Offline Symbolic Analysis to Infer Total Store Order*

Dongyoon Lee†, Mahmoud Said‡, Satish Narayanasamy†, Zijiang Yang‡
University of Michigan, Ann Arbor† Western Michigan University‡

Abstract

Ability to record and replay an execution can significantly help programmers debug their programs, especially parallel programs. Deterministically replaying a multiprocessor's execution under a relaxed memory model has remained a challenging problem. This is an important problem as most modern processors only support a relaxed memory model to enable many performance critical optimizations. The most common consistency model implemented in processors is the Total Store Order (TSO).

We present an efficient and low-complexity processor based solution for recording and replaying under the Total Store Order (TSO) memory model. Processor provides support for logging data fetched on cache misses. Using this information each thread can be deterministically replayed. A TSO-compliant casual order between the shared-memory accesses executed in different threads is then inferred using an offline algorithm based on Satisfiability Modulo Theory (SMT) solver. We also discuss methods to bound the search space during offline analysis and several optimizations to reduce the offline analysis time.

1. Introduction

A record and replay system has a number of different applications such as time travel debugging [21], forensics [15] and fault tolerance [12]. Perhaps one of the most important application is that deterministic replay can help programmers reproduce and understand non-deterministic concurrency bugs.

Software [4, 15, 23, 32] solutions are preferable, but they incur significant performance cost for precisely recording the causal order between shared-memory operations to support multiprocessor replay. Recent solutions [4, 23, 32] avoid recording the causal order to reduce the online performance cost. They rely on offline search [4] or pure chance to reproduce the causal order offline [32]. However, since they do not bound the search space, they cannot guarantee to find the causal order in a reasonable amount of time. Hardware solutions [13, 18, 26–28, 37, 39] can record for a negligible performance cost, and therefore could be used even in production systems. However, they require a complex hardware support for monitoring coherence messages to detect causal order between shared accesses and log them precisely. A low complexity solution is a must if hardware vendors were to provide replay support.

One another important limitation of many prior hardware *and* software solutions is that they guarantee replay only for sequentially consistent (SC) executions. However, most modern processors support only a relaxed memory model as SC disallows many common optimizations. For instance, SPARC and x86 based processors support variants of the Total Store Order (TSO). If a program execution violates SC while running on these processors, prior solutions [26, 28, 39] may not be able to correctly replay it. This is a serious limitation for a replay tool as non-sequentially consistent program executions are perhaps the most difficult to understand and debug for a programmer.

To address the above limitation, RTR [40] and LReplay [13] proposed additional hardware support for detecting memory accesses that violate sequentially consistency at runtime, and log their values explicitly so that they can be replayed correctly. However, this approach not only adds additional complexity to already complex recording hardware which needs to precisely detect and log the causal order for shared memory accesses, but also could increase the log size.

In this paper we present a low complexity hardware solution for supporting replay under the most commonly implemented memory model – Total Store Order (TSO). We leverage the observation made by Lee et al. [22] that logging initial register state and cache miss data is sufficient for replaying each thread in a multiprocessor. The intuition here is that a processor can observe a value produced by another processor or an external system entity only by issuing a cache miss request and fetching the corresponding memory block. Thus, we can guarantee to deterministically replay the exact same sequence of instructions and their input/output values of each recorded thread. Lee et al. [22] also proposed an offline analysis algorithm to determine a sequentially consistent causal order between shared accesses replayed across all the threads. The causal order is guaranteed to reproduce the erroneous program state seen during any recorded buggy execution. This approach has the advantage in that it requires only a simple hardware extension (primarily for logging cache misses) and delegates much of the problem to an offline search. But their solution was applicable only for a sequentially consistent processor model.

In this paper we observe that cache miss log is sufficient to deterministically replay each thread even under the TSO memory model. To determine a TSO-compliant causal order between shared accesses, we present a new offline symbolic analysis algorithm based on Yices [16] Satisfiability Modulo Theory (SMT) solver. TSO differs from SC in that it relaxes store-to-load order and store atomicity [5]. The first relaxation allows a load to be scheduled ahead of an earlier store in the same thread provided they are accessing different locations. The second relaxation allows a store's value to be made visible to a local load before it is made visible to remote loads. We discuss how these two relaxations can be encoded as first-order logic constraints and reproduce a TSO-compliant causal order using an SMT solver.

To bound the search space during offline analysis, we propose to log certain hints during recording. At periodic intervals, all the processor cores simultaneously record the number of committed memory operations along with the number of stores pending in its local store buffer. Using this information, we show that we can legally partition a multi-threaded program execution into smaller bounded intervals and determine a causal order for memory accesses in each interval separately. In our mechanism, hints needed for bounding the search can be logged without any additional communication between the processor cores.

*The work was funded in part by NSF grants CCF-0811287, CCF-0916770, and CNS-0905149.

To further reduce offline analysis time, our offline analysis eliminates a majority of cache hits from the offline search. This is based on our observation that the causal order for most cache hits can be trivially inferred from the causal order between cache misses.

We analyze the log size and performance of the offline symbolic analyzer using Apache, MySQL, Parsec, and Splash benchmarks for the TSO memory model. We find that searching for a valid TSO order is as efficient as searching for a SC order, and for some programs could be even faster than SC. We compare the efficiency of our system to two earlier approaches, one that records precise shared-memory dependency [18] and another that uses offline analysis [22].

By sacrificing precision in logging causal order, we manage to design a low-complexity processor solution. The tradeoff is the offline analysis cost. However, offline analysis need to be performed only once. Once shared memory dependencies are resolved, later replays can be very efficient. We believe that developers would be willing to pay an one-time cost to reproduce a bug (by replaying a few seconds that preceded a crash) that manifested in the production sites and during beta-testing which is where a low-overhead processor recording solution would be crucial.

2. Background

A multi-threaded program has two sources of non-determinism – program input and shared-memory dependencies. In this section we discuss how recording the data fetched on cache misses is sufficient for capturing program input [30] and for deterministically replaying each thread in isolation [22]. Later in Section 3 we describe a new offline analysis for reproducing the TSO causal order for the shared-memory operations replayed across all the threads.

2.1 Load-Based Input Logging

One common solution for recording program input is to checkpoint the initial memory and register state, and then record non-deterministic system events such as signals, interrupts, DMA, etc., along with their timestamps. However, this approach is system-dependent as the set of non-deterministic events are specific to a particular operating system. As a result, it is difficult to device a solution that can record and replay across different operating system environments.

An alternative approach is load value logging [10, 30]. Past work has shown that recording the initial register state and then logging the values of the reads (including load instructions and instruction fetches) that *first access* a memory location is sufficient for deterministically replaying a program's execution [30]. Using the recorded information, a dynamic instrumentation tool like Pin [24] can be used to replay the recorded execution. The replay can reproduce exactly the same sequence of instructions along with their input and output values. In the case of memory operations, their effective addresses are also reproduced.

Load value logging approach is system-independent in that it enables replay across different operating system environments. Intel's PinPlay tool [29, 33, 34] exploits this property to enable cross-platform architectural simulation. But PinPlay is a software-only solution that is nearly 100x slower than native execution.

2.2 Processor Support

Logging memory accesses that first access a location in software could be expensive. In a recent work, which we refer to as Replay-SMT, Lee et al. [22] discussed a complexity-effective processor solution that required just logging memory blocks fetched on cache misses. The only additional hardware state that Replay-SMT requires is a "log" bit per cache block in the outermost private cache of each processor core to determine whether an access is a first access to that location or not. Logs are stored in the local cache

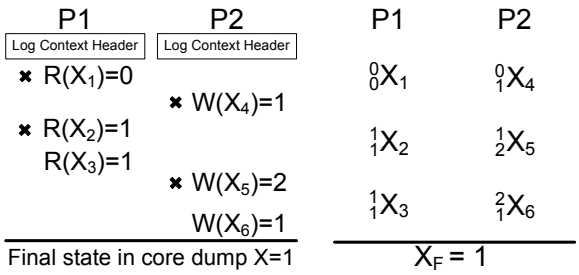

Figure 1: Replaying two threads using cache miss log

and continuously written to main memory. To begin recording a thread in a processor core, the operating system first records the *context header*, resets "log" bits and turns on logging for the processor core. The header contains the initial register state, process and thread identifier and the value of the timestamp counter of the processor core. Thereafter, when a processor reads or writes to a cache block whose log bit is not set, it logs the cache block data, its physical address and the current memory count. The memory count of a processor core is the number of memory operations committed since the last context switch. It helps the replayer determine when to read values from the log while emulating a memory operation. When a new cache block is fetched into the cache its log bit is reset. All uncacheable reads and return value of RDTSC (ReaD TimeStamp Counter) are also logged.

The above approach effectively logs cache blocks fetched on a cache miss, and also pre-fetched cache blocks when they are accessed for the first time. Thus, it guarantees to log the values of all the first accesses to memory locations during an execution. This information along with the initial register state is sufficient to replay the execution.

When a cache block is logged due to a store, Replay-SMT records the state of the cache block before the store modifies it. This helps us recreate the memory state before a store's execution during replay. Note that upgrade misses for store operations will not result in additional logs as the corresponding cache block's log bit would be set. More details on how this mechanism could support context switches, operating system code replay, self-modifying code, page faults, etc., can be found here [22].

2.3 Replaying Threads

The cache miss data and the context header is also sufficient to replay each thread when a multi-threaded program executes on a multi-processor (which includes a DMA processor). The reason is that, irrespective of the memory model, the underlying cache coherence mechanism ensures that when a processor modifies a location, the corresponding cache block is invalidated in other processors. As a result, a cache miss is always generated when a processor wants to read a value written by another processor core.

Figure 1(left) shows an example with two threads concurrently accessing a shared-memory location `X`. Right sub-script is a unique identifier for the dynamic memory access. Labels `R` denotes a read and `W` denotes a write. Memory operations marked with crosses on the left result in cache misses and therefore are logged. When P_1 executes `R(`X_2`)`, it would encounter a cache miss as the cache block containing `X` would have been invalidated when P_2 had executed `W(`X_4`)`. The new cache block fetched would contain the new value for `X` written by `W(`X_4`)` and would be logged. Thus, the cache miss log captures the input values to each thread, including the values produced by remote threads. This allows us to deterministically replay each thread and reproduce its sequence of memory operations, their addresses and values independent of other threads.

The picture on the right in the Figure 1 shows the memory trace generated by individually replaying each thread. Each memory access is represented as follows. The literal `X` denotes the address. The left superscript and subscript denote the state of `X` before and after the execution of the memory access respectively (for loads the two values will be the same). We refer to these values of a memory operation as the `old` and `new` values respectively. At the end of recording an execution, the operating system also dumps the final memory state of the program. The final state is denoted as X_F. In Section 3, we discuss an offline analysis that uses the memory trace of each thread and the final state to reproduce a causal order for shared accesses under the TSO memory model.

3. Reproducing Shared-Memory Dependencies under TSO Memory Model

In Section 2, we reviewed how a load-value based logging enables each thread to be deterministically replayed with the same sequence of memory operations along with their addresses, old and new values. However, to debug and understand parallel executions, we need the TSO-compliant causal order between shared-memory operations as well, which we now discuss.

3.1 Overview of Offline Symbolic Analysis

The goal of our offline analysis is to find a valid *causal order* between all the memory accesses executed concurrently across all the threads. The algorithm takes the memory trace of each thread and the final memory state as input. The memory trace for a thread is produced by deterministically replaying using its cache miss log (Section 2.2). It contains the memory accesses in the program order. For each memory access, we have its effective address, old value, new value and information about whether it was a cache hit or a miss. To produce the causal order between the memory accesses, we determine the memory ordering constraints that need to be satisfied, encode them as a quantifier free first-order logic formula and use a Satisfiability Modulo Theory(SMT) solver called Yices [16] to find a solution.

The algorithm to encode all the necessary constraints in the first-order logic formula is presented in Algorithm 1. A valid causal order should satisfy two constraints. First, any memory access M's old value should be same as the new value of the memory access to the same location that immediately precedes M in the derived causal order. We call this constraint as the *coherence constraint*(C_H). Because, it is the property of coherence that ensures that there exists a total global order between all memory accesses to a location under any memory model.

Second, the causal order should obey the memory ordering constraints specified by a particular memory model. We refer to these constraints as the *memory model*(C_M) constraints. These constraints of course vary across memory models. In this paper we discuss the TSO constraints and a method to encode them as the first-order logic formula.

3.2 Encoding Coherence Constraints

For each memory access M, there exists an order variable O. The values of the order variables determine the causal order for the memory operations. Lines `21-33` in Algorithm 1 presents the algorithm for encoding the coherence constraints. To encode coherence constraints, for each memory access M, the algorithm specifies the set of all memory operations that access the same location and can potentially be ordered immediately after M (which requires that their old values equals the new value of M). Special care is taken to account for the possibility that an access could be the last access to a memory location.

In the example shown in Figure 1(right), all accesses are to the same location `X`, and therefore only coherence constraints need to be satisfied to derive a valid TSO order. X_4 is the only access that can immediately follow X_1, because only X_4 has the old value that matches the new value of X_1. X_4 could be followed by either X_2 or X_5, which leads to the possibility that there could be multiple solutions for a given execution trace. For this example, both $X_1 \rightarrow X_4 \rightarrow X_2 \rightarrow X_3 \rightarrow X_5 \rightarrow X_6$ and $X_1 \rightarrow X_4 \rightarrow X_5 \rightarrow X_6 \rightarrow X_2 \rightarrow X_3$ are valid causal orders as they both obey the coherence constraints.

Though the order determined by the offline analysis might be different from the original order observed during recording, the replayed execution is guaranteed to deterministically reproduce the same final state, system output, and reproduce exactly the same sequence of instructions with the same old and new values. For example, in the case of data races, the racy accesses would have the same value, and any erroneous behavior would also be deterministically reproduced. Our symbolic analyzer can be extended to produce all possible solutions, which could also be valuable, as it could reveal many thread interleavings leading to the same erroneous state.

3.3 Encoding Memory Model Constraints for TSO

Memory model constraints specify the legal order between memory accesses to different locations. Relaxed memory models relax two types of constraints [3,5]. One is how they relax the processor-local *Instruction Reordering* axiom. Sequential consistency has the strictest requirement. It requires that program order between the memory operations of a processor is satisfied in the global total order observed between memory accesses executed by all the processors. Second is how they relax the *Store Atomicity* axiom. That is, either all or none of the processors see a store's value. SC requires store atomicity.

The TSO model is widely used in the SPARC [35] and is also similar to the x86 memory model [19, 20]. It relaxes the SC memory order constraints between memory accesses executed in a processor core as follows:

- **Instruction Reordering:** A processor may re-order a load before a store if they access different locations.
- **Store Atomicity:** A store's value may be made visible to a following local load in the same processor before it is made visible to remote processors.

Lines `35-45` in Algorithm 1 describes how we encode these memory ordering constraints. Our algorithm encodes these constraints using the order variables O of memory operations. Under SC, order of each memory operation in a processor P is constrained to be greater than the order of all the earlier memory operations that appeared in the program order in P. We relax this constraint for TSO to account for the above two relaxations (line `40`).

3.3.1 Allowing Relaxed Instruction Reordering

Instruction re-ordering relaxation in TSO allows a processor to retire a store to a local store buffer and allows following (performance critical) loads to execute. The execution shown on the left in Figure 2 is not valid under SC but is a valid TSO execution due to the relaxed instruction ordering requirement. Under SC, either Y_2 or X_4 should be the last memory operation in any valid total order. But in this example, both loads Y_2 and X_4 are executed before the stores X_1 and Y_3 respectively, which leads to a non-SC order.

If a load can be re-ordered above a store in a processor, the ordering requirement between them is not specified in the first-order logic formula. The first clause in line `40` in Algorithm 1 checks whether the ordering between a store and a load can be relaxed, and if so no ordering constraint between those two instructions will be enforced (Line `42`).

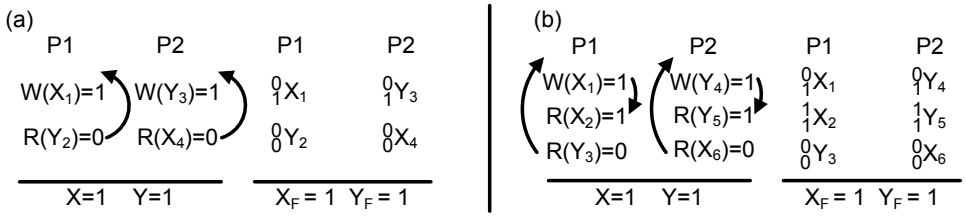

Figure 2: Two example TSO executions and their replayed memory traces with old/new values

3.3.2 Allowing Relaxed Store Atomicity

Store atomicity relaxation allows a processor to read its own store's value early. That is, a processor can forward the value of a store in the store buffer to a following load to the same location. Thereby, if a store results in a cache miss, a processor need not wait for it to resolve, and instead forward that store's value to a later load accessing the same location.

Relaxed store atomicity requirement can be accounted for in our memory model constraints. Under TSO, a load-to-load program order constraint generally cannot be broken. However, to accommodate relaxed store atomicity constraint, we make an important observation that loads can be allowed to reorder with respect to an older load that read its local store's value. Thus, if we can determine all loads that read its local store's value offline, then we can simply relax the ordering requirement between those loads and the loads that follow them in the program order. To identify the loads that read a local store's value offline, we log the memory count of the load that hit in the store buffer during recording.

The example shown on the right in Figure 2 is not valid under SC as it violates store atomicity but is valid under TSO. Under TSO, while the stores X_1 and Y_4 are temporarily held in P_1's and P_2's store-buffer respectively, the loads X_2 and Y_5 can read the value `1` written by locally buffered stores. Then, before the stores' new values become visible to the other processor, loads Y_3 and X_6 can read the old value `0` from their locally cached copies. Effectively, the load-load ordering between $X_2 \rightarrow Y_3$ and $Y_5 \rightarrow X_6$ appear to be relaxed.

Thus, we accommodate TSO's relaxed store atomicity constraint by allowing loads to be re-ordered with respect to older loads that resulted in store buffer hits during offline analysis. The second clause in line `40` in Algorithm 1 checks for the condition when a load-load order can be relaxed. The offline analysis should ensure that no remote load/store accesses are interleaved between the load that caused a store buffer hit and the previous store to the same location. This is taken care of by ensuring that, if a load results in a store buffer hit, its preceding store's immediate follower set (IFS) contains only the load that resulted in the store buffer hit (Line `11`).

Consider again the example trace in Figure 2(b). Our analyzer would relax the program order constraints $X_2 \rightarrow Y_3$ and $Y_5 \rightarrow X_6$ because X_2 and Y_5 are store buffer hits. This would allow our analysis to produce a valid causal order under TSO for this example: $Y_3 \rightarrow X_6 \rightarrow X_1 \rightarrow X_2 \rightarrow Y_4 \rightarrow Y_5$.

While relaxing the above constraints, we also specify that all memory accesses following a fence or a lock-prefixed memory operation should obey the program order.

4. Bounding Search Space

Section 3 described a way to encode first-order logic formula for determining the causal order between shared accesses under TSO. During offline analysis, a solution for the formula is found using an SMT solver [16]. However, it is impractical for an SMT solver to find a satisfiable solution for unbounded number memory accesses. We present a solution to bound the search space by logging hints that allows our offline analyzer to partition a multi-processor execution into smaller bounded intervals, and analyze each interval separately.

To bound the search space we log barrier-like hints called *Strata* at regular intervals [28]. Each processor keeps track of the length of interval by counting the number of cycles elapsed since the last Stratum log. Once a predetermined threshold is reached, all the processors simultaneously record their current memory counts. A memory count is the number of memory operations *committed* by a processor. The program execution between two Strata hints is referred to as a *Strata region*.

Each processor logs its memory counts at the same instant of time. Under SC, memory counts logged at a particular time t provide a barrier-like happens-before relation between all memory accesses committed before t and the accesses committed after t. Thus, memory accesses in different Strata regions are totally ordered. Therefore, an SMT solver can solve one Strata region at a time, starting with the last Strata region and final state. Later, solutions found for all the regions are concatenated based on the total order for Strata.

The above approach, however, is not sufficient for recording execution under relaxed consistency models and out-of-order execution. We discuss how we can ensure the correctness of the happens-before relations specified by the count of committed memory operations.

4.1 Pending Stores in Store Buffer

For clarity, we distinguish between three states of a memory access' execution: (a) a load access is said to have *executed* if it has read the value, (b) a memory access is said to have *committed* when it is committed in-order and its entry removed from the Re-Order Buffer (ROB) and (c) a store access is said to have *performed* when its value is written to the cache block (made visible to remote processors) and its entry removed from the store buffer.

Stores committed from the ROB, but not yet removed from the store buffer could violate the happens-before specified by the Strata hints. Consider the example in Figure 3(a), which shows the same trace in Figure 2(b). Ignore the dashed box for now and let us assume the following state: stores X_1 and Y_4 are committed but not yet performed (temporarily buffered in the store-buffer of their respective processors), loads X_2 and Y_5 are committed, and loads Y_3 and X_6 have only been executed but have not been committed yet. Assume a Stratum is logged at this state. Each processor logs that they have committed two memory operations. This Stratum would provide happens-before relation $Y_4 \rightarrow Y_3$, but in reality load Y_3 executed before the store Y_4 was made visible to $P1$. Similarly, an incorrect happens-before order $X_1 \rightarrow X_6$ would be enforced by the offline analyzer. As a result, it will be impossible to find a satisfiable solution for the second Strata region containing $\{Y_3, X_6\}$ as they would conflicts with the final state.

We solve this problem by logging the number of in-flight stores ($IStore$) in addition to the number of committed instructions as Strata hints. Since the stores are retired in-order from the store

Algorithm 1 ENCODING_ALGORITHM(STRATAREGION E, FINALSTATE F)

/*
Given: Memory events $E = \{e_1, e_2, \ldots, e_{|E|}\}$ and Final State F of a Strata region
Goal : Find a causal order between memory events satisfying (1) uniqueness constraints $\mathcal{C}_U$,
(2) coherence constraints $\mathcal{C}_H$, and (3) memory model constraints $\mathcal{C}_M$
*/
let $E|_p$ be a set of all memory accesses in processor p
let O_i be an event order variable of memory access e_i
let $e_i.loc$ be the memory location of e_i
let $e_i.type$ be the access type (load or store) of e_i
let $e_i.succ$ be the memory access to $e_i.loc$ in the same processor, following e_i in program order
let $e_i.sbh$ specifies if e_i resulted in a store buffer hit
let $e_i.IFS$ (Immediate Follower Set) be the set of memory accesses which can immediately follow e_i. It contains only $e_i.succ$ if $e_i.succ.sbh$ is a hit. Otherwise, it contains $e_i.succ$ and remote memory accesses to $e_i.loc$, provided their old values are same as e_i's new value
let $e_i.IntS$ (Interference Set) be the set of memory accesses including $e_i.succ$ and remote accesses to $e_i.loc$
let $e_i.LAST$ be the set containing a memory access if it is the last access to $e_i.loc$ in a thread and its new value is same as final state of $e_i.loc$ in F

/* Uniqueness Constraints */
$\mathcal{C}_U$=true;
for all pairs of memory accesses $(e_i, e_j) \in E \times E$ where $i \neq j$ **do**
  $\mathcal{C}_U = \mathcal{C}_U \wedge (O_i \neq O_j)$;
end for

/* Coherence Constraints */
$\mathcal{C}_H = true$;
for all memory accesses $e_i \in E$ **do**
  for all $e_j \in e_i.IFS$ **do**
    //order one immediate follower e_j, prevent other accesses to $e_i.loc$ from being scheduled between e_i and e_j
    $\mathcal{C}_i = \mathcal{C}_i \vee ((O_i < O_j) \wedge \bigwedge_{e_k \in (e_i.IntS - \{e_j\})}((O_k < O_i) \vee (O_j < O_k)))$;
  end for
  if $e_i \in e_i.LAST$ **then**
    //schedule e_i to be the last memory access to $e_i.loc$ and prevent other accesses in $e_i.LAST$ from being the last access
    $\mathcal{C}_i = \mathcal{C}_i \vee (\bigwedge_{e_k \in (e_i.LAST - \{e_i\})}(O_k < O_i)$;
  end if
  $\mathcal{C}_H = \mathcal{C}_H \wedge \mathcal{C}_i$;
end for

/* Memory Model Constraints */
$\mathcal{C}_M$=true;
for all $E|_p : \langle e_{p_1}, e_{p_2}, \ldots, e_{p_k} \rangle \subseteq E$ **do**
  for $i = p_k; i > p_1; i--$ **do**
    for $j = i - 1; j \geq p_1; j--$ **do**
      if $\neg(e_i.type = load \wedge e_j.type = store \wedge e_i.loc \neq e_j.loc) \wedge$ //store-to-load reordering
        $\neg(e_i.type = load \wedge e_j.type = load \wedge e_i.loc \neq e_j.loc \wedge e_j.sbh = hit)$ //store atomicity violation
      then
        $\mathcal{C}_M = \mathcal{C}_M \wedge (O_j < O_i)$;
      end if
    end for
  end for
end for

/* The SMT solver should find a solution that satisfies all the above constraints */
$\mathcal{C}_{FINAL} = \mathcal{C}_U \wedge \mathcal{C}_M \wedge \mathcal{C}_H$

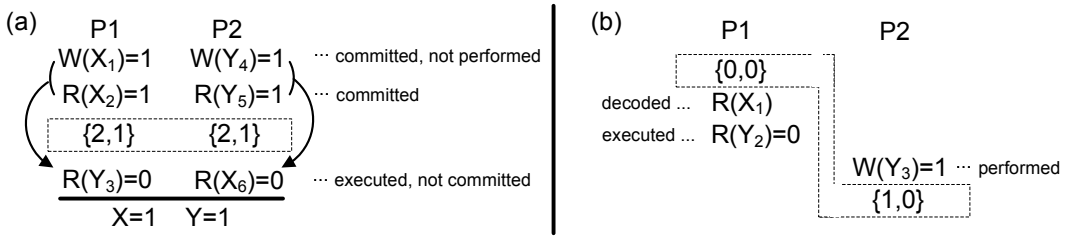

Figure 3: Two examples of recording Strata Hints under TSO

buffer, the offline analyzer can determine that the last $IStore$ stores in a thread before the Stratum log were pending in the store buffer. Using this information, while constructing the Strata regions, the offline analyzer moves the stores pending in the store buffer and its dependent loads (loads that read their value from the store buffer) to the following Strata region. Then they are analyzed with the memory accesses in that Strata region.

For the example in Figure 3(a), each processor logs both the number of committed instructions, which is two, and the number of in-fight stores, which is one. The Strata is represented as a dashed box, and the tuple inside the box shows the logged information in each processor. During offline analysis, while creating Strata regions, in-fight stores X_1 and its dependent load X_2 would be moved to the second region as the arrows indicates. Also, Y_3 and Y_4 would be moved to the second region. With this modification, now the SMT solver would be able to correctly analyze $\{X_1,X_2,Y_3,Y_4,Y_5,X_6\}$ together and arrive at valid TSO-compliant causal order.

4.2 In-flight Loads in Out-of-Order Execution

Most modern processor implementations have speculation support for breaking load-to-load memory ordering constraints to efficiently support TSO [17]. They execute a load out-of-order, and then re-execute them on commit to check if the out-of-order speculative execution was valid or not. The check would fail if there was a remote store that modified the value before the load commits. When a check fails, the load and its dependent operations are re-executed. However, recording Strata using committed memory counts is still sufficient even in the presence of out-of-order speculation.

Figure 3(b) presents an example. Say, the load Y_2 executes out-of-order returning a value of 0, but remains uncommitted. Then, the store Y_3 in P_2 executes, commits and retires from the store buffer by writing a value of 1 to memory. If Strata is created at this moment, then the loads X_1 and Y_2 in P_1 would be considered as part of the second Strata region, because those loads have not committed yet, whereas the store would be considered as part of the first Strata region. This would be an incorrect happens-before relation. However, before committing Y_2, the processor would re-execute the load and find that its value has changed, which would trigger a misspeculation recovery.

4.3 Bounding Search Space Effectively Using B-bound

Processors can determine the end of a Strata region in many ways. The simplest approach would be for each processor to count the number of processor cycles and determine the end of an interval when a threshold number of cycles had elapsed. This requires no additional communication between processors. However, the interval size does not account for the degree of communication between concurrently executing threads which is a critical factor that determines the offline analysis time. An adaptive scheme that logs adjusts Strata region size based on the amount of inter-processor communication is preferable.

We evaluate two approaches that are aware of the degree of communication between processors. One is called **d**owngrade bound (d-bound). In d-bound, each processor counts the number of invalidated or downgraded cache blocks in an interval. If any processor observes downgrades more than a pre-configured threshold, it asks all the other processors to log a Stratum hint. Since the number of downgrades implicitly capture the amount of sharing, we expect the number of shared accesses to be analyzed across Strata regions to be similar. However, d-bound requires changes to the coherence mechanism as it requires additional inter-processor communication to create Strata.

We also evaluate a second approach that is suitable for a snoop-based architecture **b**roadcast-bound (**b**-bound). In snoop-based architecture, each processor snoops the coherence messages broadcasted on the bus. We leverage this property to determine the Strata interval length. Each processor simply counts the number of broadcasted messages and when a threshold is reached a Stratum is logged. Thus, b-bound does not require additional communication between processors in a snoop-based architecture, while it can also adapt the frequency of Strata logs according to the degree of communication between concurrent threads.

5. Reducing Offline Analysis Cost Using Cache Hit Filtering

While analyzing a Strata region, causal order for many memory operations can be trivially determined and therefore filtered out from the time consuming SMT constraint analysis. First, all accesses to a location that was accessed in only one processor in a Strata region can be filtered. Second, all accesses to a location that was only read within a Strata region can be filtered. This is because any causal order between these eliminated accesses is valid within a Strata region. Filtering unnecessary memory operations can significantly reduce offline analysis time. In this section, we propose an additional filtering method called *cache hit filtering* (CHF).

Cache hit filtering is based on our observation that memory operations in between a cache-miss and its last-cache-hit (a hit before losing the read or write permission) to a location can be filtered from the offline analysis. Our recorder logs only cache blocks fetched on a cache miss, and so after replaying each thread and obtaining its memory trace, the offline analyzer can determine which memory accesses had resulted in cache misses during recording. Using this cache hit and miss information, it is also trivial to determine last-cache-hits.

Our offline analysis filters out all cache hits except last-cache-hits. Following constraints (also illustrated in Figure 4) are added to the first-order-logic formula produced by Algorithm 1.

- Remote reads and writes cannot be interleaved between write-miss and a consecutive write-hit.
- Remote writes cannot be interleaved between {write-miss, read-miss, or write-hit} and a consecutive read-hit.

The SMT solver then finds a valid causal order among only un-

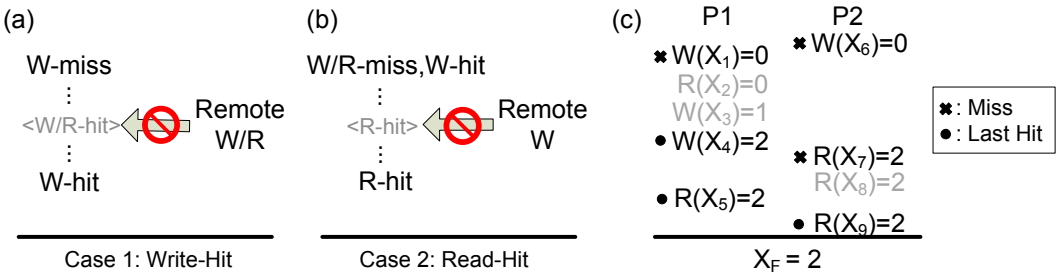

Figure 4: (a) Write-hit Property, (b) Read-hit Property, and (c) Cache Hit Filtering Example

filtered memory operations. The order for the filtered operations are inferred trivially according to the program order.

Figure 4(c) shows an example for the cache hit filtering optimization. Memory operations marked with crosses are cache misses. Memory operations marked with solid dots are last read/write cache hits. Rest of the memory operations in gray are the memory operations eliminated by the cache hit filtering optimization. Following are the constraints added to correctly support this optimization. Write X_6 and reads X_7, X_9 cannot be interleaved between X_1 (write-miss) and X_4 (write-hit). Remote write X_6 cannot be interleaved between X_4 (write-hit) and X_5 (read-hit). Note that remote reads such as X_7 and X_9 are allowed to be scheduled between X_4 and X_5.

5.1 Implications of Cache Hit Filtering

In addition to enforcing the above additional constraints, cache hit filtering also requires several modifications in our offline analysis. First, cache miss/hit information is available only at the cache block granularity. Therefore, local and read-only accesses also need to be determined at the block granularity, and not the word granularity. Otherwise, local and read-only filtering may incorrectly remove memory operations that are cache-misses or last-cache-hits which need to be preserved for enforcing cache hit filtering constraints. This may reduce the effectiveness of local and read-only filtering optimizations due to false sharing at the block granularity. However, we expect that cache hit filtering would effectively compensate for this loss opportunity.

Second, memory dependencies should also be determined at the block granularity. That is, SMT solver should consider block address to determine if two memory accesses are aliased or not. Similarly, old and new value comparison should also be performed at the block granularity. This could reduce the aliasing between values of loads and stores, and thereby reduce the search space and offline analysis time.

Third, filtering out write hits could lead to a mismatch between old and new values of unfiltered accesses. We resolve this by patching up the old and new values at the cache block granularity before we feed the filtered traces into the SMT solver.

Finally, a Stratum log may be created between a cache miss and a cache hit. This implies that a the first access to a location in a Strata region is a cache hit. SMT solver takes care of such tricky special boundary cases. For instance, if the first access to a location in a thread in a write-hit, then none of the remote accesses to that location would be allowed to be scheduled before the write-hit.

6. Results

In this section, we first evaluate the size of cache miss logs for the load-based input logging scheme. Second, we measure the Strata log size and offline analysis overhead for the TSO model and compare them to the sequentially consistent (SC) model. For SC, we also compare the sizes of Strata logs to the precise memory race logs in ReRun [18]. Third, we evaluate the effectiveness of cache hit filtering and b-bound optimization in reducing the Strata log size and offline analysis overhead. Finally, we perform sensitivity studies on different b-bound thresholds and analyze the scalability of our solution across different number of processor cores.

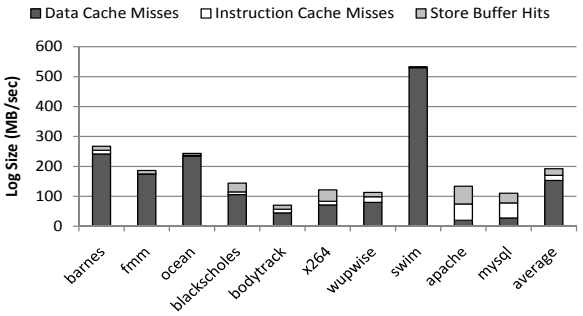

Figure 5: Cache miss and store buffer hit log size

6.1 Evaluation Methodology

Our simulation framework is based on Simics [25] for full system functional simulation and modified FeS2 [1] for cycle-accurate TSO simulation. We model 2, 4, 8, and 16 cores, each with a 32 KB private L1 cache (32-byte block, 4-way associative, 3-cycle latency) and a shared L2 cache (64-byte block, 8-way associative, 30-cycle latency). We model the MESI coherence protocol and a store buffer (32 entry FIFO, 8-byte granularity). We also model speculation support for breaking load-to-load memory ordering constraints to efficiently support TSO [17].

We use four sets of benchmarks: SPLASH-2 [38], PARSEC 2.0 [11], SPECcomp [36], and server applications. We evaluate our system with `barnes`, `fmm`, and `ocean` from SPLASH-2, `blackscholes`, `bodytrack`, and `x264` from PARSEC 2.0, `wupwise`, and `swim` from SPECcomp, and two server applications `Apache` and `MySQL`. All applications are configured to have the same number of worker threads as the number of cores. We fast-forward up to a point where all the threads are spawned and the program starts its main computation (e.g. up to the second barrier synchronization point or OMP parallelization point). Then, we collect the multi-threaded workload traces for 500 million instructions. For Apache, we use SURGE [7] to generate web requests to a repository of 20,000 files (totaling 480 MB) with 400 concurrent clients. For MySQL, we use SysBench [2] to send concurrent queries to a database containing one million records. We tested OLTP mode with 16 client threads. Except the scalability results, all other results are collected for 8-core configurations. Finally, for offline symbolic analysis, we used the Yices SMT solver [16].

6.2 Cache Miss and Store-buffer Hit Log Size

We first measure the size of cache miss and store buffer hit logs collected during recording (Figure 5). Our recording system logs cache blocks fetched on cache misses to implicitly capture program

input and values of loads dependent on remote stores. We also record instruction cache misses to support self-modifying code [22]. A cache miss log record contains cache block data (32 bytes) and a count of the number of memory instructions executed between two logs (2 bytes). Memory counts of load instructions that hit in the store buffer is logged to handle violations of store atomicity under TSO (Section 3.3.2).

On average, cache miss and store buffer log requires 192MB for one second of 8-threaded execution. For scientific benchmarks, data cache misses constitute a large portion of the total log size. On average, about 2.45% of load instructions read their values from the store buffer.

6.3 Strata Log Size and Offline Analysis Time for TSO

Figure 6 compares Strata log size and offline analysis time between the SC and TSO models. For this experiment, we apply cache hit filtering and b-bound optimization with a threshold of 10. Sensitivity results on these optimizations are presented in the later two sections.

We logged 4 bytes to record a memory count in a processor core while logging a Stratum, but this could be optimized by recording only the different in memory counts in a processor between two Strata logs. In addition, we logged the number of in-flight stores to support TSO model (Section 4). For a 32 entry store buffer, we need 5 bits to log the number of in-flight stores. On average, we need 1025KB for SC and 1185KB for TSO (15% increase) to record Strata hints for one second of program execution on an 8-core configuration.

Figure 6(b) shows that offline analysis time for TSO surprisingly decreases by 30% when compared to that of SC. Relaxing constraints could have positive or negative effect on offline analysis time. Under TSO, search space increases. But when compared to SC, in TSO, it is possible that the proportion of legal solutions to the infeasible solutions that the SMT might explore increases. If so, then the offline analysis time could be better than SC. The variation in analysis time for different applications in Figure 6(b) is a consequence of this.

On average, it takes 260 seconds to analyze one second of an 8-threaded execution under TSO. `swim` is our worst case which takes 745 seconds. This offline analysis need to be performed only once. Once shared memory dependencies are resolved, execution can be replayed with little overhead. Furthermore, we could reduce the analysis cost by parallelizing the offline analysis of different Strata regions and also improve our generic Yices solver by customizing it specifically for our problem.

For the SC model, we also compared the sizes of Strata logs to the precise race logs in one of the state-of-the-art hardware recorders, called Rerun [18]. The results show that we can save about 10 times memory race log size when compared to ReRun. However, our program input log could be larger than that of a copy-on-write based program input recorder assumed by ReRun (discussed in Section 6.2).

6.4 Performance

The performance overhead would be similar to Replay-SMT [22]. Similar to Replay-SMT we record cache misses and log Strata at semi-regular intervals. In addition, we log the number of pending stores as part of Strata hints and memory counts of loads that read from the store buffer to support TSO. Since the recording overhead is dominated by cache miss logging, the recording performance would be similar to Replay-SMT (less than 1% on average).

6.5 Effects of Cache Hit Filtering and B-Bound Optimization

In this section, we evaluate the effectiveness of cache hit filtering (CHF) and also compare d-bound to b-bound optimization. We compare three configurations under SC: SC(d-bound), SC+CHF(d-bound), and SC+CHF(b-bound). The first SC(d10.c10000) configuration, is for the technique used in Replay-SMT [22], where each processor creates Strata either after more than ten cache blocks have been **d**owngraded (d-bound) or after 10000 **c**ycles have elapsed. This is the best configuration of Replay-SMT and we consider it as the baseline for our evaluation. The second configuration, SC+CHF (d10.c10000), employs our cache hit filtering optimization over the earlier design. The third configuration, SC+CHF(b10), is a design with cache hit filtering and with b-bound. Each core creates Strata if there have been more than ten coherence **b**roadcast messages (b-bound). Without cache hit filtering, b-bound optimization alone is not effective because there could be a Strata region with high hit rate and less sharing, leading to a large number of unfiltered accesses.

Figure 7 shows the result of filtering local, read-only, and cache-hit memory operations. For the baseline SC(d10.c10000) configuration we need to analyze less than 0.4% of total memory operations. Applying cache hit filtering (SC+CHF(d10.c10000)) reduces the effectiveness of local or read-only filters due to false sharing (we have to analyze at the block granularity if we employ cache hit filtering). However, cache-hit filtering effectively compensates for the loss. Figure 8 shows the average and maximum number of unfiltered accesses per Strata region. This shows that the cache-hit filtering is effective especially when we combine it with the b-bound optimization represented as SC+CHF(b10). When the two optimizations are applied together the average number of memory operations per Strata reduces by nearly 40% compared to the baseline (SC(d10.c10000)). Programs like `ocean` with a high cache miss rate does not benefit from CHF optimization.

Also, the maximum number of operations per Strata region, which determines the worst case analysis time, reduces significantly in most cases. Maximum number of memory operations per Strata region is an important measure given that the offline analysis time grows exponentially with the number of memory operations to be analyzed together. The maximum number of memory operations across all Strata regions in all programs is less than 90 after employing CHF and b-bound optimizations. Without those optimizations it was nearly 290 (`wupwise`).

One interesting property of b-bound is that the standard deviation of the maximum number of unfiltered accesses across different benchmarks is a lot smaller than d-bound (16.9 vs 76.6). This indicates that b-bound with CHF is a better application-independent predictor of how large each Strata region should be. Furthermore, b-bound is simpler than d-bound in terms of hardware implementation for a snoop-based architecture, because d-bound requires additional communication among cores but b-bound does not.

Figure 9 shows the result of cache hit filtering and b-bound optimizations on Strata log size and offline analysis overhead. The b-bound optimization reduces the Strata log size by nearly three times. Cache hit filtering does not affect the size of Strata log, because it is an offline filter employed to reduce the number of memory operations that need to be analyzed within a Strata region.

Cache hit filtering reduces offline analysis time by reducing the number of memory operations that need to be analyzed. Also, to support CHF optimization, we perform analysis at the cache block granularity. While this may reduce the effectiveness of local and read-only filters as discussed before, it could reduce the search space by reducing the amount of aliasing between the old

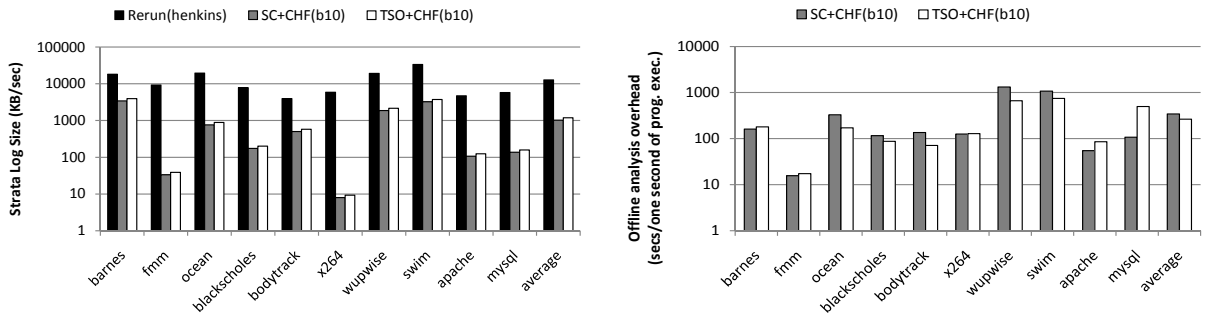

Figure 6: Strata log size and offline analysis overhead under SC and TSO memory models

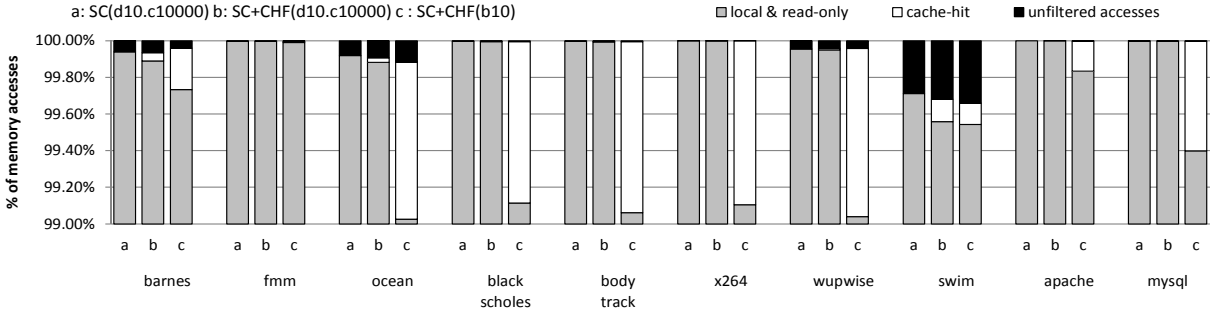

Figure 7: Effectiveness of local, read-only, and cache-hit filtering

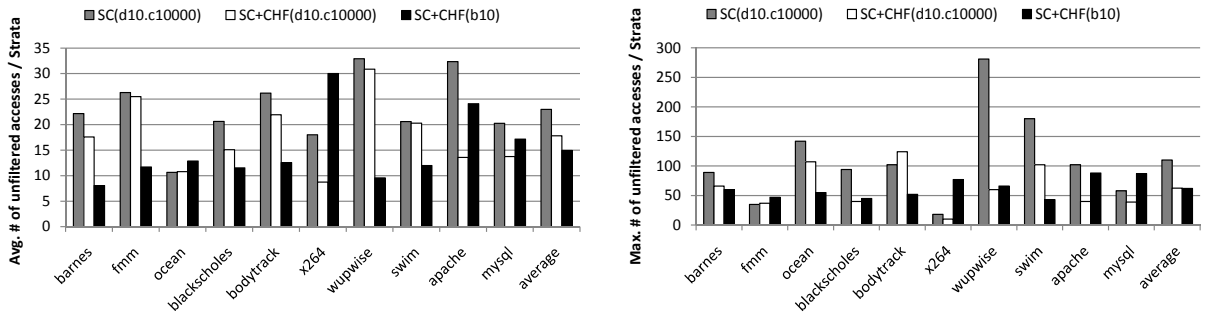

Figure 8: Average and maximum number of unfiltered accesses per Strata region

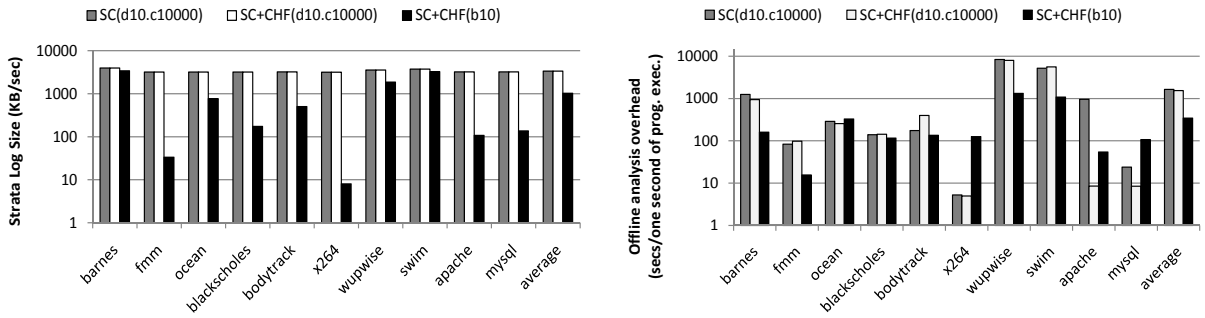

Figure 9: Effectiveness of b-bound and Cache Hit Filtering (CHF) in reducing Strata log size and offline analysis overhead

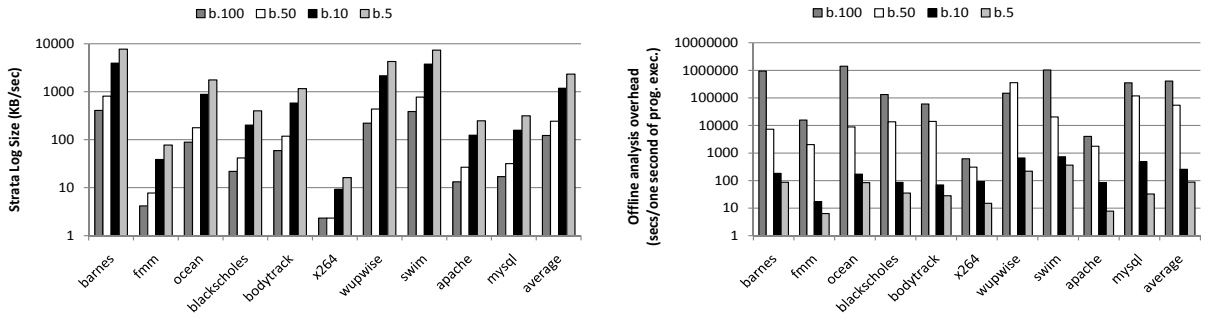

Figure 10: Strata log size and offline analysis overhead for different b-bounds

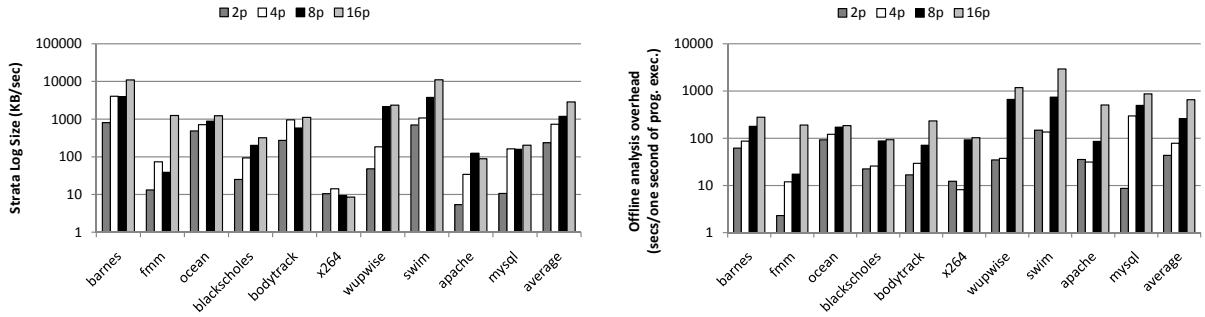

Figure 11: Strata log size and offline analysis overhead for different number of processors

and new values of memory operations which in turn reduces the legal follower set for a memory operation. On average, compared to SC(d10.c10000), SC+CHF(d10.c10000) shows 6% of improvement on offline analysis overhead. However, together with b-bound, SC+CHF(b10) shows impressive improvements: 3x less Strata log size and 4.8x less offline analysis time on average.

6.6 Sensitivity Studies

In this section, we present sensitivity studies varying b-bounds to illustrate the tradeoff between Strata log size versus versus offline analysis time. Figure 10 shows that on average, Strata log size increase approximately linearly as the b-bound decreases from 100 to 5: 122KB/sec, 242KB/sec, 1185KB/sec, and 2333KB/sec, respectively. On the other hand, it takes 54261 seconds with b-bound of 50 to analyze one second of 8-threaded execution, whereas it only takes 258 seconds with b-bound of 10, which is about 210 times of improvement. The user or the operating system can specify the bound based on the trade-off that one is willing to pay.

We also present scalability results with different number of processors for a constant b-bound of 10. Figure 11(a) shows that on average Strata log size increases by 3.1x, 1.6x, and 2.4x as the number of cores doubles from 2 to 16. Similarly, Figure 11(b) shows that offline analysis cost increases by 1.8x, 3.3x, and 2.5x respectively.

7. Related Work

Over the last few decades, there have been a number of proposals for developing replay systems from both software and hardware communities. However, most of them are limited to the SC memory model due to its simplicity. Supporting relaxed consistency memory model remains to be a challenging problem. To the best of our knowledge, only a few replay systems support replay under a relaxed memory model. RTR [40] is the first to propose a solution for TSO. It proposed to dynamically detect loads that violate SC while executing on a TSO processor and then explicitly record their values. Detecting SC violation requires monitoring if a load memory location is modified between the time the load accesses the location and the time when all the preceding memory accesses have finished. In addition, RTR requires hardware support for detecting and logging shared-memory dependencies precisely. Instead, we propose to record Strata along with cache miss information. Strata require no communication between processor cores or changes to coherence mechanism to record them. Instead, at fixed periodic intervals each processor core records their memory counts and outstanding stores in the store buffer. Thus the required modifications are "local" to a processor core. We believe that a local solution is simpler than a solution that require changes to the coherence mechanism, as coherence protocol design and verification is hard. Effectively, our solution reduces hardware complexity of the recorder by relaxing the precision required in recording shared-memory dependencies.

ReRun [18] also uses the same technique as RTR to detect potentially SC-violating loads. LReplay [13] records pending period information and supports some relaxed models such as the Godson-3 consistency by logging load instructions violating SC. However, it only considers store atomic multiprocessor systems, which is not the case for the TSO memory model in x86 processors.

Recent hardware-assisted replay systems [13, 18, 26–28, 37, 39, 40] show that recording shared memory dependencies with hardware support only incur less than 1% of performance overhead. Therefore, they have focused on reducing the hardware cost to detect and log shared memory dependencies and also reducing the log size for longer recording. Unlike other hardware-assisted systems, Replay-SMT [22], which we improve upon in this paper, does not record shared memory dependencies during recording. Instead, it determines shared memory dependencies offline. As a result, Replay-SMT requires much simpler hardware support.

Recent work on software-only replay systems achieved relatively low logging overhead by not eagerly recording shared-memory dependencies and relaxing the fidelity level of replay. ODR [4], PRES [32], and Respec [23] record less information than necessary to reproduce the same interleaving between threads. Though the overhead of the solutions are significantly better than earlier software solutions, they are still on the order of 20-30% or more. Whereas, processor based solutions including ours incur negligible performance cost, which may allow us to monitor production runs. Also, ODR and Respec solutions are output-deterministic in that they only guarantee that a replayed execution's output is same as that of original execution. Deterministic guarantee of our system is stronger because we reproduce the exact same sequence and control flow for each thread's execution along with their input and output values. Respec [23] does not support offline replay, incurs at least 2x throughput overhead, and also cannot record and reproduce a non-SC execution. PRES [32] logs hints such as happens-before synchronization order, path trace, or share-memory order and repeated replays till it produces an execution that matches the trace logged. However, without a bounded and guided search, replay is not guaranteed to find a valid order. Especially, non-SC execution may not be reproduced through repeated replays. ODR [4] also logs hints such as system input trace and path traces, and employs a symbolic analysis. ODR models only a hypothetical lock-order consistency, which is weaker than TSO. As a result, ODR may find an impractical schedule on race, which requires that some memory operations in the derived schedule need to be flipped to understand racy behavior correctly. Also, offline search in ODR is over the entire execution and is not bounded. The authors mention that it may not complete for some executions. Whereas, we bound the search by logging Strata hints during recording. It is based on a technique that estimates the number of shared-memory dependencies.

Recent work has produced solutions for ensuring that every multithreaded execution would follow the same schedule for a given program input [6, 8, 9, 14, 31]. Thus, causal order need not be logged, and only program input need to be recorded to reproduce an execution. Also, guaranteeing determinism at the language level [8, 9] could improve programmability. However, these solutions require additional runtime support to enforce restricted thread schedules which may also reduce performance [6, 9, 14]. Solutions that guarantee determinism at the language level may also impose restrictions on programmers [8] and the type of parallelism that can be exploited [9].

8. Conclusion

Deterministic replay could significantly help programmers understand a multi-threaded program execution. In this paper, we take an important step towards supporting replay of executions under a relaxed consistency model. We built upon a cache-miss logging approach which was sufficient to deterministically replay each thread. We employed a new offline analysis to determine the causal order between shared operations under the TSO model. We also discussed complexity-effective solutions for logging Strata hints that allowed us to bound the offline analysis, and cache-hit filtering optimization to reduce the number of memory operations that need to be analyzed to determine the causal order. These optimizations reduced the Strata log size by 3x and offline analysis time by 4.8x on average. We showed that offline analysis could in fact be 30% more efficient for a TSO execution that an SC execution.

References

[1] Fes2 simulator. http://fes2.cs.uiuc.edu.

[2] Sysbench. http://sysbench.sourceforge.net.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[4] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, October 2009.

[5] A. Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, 2006.

[6] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI '10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[7] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *In Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1998.

[8] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *OSDI '10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[10] S. Bhansali, W. Chen, S. de Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, June 2006.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[12] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[13] Y. Chen, W. Hu, T. Chen, and R. Wu. Lreplay: A pending period based deterministic replay scheme. In *ISCA*, Saint-Malo, France, June 2010.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2009. ACM.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symposium on Operating System Design and Implementation*, 2002.

[16] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[18] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *International*

Symposium on Computer Architecture, 2008.

[19] Intel Corporation. Intel 64 architectures memory ordering white paper. Technical report, 2007.

[20] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual (rev.30). Technical report, 2009.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX 2005 Annual Technical Conference*, 2005.

[22] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline Symbolic Analysis for Multi-Processor Execution Replay. In *International Symposium on Microarchitecture (MICRO)*, New York, NY, December 2009.

[23] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, PA, March 2010.

[24] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[25] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[26] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *International Symposium on Computer Architecture*, 2008.

[27] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS*, pages 73–84, 2009.

[28] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2006.

[29] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, June 2006.

[30] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd Annual International Symposium on Computer Architecture*, June 2005.

[31] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108, New York, NY, USA, 2009. ACM.

[32] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, October 2009.

[33] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[34] C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *IISWC*, pages 173–182, 2008.

[35] C. SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[36] SPEC. Standard performance evaluation corporation - http://www.spec.org.

[37] G. Voskuilen, F. Ahmad, and T. Vijaykumar. Timetraveler: exploiting acyclic races for optimizing memory race recording. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 198–209, New York, NY, USA, 2010. ACM.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[39] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th ISCA*, San Diego, CA, 2003.

[40] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS-XII*, pages 49–60, 2006.