages 48–59, 1995.

[MCS91]   J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9, February 1991.

[MLH94]   Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.

[MSB+05]   Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, September 2005.

[RDK15]   Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies. In *HPCA*, pages 186–197, 2015.

[RK12]   Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *PACT*, pages 241–252, 2012.

[RK15a]   Alberto Ros and Stefanos Kaxiras. Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In *ISCA*, pages 427–438, 2015.

[RK15b]   Alberto Ros and Stefanos Kaxiras. Fast&furious: A tool for detecting covert racing. In *PARMA and DITAM*, pages 1–6, 2015.

[RK16]   Alberto Ros and Stefanos Kaxiras. Racer: TSO consistency via race detection. In *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[SA15]   Hyojin Sung and Sarita V. Adve. DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations. In *ASPLOS*, pages 545–559, 2015.

[Sco13]   Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, 2013.

[SH96]   Douglas C. Schmidt and Tim Harrison. Double-checked locking - an optimization pattern for efficiently initializing and accessing thread-safe objects. In *PLoP*, 1996.

[SKA13]   Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient hardware support for disciplined non-determinism. In *ASPLOS*, pages 13–26, 2013.

[SLKR16]   Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *ISPASS*, 2016.

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

[WOT+95]   Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.