Offline Symbolic Analysis for Multi-Processor Execution Replay

Dongyoon Lee   Satish Narayanasamy   Mahmoud Said   Zijiang (James) Yang

University of Michigan, Ann Arbor   Western Michigan University

{dongyoon,nsatish}@umich.edu   {mahmoud.said,zijiang.yang}@wmich.edu

Abstract

Ability to replay a program's execution on a multi-processor system can significantly help parallel programming. To replay a shared-memory multi-threaded program, existing solutions record the program input *(I/O, DMA, etc.) and the* shared-memory dependencies *between threads. Prior processor based record-and-replay solutions are efficient, but they require non-trivial modifications to the coherency protocol and the memory sub-system for recording the shared-memory dependencies.*

In this paper, we propose a processor-based record-and-replay solution that does not require detecting and logging shared-memory dependencies to enable multi-processor replay. It is based on our insight that, a load-based checkpointing scheme that records the program input has sufficient information for deterministically replaying each thread. We propose an offline symbolic analysis algorithm based on a SMT solver that determines the shared-memory dependencies using just the program input logs during replay. In addition to saving log space, the proposed approach significantly reduces the hardware support required for enabling replay.

1. Introduction

Ability to replay a program's execution has a number of applications. Using a record-and-replay system, one can build a time-travel debugger [9]. It helps a programmer debug non-deterministic bugs that are prevalent in multi-threaded programs. Recently, several hardware techniques [24, 19, 8, 14, 15] have been proposed for efficiently recording a program's execution. Using hardware support, a realistic execution of a program can be recorded and analyzed using debugging and dynamic analysis tools [4]. Given the efficiency of these schemes, even production runs can be recorded. In the event of a system crash, it provides programmers with significantly more information than the crash dumps containing just the final program state.

To replay a shared-memory multi-threaded program's execution on a multi-processor system, existing solutions record two types of information at runtime. A program's input and the shared-memory dependencies between its concurrent threads.

A program's input can be recorded by checkpointing the program's initial register and memory state, and then logging all the non-deterministic system events such as I/O, DMA, interrupts, etc. This could be supported in the system software with the help of an efficient copy-on-write checkpointing mechanism in the processor [22, 21]. This approach is system-dependent. Alternatively, a system-independent load-based checkpointing scheme [19, 4] for recording a program's input. This scheme records the initial register state and the values of a subset of load instructions (thereby, it implicitly captures the system input from I/O, DMA, interrupts, etc.). We refer to this approach as load-based checkpointing scheme.

To detect and log all the shared-memory dependencies, a software solution would require monitoring every memory access, which would significantly degrade performance. There has been significant work in the recent years to provide efficient hardware support [24, 16, 25, 8, 14] for logging the shared-memory dependencies. These solutions not only detect and log the shared-memory dependencies, but they also reduce the log size by not logging a dependency if it is transitively implied by an earlier dependency log. However, significant changes need to made to a processor design to accomplish these tasks. For example, the state-of-the-art solution called ReRun [8] tracks the read/write sets in each processor core using bloom filters, a Lamport Scalar Clock [10] in each core, piggy-backs coherence messages with additional information, and detects the shared-memory dependencies by monitoring those messages. The coherence mechanism is one of the most difficult to verify sub-system in a processor. For example, AMD64 processor had 9 design bugs that were related to the multi-processor support [18]. A simpler and efficient recording solution for multi-processor systems will make it feasible for hardware vendors to provide support for replay, which is the focus of this paper.

We propose a record-and-replay solution for multi-threaded programs that does not require the recorder to detect and log shared-memory dependencies at all. We show that if we use a load-based checkpoint mechanism for recording program input [19] (instead of a system-dependent scheme used in prior work), we do not have to record shared-memory dependencies.

For each thread in a program, a load-based checkpoint mechanism records the thread's initial register state and the values of a subset of the load instructions executed by the thread. We observe that this information alone is sufficient for deterministically replaying each thread in isolation, independent of the other threads, and this does not require shared-memory dependencies. Deterministic replay of a thread in isolation gives us the exact same sequence of instructions executed by the thread during recording along with the input/output values of each instruction (input for a memory operation includes its address as well). Thus, we can support replay of each thread in a multi-threaded program's execution on a multi-processor system without having to log its shared-memory dependencies.

However, to debug a multi-threaded program, a programmer would still need to be able to understand the interactions between the threads, which requires shared-memory dependencies during replay. This information, however, can be determined using an offline analysis. Using load-based checkpoint logs we can deterministically replay each thread and determine the sequence of memory operations executed by that thread along with their addresses, and their input/output values. From the core dump, we also know the final state of every memory location. Using this information, we encode the program order and input/output constraints for each memory operation in the form of a satisfiability equation. The solution for this equation is found using a SMT (Satisfiability Modulo Theory) solver called Yices [7] to determine a total order for the memory operations executed across all the threads. The derived order guarantees the same input and output values for all the instructions executed by the program. To bound the time complexity of the SMT solver, processor logs hints during recording. These hints consist of instruction counts logged independently by each processor core at the end of an interval whose length is pre-determined in terms of number of processor cycles elapsed since the end of the last interval.

We discuss hardware support for load-based logging scheme, which essentially requires support for logging the cache misses. In addition, each processor core records its instruction count as a hint in the log after a specified number of processor cycles has elapsed (note, this does not require any communication between the cores). Thus, we are able to significantly reduce the hardware support required for recording a multi-threaded program's execution by avoiding the need for detecting and logging shared-memory dependencies altogether.

The proposed solution has a trade-off. It drastically reduces the hardware support and log space required for recording a program. But, we have to pay the cost for offline analysis once before we can replay the recorded execution. The offline analysis need to be performed only once and not every time a recorded execution is replayed. In other words, our solution increases the time to replay, but not the replay performance itself (therefore, debugging using replay could still be interactive). We believe that this offline analysis cost is an acceptable trade-off, because efficiency matters more for a recorder than for a replayer. Using an efficient hardware recorder, production runs can be monitored. Also, it enables programmers and beta-testers to record a realistic program execution, and analyze them using various dynamic analysis tools such as data race detectors offline. Replayer could be implemented in software. We believe that a simpler hardware solution proposed in this paper could be a good first step that could go a long way in convincing hardware vendors to provide program execution recording support in their next generation processors.

This paper makes the following contributions:

- We propose a new direction for developing a record-and-replay hardware solution that does not require recording support for detecting and logging shared-memory dependencies. This is based on our observation that a load-based program input log is sufficient for deterministically replaying each thread in isolation (independent of the other threads), and that the information obtained from replaying each thread is sufficient for determining the shared-memory dependencies offline.

- We propose a novel symbolic analysis based on a SMT solver to determine the shared-memory dependencies offline from the program input logs and hints recorded at runtime. This is the first work that uses a symbolic analysis to resolve the shared-memory dependencies offline, which reduces the complexity and cost of the recorder.

- The proposed recording solution is both complexity efficient and also results in smaller log sizes. We also show that without any additional hardware support, we can also support relaxed consistency models.

- We analyze the performance of the logger and the offline symbolic analyzer using Apache, SPECjbb2000, SPECfp and Splash benchmarks. We compare the size of the logs containing hints to the state-of-the-art shared-memory dependency logging scheme called ReRun [8].

2. Background and Motivation

Software community has worked on developing replay solutions for over two decades [11]. Recent developments such as ReVirt [6], Flashback [23] and ReTrace [26] can record and replay a program execution on a uni-processor system for less than 10% performance overhead. These solutions just record the program input. The most common solution used to record the program input is a copy-on-write checkpointing scheme that captures the initial memory and register state of an application. In addition, they explicitly detect every non-deterministic system event such as a system call, DMA, I/O, etc., and log its values along with a timestamp. We refer to this approach as *system-dependent* program input logging approach, because an execution recorded using this approach can be replayed only on a system that is same as the one where it was recorded. The disadvantages and the complexity of such an approach are discussed more in detail in [17, 19].

Software vendors such as VMWare, Microsoft and Intel have all invested recently in developing replay solutions [26, 4, 17]. With the advent of multi-cores, parallel programming is going to become mainstream. Replay solutions are especially useful for debugging a multi-threaded program's execution on a multi-processor system, as parallel executions are notoriously difficult to reproduce and understand.

Replaying an execution on a multi-processor system, however, remains a very challenging problem. In addition to logging program input, existing solution require support for logging the shared-memory dependencies. For a software component to detect and log the shared-memory dependencies between the threads, it might have to examine the execution of every memory operation, which would clearly lead to a high performance cost. Several researchers have proposed to solve this problem using hardware support.

Bacon and Goldstein [2] proposed to record all the coherence traffic in a snoopy bus-based multi-processor system. FDR [24], RTR [25], Strata [16], DeLorean [14], and ReRun [8] are all recent developments that improved the hardware design for logging the shared-memory dependencies. They focused on reducing the log size and the amount of hardware states required to detect and log the shared-memory dependencies. Though these solutions have been successful in reducing the cost of hardware real estate, the hardware complexity of these solutions is still high.

To illustrate the complexity of a shared-memory dependency logger, we now discuss two state-of-the-art hardware designs DeLorean [14] and ReRun [8] in some detail. In Section 5, we compare the sizes of our logs to that of ReRun's. DeLorean assumes support for BulkSC [5]. It divides a thread's execution into what are called as *chunks*. The underlying BulkSC mechanism ensures that each chunk is executed atomically. Given this execution environment for a multi-threaded program, each core in DeLorean records the sizes of the chunks executed, and a global arbiter records the total order between all the chunks executed in different cores. Log size could be further reduced at the cost of some performance overhead by constraining the recorded execution by enforcing a pre-defined chunk size and an order between those chunks. DeLorean drastically reduces the log size required for recording the shared-memory dependencies. However, there is a significant hardware cost associated with the design, which includes support for BulkSC [5], arbiter for logging the total order, and support for logging the chunk sizes.

ReRun [8] forms episodes and records their sizes along with a total order between them. The total order is recorded tracking and logging a Lamport Scalar Clock [10] in every core and in every memory bank of the shared-cache. Similar to a chunk, an episode is also a sequence of instructions that appear to be atomic in an execution. However, episodes are constructed differently in ReRun. Before sending an invalidation acknowledgment or a data update message to a request from another core, a core terminates its episode if it finds a read-write or a write-write conflict between the requesting access and one of its past accesses. An episode could also be terminated when a cache block needs to be evicted, because the core would not receive coherence messages for the cache block after it gets evicted. Thus, ReRun ensures that the atomicity property for the episodes is preserved. The history of memory accesses in a core are concisely tracked using two bloom filters, one for the read set and another for the write set.

ReRun is very efficient in terms of log size (about 4 bytes/kilo-instruction) and performance. However, it needs hardware support for creating episodes and logging their sizes along with their total order. For each core, it needs two bloom filters a timestamp register to hold the Lamport Scalar Clock per core, and a memory counter. Each memory bank in the shared-cache also has a timestamp register. The coherence messages are piggy-backed with the timestamps. The hardware complexity in supporting this functionality is clearly significant.

ReRun and DeLorean record shared-memory dependencies, but to support replay, in addition we need system support for recording program input. Capo [15] provided a good abstraction for implementing a record-and-replay system using hardware support. It advocated software system support for recording program input, which requires a copy-on-write checkpoint mechanism and support for logging all the non-deterministic events. But the performance cost of a software-based program input recording approach could lead to about 20% to 40% performance loss as shown in Capo [15]. However, this cost could be potentially reduced using a hardware copy-on-write checkpointing scheme like SafetyNet [22] or ReVive [21].

Hardware vendors are likely to incorporate support for record-and-replay in their next generation processors, if we can arrive at a simpler solution, which is the focus of this paper. We propose a replay solution that does not require a recorder to detect and log the shared-memory dependencies at all. Existing solutions such as Capo [15] use a system-dependent program input logging approach in addition to logging the shared-memory dependencies. Instead, we employ just the load-based program input logging approach used in BugNet [19] (also used in Microsoft's software record-and-replay system iDNA [4]). In the original proposal, BugNet also assumed support for logging shared-memory dependencies. But, we show that BugNet's program input log is sufficient for deterministically replaying each thread in isolation independent of the other threads. To aid debugging, we determine the shared-memory dependencies offline using a symbolic analysis based on Yices SMT solver [7]. We discuss an efficient hardware design for recording the load-based program input checkpoint logs (which essentially consists of just logging the cache block that is fetched on a cache miss). The only additional cost we pay to support replay of a multi-processor execution is that we log hints to bound the complexity of the SMT solver. But, these hints are not based on detecting shared-memory dependencies nor they require changes to the coherence mechanism. Thus, the proposed solution saves shared-memory dependency log size (hint log is about 10x more efficient than ReRun's log [8]), but more importantly, we reduce the complexity

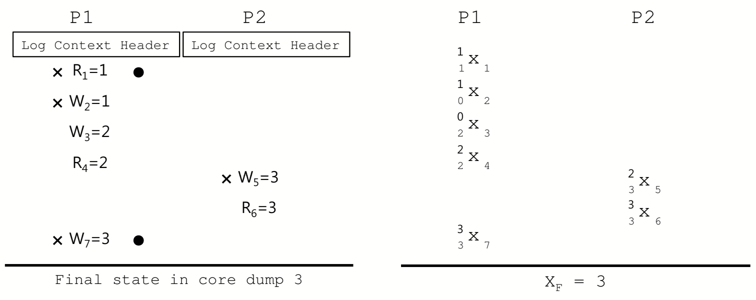


Figure 1. Load-Based Logging Example

of the hardware based record-and-replay system.

3. Load-Based Checkpointing Architecture

The load-based checkpointing scheme was originally proposed in the BugNet architecture [19] as an alternative to a system-dependent logging scheme for recording program input. A system-dependent logging scheme used in most record-and-replay systems includes support for copy-on-write checkpointing to recreate an initial memory and register state, and also provides support for logging non-deterministic events such as I/O, interrupts, DMA, etc along with their timestamps. Apart from BugNet [19], Microsoft's iDNA [4] also uses a load-based checkpointing scheme for recording program input to enable replay of a multi-threaded program on a multi-processor system. Another tool called pinSEL [17, 20] used at Intel also implements a load-based checkpointing scheme using Pin [12] to enable replay of multi-threaded programs. However, these software solutions incur more than ten times performance overhead [4, 20] than the native execution. Also, unlike our proposal, they also record the shared-memory dependencies [20] at runtime to replay the multi-threaded programs. In this paper, we show that this is not necessary. Our goal in this paper is to provide processor support just for load-based checkpointing scheme, and use an offline software analysis to determine the shared-memory dependencies during replay. Though our focus in this paper is an efficient hardware solution, our solution could also help reduce the recording overhead of software tools such as the pinSEL [20].

In this section, we briefly describe our load-based checkpointing scheme. Our scheme is a modified version of BugNet [19], where we extend the original design to support replay of the full system and programs with self-modifying code. We also describe a complexity-effective architecture design to support this scheme efficiently, and finally we present a qualitative comparison to system-dependent logging approach. We also describe its unique property that allows us to replay each thread in a multi-threaded program in isolation without the shared-memory dependencies. Then we describe additional architectural support required for logging hints that helps us bound the complexity of our offline analysis. In the next Section 4 we describe an offline algorithm that determines the shared-memory dependencies from a load-based program input log and a hint log.

3.1. Load-Based Program Input Logging

Let us first consider recording a single-threaded program's execution on a uni-processor system without any system events such as I/O, DMA, context switches or an interrupt. A key insight in the BugNet [19] recording scheme is that to replay an interval of a program's execution, it is sufficient to record the program's initial register state, and then record the values of all the load instructions executed by the program during the interval. The value of a load instruction is recorded along with the instruction count corresponding to the load instruction. Unlike in BugNet[19], we consider instruction read also as a load as it allows us to handle programs with self-modifying code. The instruction count of a memory instruction is the number of instructions that the program had executed since the beginning of the recorded interval. Using this information, a tool like Pin [12] could be used to replay the recorded interval of a program's execution. The replayer essentially simulates the register and the memory states for the program. First, the simulated register states are initialized from the log, which also includes the program counter state. All the memory states are initialized to invalid states. Then, the first instruction specified by the program counter needs to be loaded. Since BugNet treated an instruction read as a load, its machine code can be found in the recorded log. If the instruction is a non-memory operation, it is executed by reading the input values from the register states, and then writing back the output to a simulated register state. If the instruction is a load, its effective address is computed from the input register states. In addition, the load's value is read from the log and the simulated memory state is updated with that value. If the instruction is a store, its input values are read from the register state, its effective address is computed, and the simulated memory state is updated with the store value. Thus, a program is deterministically replayed with exactly the same sequence of instructions along with their input and output values. The register and memory states for the program is also deterministically reproduced.

Recording every load value (including instruction reads) would lead to large log sizes. But, BugNet logs a load value for a load, only if that load is the first memory access to the location that it accesses. The load whose value is recorded is called a *first-load*. The values for a non-first-load need not be recorded. When the replayer wants to execute a non-first-load its memory value can be found in the simulated memory state.

Figure 1 shows a sample execution. Consider just the first four instructions executed by processor `P1`. All these instructions access the same memory location. `R1` is the first-load (with a return value `1`), and therefore it is logged (indicated by the solid dot on the right-side of the instruction). The values of the next three memory operations are not logged as they can be deterministically replayed.

To begin logging a program's execution, the operating system first records the *context header* and turns on logging for the processor core. The header contains the initial register state, a process identifier and the value of the timestamp counter of the processor core. To record first-load logs we need processor support. BugNet [19] used a bit per memory word in the private cache of a processor to determine if that location has been logged for the program or not. We use an even simpler design, where we just record the cache block fetched on a (read or write) cache miss, because any first access to a location would result in a compulsory cache miss. In the case of a write miss, the data recorded for the cache block are the values before executing the write. This might slightly increase the log size as we would log a miss even if the access is a write. In BugNet, a memory location's value need not be logged if the first access is a write, because the write can be deterministically replayed. `W2, W5, and W7` operations (denoted with cross marks) in Figure 1 are write misses and are logged in our design.

The record logged on a cache miss consists of the instruction count of the memory operation that resulted in the cache miss, and the data of the cache block fetched. To further simplify the design, we also choose not to use any local log buffers, but instead directly write-back the cache block to the log space allocated in the L2-cache or in the main memory. Note that any read from an uncacheable memory-mapped location would always be logged as it will not be a cache hit. Thus, non-deterministic input read from system devices such as network cards are correctly captured. Also, `RDTSC` (Read TimeStamp Counter) instruction in the x86 architecture is also treated as a uncacheable load, and its return value is recorded.

3.2. Handling System Events

The previous section assumed a uni-processor system, and also that there are no system events such as that affect a program's execution. We now relax the latter constraint. Unlike BugNet [19], we choose to record the execution of the full system including the operating system code. An interrupt, or a system call, or another program can context switch a thread. But all that the operating system needs to do is terminate the current log by logging the current instruction count for the processor core (so that the replayer would know when to context switch), log a context header for the new program that is about to execute, and let the processor continue to log information about its cache misses. The logging can continue for a processor till the end of a checkpoint interval (note, a checkpoint interval may span multiple context switches). The only other change that we need to make to our earlier design is record the physical address of the cache block when it is logged on a cache miss.

To replay a checkpoint interval, the replayer starts from the first context header and continues to emulate the *physical* memory state of the system. When we find a record for a memory access in the log during replay, the replayer gets the physical address of the memory state that needs to be updated with the value read from the log. When the execution during replay reaches the next context header (determined by comparing the emulated instruction count with the instruction count that was logged on a context switch), the emulated register state is updated with the values from the next context header. Then the replay can proceed normally.

The above approach ensures replay of the entire execution on a processor core for an interval. Using the process identifier logged in the context header, the replayer could provide the programmer with information about which application is replayed at any instant.

Whenever there is a page fault, the operating system starts a new checkpoint interval so that the physical address information recorded for each cache miss is consistent within a checkpoint interval. Thus, information recorded for each checkpoint interval contains sufficient information to replay that interval independent of the other intervals.

3.3. Multi-Processor Replay

We now discuss support for recording a full system execution on a processor with multiple processor cores (which includes a DMA processor). Each processor core has log space allocated to it in the main memory by the operating system. To start recording for a checkpointing interval, the operating system first records the context header for each core, and then lets each core log their cache misses into the private log allocated to it. When a thread on a core is context switched out, the operating system performs the same tasks that we described earlier for a uni-processor system.

Consider the logs of two processors shown in Figure 1. All the memory operations shown access the same memory location. The memory operations marked with a cross are the ones that result in a cache miss, and therefore result in a log record. Notice that there are shared-memory dependencies between the two executions. In any cache coherent multi-processor, before a node can write to a memory block, it first has to gain exclusive permission to that cache block. This results in invalidation of this memory block privately cached in all the other nodes. As a result, when a processor core tries to read a value written by another processor core, it results in a cache miss. `W7` shown in the figure is an example. Thus, our logging mechanism implicitly captures the new value produced by a remote processor. This is the key property that allows us to replay the execution of a processor core independent of the other cores. We achieve this without any changes to the coherence protocol.

To replay the execution of `P1` in this two processor multi-core system, the replayer simply takes the log recorded by `P1`, initializes the register state, and starts the replay. The replay produces exactly the same sequence of instructions as in the recording phase, along with the input and output values of those instructions. For each memory operation, the replayer can determine its memory address. Also it reproduces the value read or written by a memory operation, which we refer to as the *new* value for the memory operation. Finally, as we described earlier, for a write cache miss we log the value before it is modified by the write. Thus, the replayer can reproduce the value of the location just before the execution of a write, which we call the `old` value for the memory operation.

Thus, without any additional support for a multi-processor system, just using the program input log for each processor core, the replayer reproduces the exact same sequence of memory operations that were executed during recording, along with their addresses, old and new values. The figure on the right in Figure 1 shows the information produced after replay of each processor core's execution. The memory operations are labeled using the address location that they access (in this example, all the accesses are to the same location `x`). The left super-script of a memory operation denotes its old value, and the left sub-script denotes its new value. Along with this information, the operating system records the final memory system state when the execution completes during recording (similar to the core dump collected on a system crash). In the example, the final state of `x` is `3`. In Section 4, we discuss how this information is also sufficient for reproducing the shared-memory dependencies.

3.4. Discussion

We summarize the key additions to the operating system and hardware to support the logging approach that we discussed. The operating system needs to provide support for creating a checkpoint at regular intervals or on a page fault. Creating a checkpoint requires logging the context header for each processor core in its local log (context header does not contain the memory state). Also, on a context-switch it needs to log the context header for the newly scheduled process or thread. The processor on the other hand needs to provide support for logging the data of the cache block fetched on a cache miss, its physical address, and the instruction count. When compared to the system-dependent logging approach that we discussed in Section 2, we believe that this approach is a lot simpler.

With the above system support, we can deterministically replay the execution of a processor core in a multi-processor system. This approach is also system-independent. The recorded information can be replayed on an operating system different from the one where it was recorded, which could improve the usability of such a record-and-replay solution. In fact, Intel's pinSEL tool [17, 20] exploits this property to enable cross-platform architectural simulation. Capo [15] discusses about recording and replaying a subset of the processes executing in a system using a notion of replay spheres. We believe that such a flexibility could also be provided in our system (in fact, our system might be able to support replay spheres that includes just a subset of the threads in a multi-threaded program), but we leave that design for future work.

4. Offline Analysis Using SMT Solver

In Section 3 we described how our load-based program input recording approach is sufficient for deterministically replaying the execution of each processor core independent of executions in the other cores. However, a programmer would need the shared-memory dependencies to understand and debug a multi-threaded program.

In this section, we first describe the problem statement and provide an overview of our offline analysis for determining the shared-memory dependencies from the program input logs. Then we describe support for recording light-weight hints during recording, which bounds the complexity of the offline analysis. Finally, we describe how we encode our problem as a formula in first-order logic for which Yices SMT solver [7] can find a satisfiable assignment. The solution produced by the SMT solver gives us a total order between the memory operations executed in different processor cores.

4.1. Problem Statement and Offline Analysis Overview

To replay a checkpoint interval, we would have recorded the load-based program input log for each core. In Section 3 we discussed how using this program input log, a replayer can deterministically replay the execution of each core, and produce a trace of memory operations along with their physical addresses, old and new values. Figure 2 shows two examples, where each example is for a different multi-processor execution. Given this information, an offline analyzer needs to determine the shared-memory dependencies. That is, it needs to determine a total order for the memory accesses accessing the same location

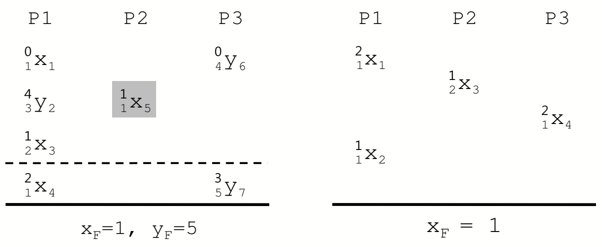


Figure 2. Two trace examples after replaying the execution of each processor core independently using load-based input logs. The dotted line represents a Strata hint.

such that it satisfies two properties: (a) consistent load-store semantics. That is, a memory access's old value should be same as the new value of the memory access that immediately precedes it in the derived total order for the location it accesses, (b) the program order between the memory accesses to different locations execute in a processor core needs to be satisfied (the strict program order could be relaxed to support relaxed consistency model, which we discuss later). We encode these two constraints for each memory operation as a first-order logic formula and use a SMT solver to find a satisfiable solution (Section 4.3 describes this in detail).

In left example shown in Figure 2, we need to derive a total order between `x1, x3, x4, x5` and between `y6, y2, y7`. For accesses to location `y`, the only valid total order is `y6->y2->y7`. For `x`, `x5` can be ordered between `x1` and `x3`, or as the last access to the location `x`. *Both are valid orders*, because of the following reason.

We say that both total orders are valid for replay, because both of those two orders would produce exactly the same sequence of instructions with exactly the same input and output values for those instructions. Therefore, any bug exposed in an execution with one total order would also be exposed in an execution with the other total order. This means that the derived total order might be different from the total order seen during recording. But it does not make a difference for debugging, because the two executions are equivalent (as they produce the same sequences of instructions with the same input/output values).

4.2. Recording Strata Hints

A multi-processor execution for a checkpoint interval could contain millions or even billions of memory accesses. This leads to an exponential increase in the search space for the SMT solver. Therefore, we need to bound the analysis window to a reasonable limit. We achieve this by log hints during recording. We now explain what the hints are and how they can be recorded without incurring significant hardware complexity.

Our hints are similar to Strata [16]. A stratum log consists of the (committed) instruction counts of all the processor cores at an instant in the recorded execution. In Figure 2, a stratum hint is shown using a dotted line. A stratum log recorded at a particular instant of time t during an execution gives the replayer a happens-before relation between all the memory accesses that were executed before t and all the memory operations that were executed after t. For example, the replayer uses the stratum log shown in Figure 2 to determine that `x1, y2, x3, x5, y6` executed before `x4, y7`.

All the cores in a multi-core processor can log a stratum without any synchronization or communication between them. In modern multi-core processors, each core already has a timestamp counter, which is read by instructions like `RDTSC` [1]. It is updated by every core at every cycle, and they are kept synchronous across the cores [1]. After every N cycles, each processor core records its current instruction count in its load-based program input log (described in Section 3) along with a type bit that specifies that this instruction count belongs to a stratum. Using this information recorded in each processor's program input log, the replayer constructs the strata offline. Note that the strata that we log does not record the shared-memory dependencies, unlike in the earlier design that used Strata [16]. A stratum in our design simply bounds the window of our offline analysis. Whereas, in previous work [16] strata were used to record the shared-memory dependencies and therefore its semantics are very different from that of the strata hints we use. As a result, previous work that uses strata [16] also has the same complexity issues that we described in Section 2 for designs like ReRun [8] and DeLorean [14].

In Section 5 we analyze how frequently we need to log a stratum and show that logging a stratum every hundred thousand processor cycles is sufficient for most applications, and the space overhead is more than 10 times less than the race logs recorded in ReRun [8].

We refer to the memory operations executed between two strata as a strata region. The offline analysis can analyze one strata region at a time, and thus its search space is bounded. However, for analyzing a strata region, the analyzer needs to know the final memory state at the end of the strata region. For the last strata region in the recorded execution, we know the final state from the core dump. Thus, we analyze the last strata region first. Once a strata region is analyzed, the initial memory state for that region can be determined. The initial memory state of a strata region is guaranteed to be same as the final memory state of the strata region that immediately precedes it. This is an important property, as it ensures that we do not have to backtrack our analysis for older strata regions. We now provide an informal proof for why this property is true.

Consider the second example show on the right in Figure 2. All the memory operations are to the same location `x`. Analysis over this strata region produces a total order for these memory operations. The old value for the *first memory operation* in the total order derived for this strata region is the initial value for `x` at the beginning of the strata region. We need show that this initial value will be the same in any valid total order that satisfies the load-store semantics constraint (described in Section 4.1). Observe that there is a one-to-one mapping between the set of old values (which includes the final state) and the set of new values for the memory operations accessing a memory location, except for one element in the old-value set. That one unmatched old value has to the be initial value. In the left example in Figure 2, the set of old values is $\{2, 1, 2, 1, 1\}$ (which includes the final state 1) and the set of new values is $\{1, 2, 1, 1\}$. The only unmatched old value is `2`. The first memory operation `x` in the derived total order has to be a memory operation with its old value as `2`. Otherwise, load-store semantics would be violated. Notice that there are two valid total orders for operations accessing `x`. In one order, the first memory operation to `x` is `x4` and in the other it is `x1`. The offline analyzer might produce either of these valid orders, but both will have exactly the same initial value as in the recorded execution. Thus the initial values determined for a strata region is deterministic.

4.3. Offline Symbolic Analysis

Now we describe how we encode the load-store semantics and program order constraints for memory accesses in a strata region as a first-order formula. Before we perform the encoding, we perform two reductions. One, we eliminate read-only accesses. It includes all accesses to a location that is only read in the strata region, because any order between them is a valid order. Two, we eliminate local accesses. It includes all accesses to a location that is accessed in only one processor core. Note the read-only and local property needs to be true only within a strata region. A memory access in a smaller strata region will have a higher probability of either begin a local or read-only. We also eliminate read/write accesses to uncacheable memory locations.

4.3.1. Encoding Satisfiability Equations Satisfiability modulo theories (SMT) generalizes Boolean satisfiability (SAT) by adding linear (in)equalities, arithmetic, arrays, lists and other useful first-order theories. By implementing theories like arithmetic and inequalities, SMT solvers have the promise to provide higher performance than SAT solvers that work on bit level encodings. Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. We use the Yices [7] SMT solver.

The algorithm to encode the constraints for a set of memory operations in a strata region is shown in Figure 1. We use two variables for each memory event to represent the SMT formula. One variable is called the Partial Order (PO) variable represented as O_i, whose values give us a partial order for all the memory operations and ensures the program order for memory events of a processor core. The second variable is called the Total Order (TO) variable represented as M_i, whose values give us a total order for the set of memory accesses to the memory location M. We explain our encoding using the example in Figure 2 without assuming that there is a strata log (that is encoding for the entire execution is presented just for clarity).

The encoding for a strata region takes two kinds of inputs: a set of concurrent memory event traces and a final state dump. The set of final state dump is defined as $D = \{(v_1, val_1), \ldots, (v_n, val_n)\}$, where v_i is a memory location and val_i is its final value. The set of memory event traces is defined as $E = \langle E_1, \ldots E_n \rangle$, where E_p is the memory event trace obtained from processor core p. Let $|E|$ be the total number of events. A memory event $e_i \in E_p$ has the form (v, val_1, val_2), where v is the physical address, val_1 is the old value and val_2 is the new value.

The domain of PO variables is $[1..|E|]$. Let $|E_X|$ be the number of events in E that access memory location X. Then the domain of TO variables of memory events accessing X is $[1..|E_X|]$. Thus for the left example in Figure 2, we have $O_1, \ldots, O_7 : [1..7]; X_1, X_3, X_4, X_5 : [1..4]; Y_2, Y_6, Y_7 : [1..3]$.

Lines $1-4$ in Algorithm 1 encode the program order constraints. For our example, the program order constraints are $(O_1 < O_2 < O_3 < O_4) \wedge O_5 \wedge (O_6 < O_7)$. Lines $5-8$ in the algorithm specify the uniqueness constraint for the TO variables that correspond to memory events accessing the same location. This is necessary for TO variables to ensure that we get a total order for memory events accessing the same location. However, this constraint is not necessary for PO variables.

Lines $9-17$ encode the load-store semantic constraints. For each access to x our encoder decides which memory events can be its immediate follower to the same memory location. We call this legal set as FSML (Follower to the Same Memory Location). For a memory event $e = (v, val_{old}, val_{new})$, another event $e' = (v, val'_{old}, val'_{new})$ can be its FSML either if $val_{new} = val'_{old}$. A memory event $e \in E_p$ can have an empty FSML if e is the last memory access to a location v in the processor core p.

We provide the load-store constraints for few memory events for the example on the left in Figure 2. For $x1$, constraint is $(X_1 = X_3 - 1 \wedge O_1 < O_3) \vee (X1 = X_5 - 1 \wedge O_1 < O_5)$, because $x1$'s FSML can only be $x3$ and $x5$. For $x5$, constraint is $(X_5 = X_3 - 1 \wedge O_5 < O_3) \vee (X_5 = 4)$, because $x5$ can precede $x3$ or could be the last access to the location x. For $y6$, the constraint is $(Y_6 = Y2 - 1 \wedge O_6 < O_2)$, because it can precede only $y2$. For $y7$, the constraint is $Y7 = 3$, because only y's final state value matches the new value of $y7$.

The SMT formula is then given to the Yices SMT solver [7]. The satisfiable solution that it finds gives us the values for the total order and partial order variables of every memory event, which gives us the shared-memory dependencies.

Algorithm 1 SMTENCODING(STRATASET E, FINALSTATE D)

```
1:  for $p = 1; p \leq P; p++$ do
2:      Let $E_p = \langle e_m, e_{m+1}, \ldots, e_n \rangle$;          // Memory events in processor core $p$
3:      add constraint $O_m < O_{m+1} < \ldots < O_n$;          // Program order constraint
4:  end for
5:  for all memory location $v$ do
6:      Let $M_v = \{e_p, e_{p+1}, \ldots, e_q\}$ be the memory event set that access $v$;
7:      add constraint $X_p \neq X_{p+1} \neq \ldots \neq X_q$;          // Uniqueness constraint
8:  end for
9:  for all $e_i = (x, i_{old}, i_{new}) \in E$ do
10:     let $FSML_i$ be the FSML set of $e_i$
11:     $\mathcal{C}_i = \bigvee_{e_f \in FSML_i} (X_i = X_f - 1 \wedge O_i < O_f)$;          // load-store semantic constraint
12:     if $x = i_{new} \in D$ and $e_i$ is the last access to $x$ in $p = processor(e_i)$ then
13:         $\mathcal{C}_i = \mathcal{C}_i \vee (X_i = |M_x|)$;
14:     end if
15:     add constraint $\mathcal{C}_i$
16: end for
```

4.4. Supporting Relaxed Consistency Models

Relaxed consistency models can be supported in our solution without any additional processor support. In our current encoding, the program order constraint specifies a total order between all the memory operations executed by a processor core. But, say in a system that support Total Store Order (TSO) consistency, a total program order would be specified only for the store operations executed in a processor core. This would relax the ordering of loads with respect to its preceding stores.

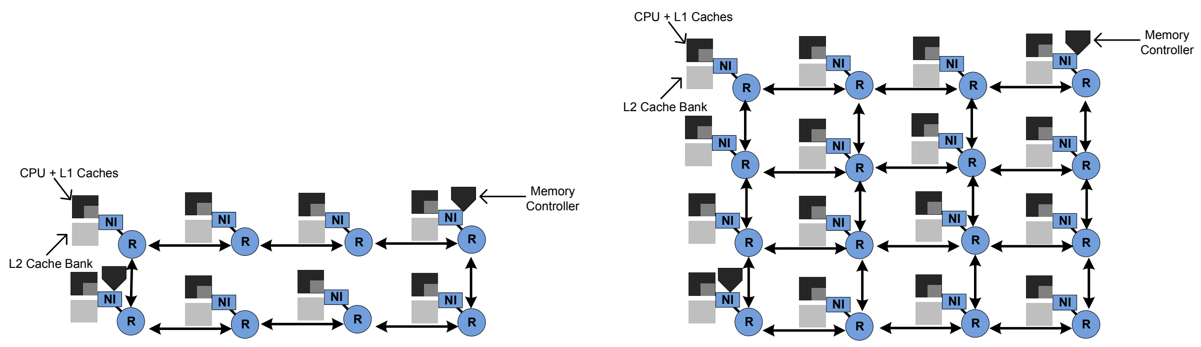


Figure 3. Processor Model

5. Results

A key advantage of our proposal is that it is complexity-efficient. In this section, we analyze the log size and performance overhead of our approach. We also compare the Strata hint log size (required for our offline analysis) to the memory race log size in ReRun [8], which is the state-of-the-art hardware solution for recording shared-memory dependencies.

5.1. Evaluation Methodology

We use Simics [13] as the front end for full system functional simulation, and our cycle accurate simulator as the back end. We analyze two processor configurations, one with 8 cores and another with 16 cores. The processor configuration is shown in Figure 3. We model MESI coherence protocol. We model a ring network for the 8-core system with two memory controllers. We use a mesh for the 16-core system also with two memory controllers. Other processor configurations are listed in Table 1. We picked a representative set of benchmarks from various benchmark suites, which are listed in Table 2.

Processor Pipeline	2 GHz processor, 128-entry instruction window
Fetch/Exec/Commit width	2 instructions per cycle in each core; only 1 can be a memory operation
L1 Caches	32 KB per-core (private), 4-way set associative, 32B block size, 2-cycle latency, write-back, split I/D caches, 32 MSHRs
L2 Caches	1MB banks, shared, 8-way set associative, 64B block size, 6-cycle bank latency, 32 MSHRs
Main Memory	4GB DRAM,up to 16 outstanding requests for each processor,320 cycle access, 2 on-chip Memory Controllers.
Network Router	2-stage wormhole switched, virtual channel flow control, 6 VC's per Port, 5 flit buffer depth, 8 flits per Data Packet, 1 flit per address packet.
Network Topology	a ring for 8-core, and 4x4 mesh for 16-core, each node has a router, processor, private L1 cache, shared L2 cache bank (all nodes) 2 Memory controllers, 128 bit bi-directional links.

Table 1. Baseline Processor, Cache, Memory, and Network Configuration

5.2. Strata Region Length

Our first analysis is to determine the appropriate length for the strata regions. We experimented with four different processor cycle bounds. Figure 4(a) shows the number of strata intervals for each bound. Each strata interval would have different

Programs	Description
SPECjbb	Java ServerWorkload. SPECjbb2000 is a Java based benchmark that models a 3-tier system. We use as many warehouses as there are number of processors and we start measurements 30 seconds after ramp-up time.
Apache	Static Web Serving. We use Apache 2.0.43 for SPARC/Solaris 10 with default configuration. We use SURGE [3] to generate web requests. We use a repository of 20,000 files (totaling 500 MB). We simulate 400 clients, each with 25 ms think time between requests.
SPEComp	We used SPEComp2001 as another representative workload. We present the results from `applu`, `art` and `swim` due to space constraints. Fast forwarded for 100 million instructions.
SPLASH 2	SPLASH is a suite of parallel scientific workloads. We consider `barnes`, `ocean` from this suite. Each benchmark executed sixteen parallel threads running on a eight way CMP. Fast forwarded for 100 million instructions.

Table 2. Benchmarks

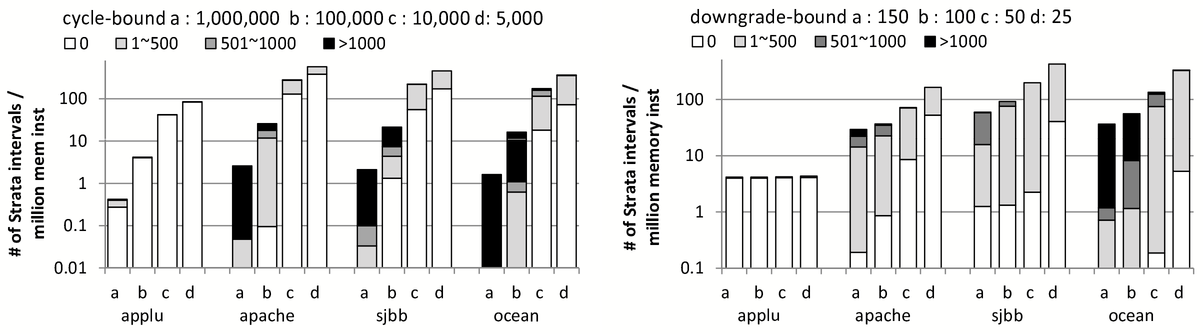


Figure 4. Distribution of unfiltered memory events in a strata interval (classified over four ranges) for different bounds. The y-axis shows the number of intervals seen for a particular unfiltered memory event range for million memory instructions on average.

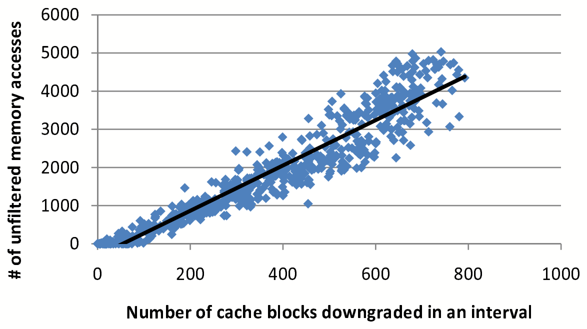
number of *unfiltered* memory accesses to analyze (accesses left after filtering read-only and local accesses from a strata region) depending on the program characteristics. More memory events per strata region would increase the cost of offline analysis. Therefore, we would like the unfiltered accesses per strata region to be less than 500 (reason is discussed later in Section 5.3). Figure 4(a) shows that for programs like `applu` even if we use a bound of a million cycles (a stratum is created after a million processor cycles has elapsed), most intervals would still be left with less than 500 unfiltered accesses. But for programs like `Ocean` we need a lower cycle bound, because we find many intervals with more than 1000 unfiltered accesses if we use a higher bound. Based on the application to be recorded, the operating system can set the cycle bound appropriately.

We also tried to define a bound that is based on a metric different from the processor cycles. Each core counts the number of downgrade requests (invalidation or downgrade exclusive permission), and if any core reaches a predefined bound, then it sends a message to all the nodes to log a stratum. This approach is more complex than using a cycle-bound approach, but could reduce log size. Figure 5(a) plots the number of unfiltered accesses versus the downgrade count for different program intervals (measured in terms of cycles). As one can see, there exists significant correlation between the downgrade requests and unfiltered accesses, because they are a better indicator of the sharing behavior in an application (more sharing would result in more memory accesses in an interval). Figure 4(b) shows results for downgrade bound approach for four different d values. For example, for `SPECjbb`, compare configuration a in cycle bound and downgrade bound approach. In the downgrade bound approach there are no intervals with greater than 1000 unfiltered accesses.

Filtering the local accesses and read-only accesses within a strata region eliminates over 99% of memory accesses from offline analysis, except for `Ocean`. Figure 5(b) shows this result for a configuration where the strata regions are constructed when the downgrade count in any processor reaches 100 (d = 100).

5.3. Log Size and Symbolic Analysis Performance

Figure 7(b) shows the time taken (y-axis uses a log scale) to analyze strata regions with different numbers of unfiltered accesses. The strata regions are from the executions of `Ocean, Apache, SPECjbb` when we used a high bound on the number of processor cycles. We can analyze an interval with about 500 unfiltered access in about 5 seconds on average.



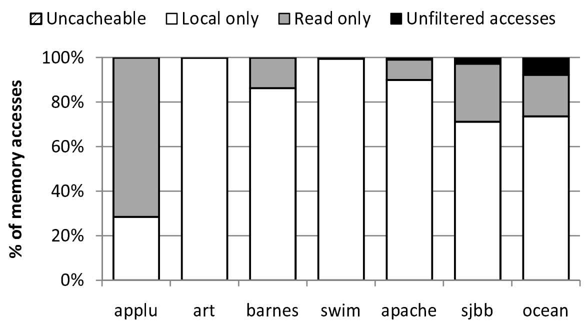
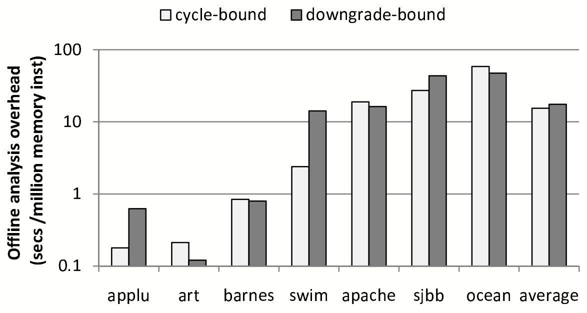


Figure 5. Correlation of downgrade counts with the number of unfiltered accesses. The figure on the right shows the effectiveness of filtering read-only and local accesses.



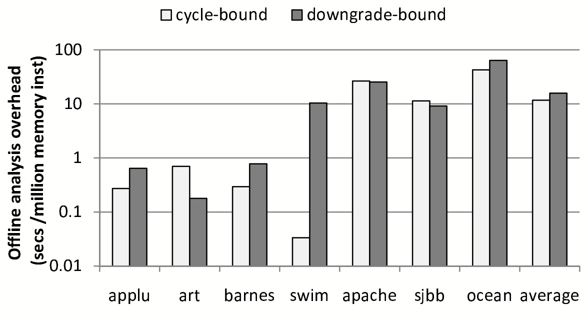
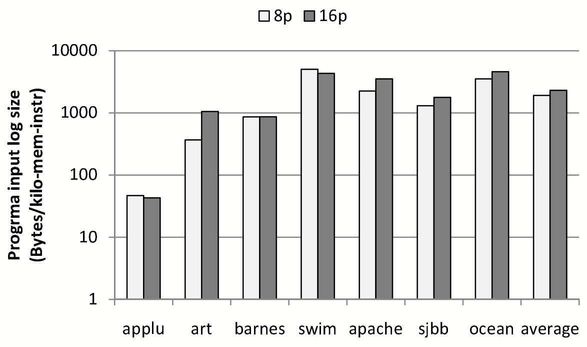


Figure 6. Offline Analysis Cost.

Figure 6 shows the time taken to analyze million memory instructions (filtered and unfiltered accesses) on average. The user or the operating system specifies the bound based on profiling the program characteristics (it could also be based on the log size versus offline cost one is willing to make). Offline analysis takes about 15 seconds to analyze an execution with a million memory operations on average. This cost is paid only once before replay. Once analyzed, the replayer need not incur this analysis cost, and therefore can be interactive. The results also show that the simpler cycle-bound approach is sufficient.

Figure 7(a) shows the log size for our program input logs. Figure 8 shows the strata log size. We need about 10 bytes of strata log for every thousand memory instructions executed in the 8-processor configuration. We also show the memory race log size for ReRun [8], one with ideal bloom filter, and another that uses bloom filters of sizes 32 bytes for read set and 128 bytes for the write set using Jenkins' hash function. ReRun requires a log that is 10 times larger. But, more importantly, our approach is complexity-effective.



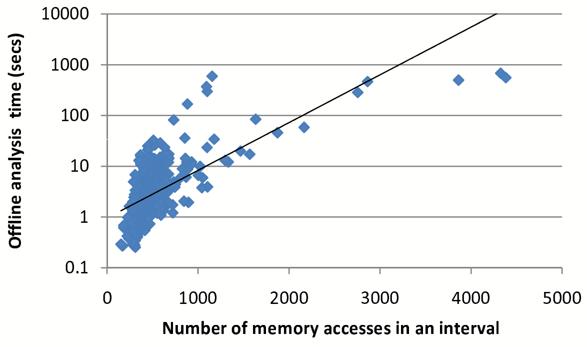


Figure 7. Program input log size. Scalability of offline analysis.

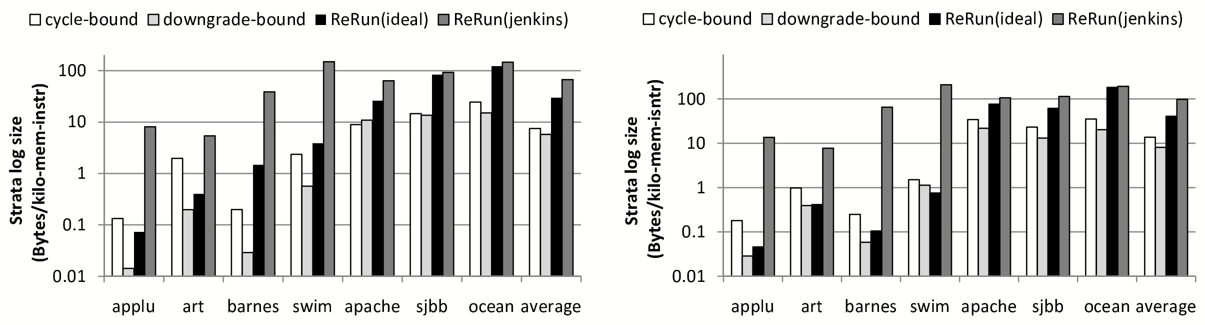


Figure 8. Strata log size for 8 and 16 processor configurations.

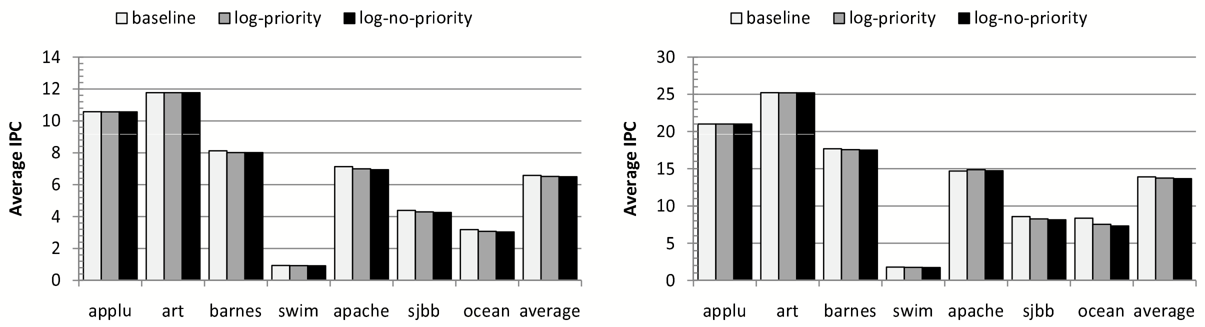


Figure 9. Performance overhead for 8 and 16 processor configurations.

5.4. Recording Performance

We analyze the performance overhead for the recording for the processor configuration described in Section 5.1. On a cache miss, the cache block fetched is directly written back to the main-memory along its physical address and the current instruction count of the processor core. We evaluated an optimization were the packets that write-back recorded logs are given a lower priority in the routers. Figure 9 shows the performance degradation. The worst degradation is for `ocean` in the 16-processor configuration (12.65% slowdown). The priority optimization reduces the overhead to 10.06%. On average, the non-prioritized scheme incurs 1.3% and 1.73% slowdown 8 and 16 processor core configurations. Whereas, prioritized scheme incurs 0.99% and 1.17% overhead respectively.

6. Conclusion

Support for deterministic replay could be extremely useful for developing parallel programs. Over the past few years, the architecture community has made significant progress is developing hardware designs that are both performance and space efficient. In this paper, we focused on reducing the hardware complexity of a recorder. We discussed a solution, where a program input log consisting mainly of the initial register state and cache miss data was sufficient for ensuring replay of the execution in multi-processor system. Much of the complexity is off-loaded to a novel symbolic analysis algorithm, which uses a SMT solver and determines the shared-memory dependencies from the program input logs. We believe that the proposed approach is simple enough that the hardware vendors could soon adapt it and include it in their next generation processors.

References

[1] "http://en.wikipedia.org/wiki/time_stamp_counter."

[2] D. F. Bacon and S. C. Goldstein, "Hardware assisted replay of multiprocessor programs," in *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM Press, 1991, pp. 194–206.

[3] 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*.

[4] 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.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *ISCA*, 2007, pp. 278–289.

[6] 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.

[7] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *Proceedings of the 18th Computer-Aided Verification conference*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 81–94.

[8] D. R. Hower and M. D. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," in *International Symposium on Computer Architecture*, 2008.

[9] 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.

[10] L. Lamport, "Time, clocks and the ordering of events in a distributed system," in *Communications of the ACM*, 1978.

[11] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transaction on Computers*, vol. 36, no. 4, pp. 471–482, 1987.

[12] 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.

[13] 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*, vol. 35, no. 2, pp. 50–58, 2002.

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

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

[16] 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*, 2006, pp. 229–240.

[17] 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.

[18] S. Narayanasamy, B. Carneal, and B. Calder, “Patching processor design errors,” in *ICCD*, 2006.

[19] 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.

[20] C. Pereira, H. Patil, and B. Calder, “Reproducible simulation of multi-threaded workloads for architecture design exploration,” in *IISWC*, 2008, pp. 173–182.

[21] M. Prvulovic, Z. Zhang, and J. Torrellas, “Revive: Cost effective architectural support for rollback recovery in shared-memory multiprocessors,” in *Proceedings of the 29th Annual International Symposium on Computer architecture*. IEEE Computer Society, 2002, pp. 111–122.

[22] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, “Safetynet: Improving the availability of shared-memory multiprocessors with global checkpoint/recovery,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 123–134.

[23] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, “Flashback: A lightweight extension for rollback and deterministic replay for software debugging,” in *USENIX Annual Technical Conference*, 2004, pp. 29–44.

[24] M. Xu, R. Bodik, and M. Hill, “A flight data recorder for enabling full-system multiprocessor deterministic replay,” in *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.

[25] M. Xu, M. D. Hill, and R. Bodik, “A regulated transitive reduction (rtr) for longer memory race recording,” in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 49–60.

[26] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman, “Retrace: Collecting execution trace with virtual machine deterministic replay,” in *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*.