Offline Symbolic Analysis for Multi-Processor Execution Replay

Dongyoon Lee[†], Mahmoud Said[*], Satish Narayanasamy[†], Zijiang Yang[*], Cristiano Pereira[‡]

University of Michigan, Ann Arbor[†] Western Michigan University[*] Intel[‡]

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 its *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 execution replay. We show that a load-based checkpointing scheme, which was originally proposed for just recording program input, is also sufficient for replaying every thread in a multi-threaded program. Shared-memory dependencies between threads are reconstructed *offline*, during replay, using an algorithm based on an SMT solver. In addition to saving log space, the proposed solution significantly reduces the complexity of hardware support required for enabling replay.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design, Performance, Reliability

Keywords

Multi-processor Replay, SMT solver, Shared-Memory Dependencies

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 [11]. It also helps a programmer debug non-deterministic bugs that are prevalent in multi-threaded programs. Heavy-weight dynamic analysis tools, such as data-race detectors, cannot be used to analyze an unperturbed natural program execution. However, with an efficient recorder, one could record a realistic program execution, and then analyze that execution using any heavy-weight dynamic analysis tool by replaying it [5]. If the recording overhead is insignificant, then even production runs could be continuously recorded. In the event of a software failure, recorded information would provide programmers with significantly more information than crash dumps used today.

MICRO'09, December 12–16, 2009, New York, NY, USA.

Recently, several hardware techniques [10,16,17,22,27] have been proposed for efficiently recording a program's execution. To replay a shared-memory multi-threaded program's execution on a multi-processor system, existing solutions record two types of information at runtime – the program's input and the shared-memory dependencies between 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 information can be recorded in the operating system (or virtual machine) using a copy-on-write checkpointing mechanism. This could be made efficient using processor support [24, 25]. However, this approach is system-dependent. Instead, we could use a system-independent load-based checkpointing scheme called BugNet [22] for recording a program's input. This scheme records the initial register state and the values of load instructions executed by the recorded program. The recorded load values implicitly capture the system input from I/O, DMA, interrupts, etc., and thereby avoids the complexity in detecting and logging numerous types of non-deterministic system events. Thus, BugNet is system-independent, and we will refer to this approach as load-based checkpointing scheme.

In addition to recording a program's input, existing solutions, including BugNet, assume support for detecting and recording shared-memory dependencies between concurrent threads. Any software based solution to this problem [**?**, 20] would require monitoring every memory access, and therefore would result in an order of magnitude slowdown. There has been significant work in the recent years to provide efficient hardware support [10, 16, 19, 27, 28] for detecting and logging shared-memory dependencies. Processor-based solutions not only detect and log shared-memory dependencies, but they also optimize the log size by not logging a dependency if it is transitively implied by an earlier dependency log. However, significant changes need to be made to a processor design to accomplish these tasks. For example, the state-of-the-art solution called ReRun [10] maintains read/write sets in each processor core using bloom filters, tracks a Lamport Scalar Clock [12] in each core, piggy-backs coherence messages with additional information, and detects shared-memory dependencies by monitoring those coherence messages. The coherence mechanism is one of the most difficult to verify subsystem in a processor. For instance, AMD64 processor had 9 design bugs that can be attributed to multi-processor support [18]. Our goal is to enable a low-cost hardware solution that does not require changes to the coherence sub-system, and thereby make it feasible for processor manufacturers to provide support for multi-processor execution replay.

We propose a record-and-replay solution that does not require any support for detecting and logging shared-memory dependencies. We assume sequentially consistent memory model in this work, but we can extend our solution to support relaxed memory models as well. We show that if we

use a load-based checkpointing mechanism for recording program input [22] (instead of a system-dependent program input recording scheme), we do not have to record shared-memory dependencies at all. The reason is as follows. For each thread in a multi-threaded program, a load-based checkpointing 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. This does not require us to record and replay shared-memory dependencies. By deterministically replaying each thread in isolation, we can reproduce the exact same sequence of instructions executed by a thread during recording. The replay also reproduces the input and output values of those instructions (input of a memory operation includes its address as well).

Thus, using load-based checkpoints, we can deterministically replay each thread in isolation without having to reproduce shared-memory dependencies between the threads. However, to debug a multi-threaded program, a programmer would still need to be able to understand the interactions between the threads. This would require reproduction of shared-memory dependencies during replay. This information, however, can be determined using an offline analysis, which works as follows.

Using load-based checkpoints, we deterministically replay each thread. This gives us the trace of all the memory operations executed by a thread along with the address and input/output values of those memory operations. From the final core dump, we also obtain the final state of every memory location. Using all of this information, we determine the program order and input/output constraints for all the memory operations executed by all the threads. These constraints are then encoded in the form of a satisfiability equation. The solution for this equation is found using a SMT (Satisfiability Modulo Theory) solver called Yices [9]. The solution gives us the total order for the memory operations executed by all the threads, using which a programmer can reason about the dependencies between the threads.

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 regular intervals. The length of an interval is fixed and it is specified in terms of the number of processor cycles. Thus, to log these hints, we do not require additional communication between the processor cores.

In addition to logging hints, we require hardware support for supporting load-based checkpointing, which essentially requires support for logging cache misses. 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 complexity of hardware and software support required for recording a program. But, we have to pay the cost for offline analysis before we can replay a recorded execution. The offline analysis need to be performed only once and not every time we replay. 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 the cost of offline analysis is an acceptable trade-off, because efficiency matters most 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 execution of a program, and analyze them using various heavy-weight dynamic analysis tools such as a data race detector by replaying it. Though an efficient replayer would also be useful, an efficient recorder is more important. We hope that our low-complexity hardware-based recording solution would convince processor manufacturers to incorporate replay support in the next generation processors.

2. Background and Motivation

Success of multi-core processors depends largely on whether we can make parallel programming accessible to mainstream programmers. This could be aided through technologies such as record and replay. Record and replay is especially useful for debugging multi-threaded programs as concurrency bugs are notoriously difficult to reproduce and understand. Industry has recognized this need as many leading companies including IBM, VMWare, Microsoft and Intel have all invested in developing replay solutions [5, 7, 21, 29].

Software community has been working on replay solutions for over two decades [13]. Recent developments such as ReVirt [8], Flashback [26] and ReTrace [29] can record and replay a program's execution on a uni-processor system for less than 10% performance overhead. Mostly, these solutions record just the program input, which is sufficient for replaying an execution on a uni-processor system. The most common solution used to record program input consists of a copy-on-write checkpointing scheme that records a program's initial register *and* memory state, and a system event logger that records return values and timestamps of all non-deterministic system events (such as system calls, DMA, I/O, etc.). We refer to this approach as *system-dependent* program input logging approach, because an execution recorded using this approach can be replayed only in a system environment that is same as the one in which the execution was recorded. In depth discussion of the disadvantages and the complexity of a system-dependent program input recording solution can be found here [21, 22].

Thus, there are solutions for efficiently recording and replaying program executions on a uni-processor system. Replaying an execution on a multi-processor system, however, remains a very challenging problem. Because, in addition to recording program input, existing solutions require support for logging shared-memory dependencies as well. For a software recorder to detect and log shared-memory dependencies between threads, it has to examine the execution of every memory operation. This is clearly expensive in terms of performance cost. Several researchers in the recent past have proposed to address this problem using hardware support.

Bacon and Goldstein [3] proposed to record all the coherence traffic in a snoopy-bus based multi-processor system. FDR [27], RTR [28], Strata [19], DeLorean [16], and ReRun [10] are all recent developments that improved the hardware design for logging shared-memory dependencies. They all focused on reducing the log size and the amount of hardware states required to detect and log shared-memory dependencies. While they succeeded in reducing the cost of hardware real estate, the hardware complexity of those solutions is so high that processor manufacturers have been reluctant to adopt them.

To illustrate the complexity of a shared-memory dependency logger, we now discuss two recently proposed hardware solutions, DeLorean [16] and ReRun [10]. In Section 5, we compare the sizes of our logs to those of ReRun's.

DeLorean assumes support for BulkSC [6]. 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, each core in DeLorean just needs to record the size of each chunk that it executes. Also, a global arbiter records the total order between chunks executed in different cores. DeLorean drastically reduces the log size required for recording shared-memory dependencies. However, it introduces additional hardware complexity for supporting BulkSC [6], a global arbiter for logging the order between chunks, and support for logging chunk sizes.

ReRun [10] forms episodes and records their sizes along with a total order between them. Similar to a chunk in DeLorean, an episode is also a sequence of instructions that appear to be atomic in an execution. But ReRun differs from DeLorean in implementation details, such as the conditions for terminating an episode. Before sending an invalidation acknowledgment or a data update message to any coherence request, a processor core in ReRun terminates its episode if it detects a read-write or a write-write conflict between the requesting access and one of its past memory accesses. An episode is also terminated when a cache block gets evicted, because the core would no longer receive any coherence message for the evicted cache block. Thus, ReRun guarantees atomicity property for an episode.

ReRun is very efficient in terms of log size (about 4 bytes/kilo-instruction) and performance. However, it needs significant hardware support. For each core, ReRun 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. While ReRun is efficient in terms of area cost, the complexity of additional control logic in correctly creating the episodes based on shared-memory dependencies and logging a total order between them is significant.

ReRun and DeLorean can record shared-memory dependencies efficiently. But to support replay, in addition, we also need system support for recording program input. Capo [17] discussed the problem of interfacing software system support with a hardware-based record-and-replay system. It advocated software system support for recording program input, which requires a copy-on-write checkpoint mechanism and support for logging non-deterministic system 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 [17]. However, hardware techniques like SafetyNet [25] or ReVive [24] could be used to reduce the cost of copy-on-write checkpointing mechanism.

Instead of using a combination of a software-based program input recorder *and* a hardware-based shared-memory dependency recorder, we propose a design that requires just a hardware supported load-based program input recorder similar to BugNet [22]. In the original proposal, BugNet had assumed additional 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. We also discuss an algorithm based on Yices SMT solver [9], which can be executed offline to determine shared-memory dependencies from the load-based program input checkpoints.

Our approach has three main advantages over prior solutions such as ReRun [10]. First, it is complexity-efficient. To support load-based checkpointing scheme, essentially we only need hardware support for logging cache blocks fetched on cache misses, and support for logging the values of each processor core's memory counter at periodic intervals (which serve as hints for bounding the search space of our SMT solver). Also, unlike ReRun, we do not need software system support for logging non-deterministic system events (which could be very tedious to support in an operating system as there are numerous types of non-deterministic system events to detect and log). Therefore, we also reduce the complexity of the software component of a record-and-replay system as well. Second, we reduce the space overhead as the hint log is about four times smaller than the ReRun's log. Finally, a load-based logging approach is system-independent, which gives developers working in remote sites the flexibility to replay a program's execution in different system versions/environments.

3. Load-Based Checkpointing Architecture

The load-based checkpointing scheme was originally proposed in the BugNet architecture [22] as an alternative to the system-dependent logging scheme for recording program input. The original goal of BugNet was to avoid the system complexity in detecting and recording all types of non-deterministic system input. For this reason, Microsoft's software replayer called iDNA [5] also uses a load-based checkpointing scheme for recording program input. Using iDNA, we can record and replay a multi-threaded program on a multi-processor system. Another software tool called pinSEL [21, 23], currently used at Intel, also implements a load-based checkpointing scheme using Pin [14] to enable replay of multi-threaded programs. However, iDNA and pinSEL incur more than ten times performance overhead [5, 23] when compared to the native execution. Also, unlike our proposal, they also record inter-thread dependencies [23] to replay multi-threaded programs.

In this paper, we show that there is an added advantage to using a load-based checkpointing as we do not have to record shared-memory dependencies. These dependencies can be determined offline by analyzing load-based checkpoint logs. Though our focus in this paper is an efficient hardware solution, our solution could also help reduce the recording overhead of software tools based on load-based checkpointing such as the pinSEL [23], which currently record shared-memory dependencies.

In this section, we briefly describe our load-based checkpointing scheme. Our scheme is a modified version of BugNet [22]. We extend the original design to support replay of the full system and programs with self-modifying code. We discuss a complexity-effective architecture design to support this scheme efficiently. We also describe its unique property that allows us to replay each thread in a multi-threaded program in isolation without the information about shared-memory dependencies. Then we describe additional architectural support required for logging hints that help us bound the complexity of our offline analysis described in Section 4.

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 BugNet [22] 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. The instruction count of a memory instruction is the number of instructions that the

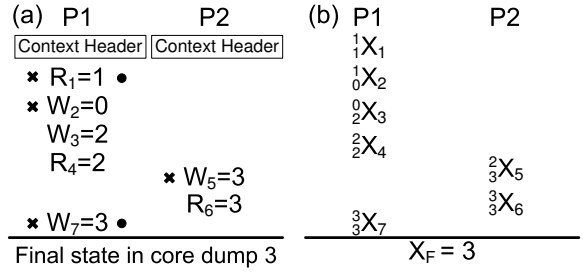

Figure 1: Load-Based Logging Example

program had executed since the beginning of the recorded interval. Unlike in BugNet [22], we also consider instruction read as a load. This extension allows us to handle programs with self-modifying code.

Using the recorded log, a tool like Pin [14] could be used to replay. The replayer emulates the states of the register and the virtual memory of the recorded program. It starts by initializing the register states by reading from the log. It includes the program counter state as well. All the memory states are initialized to invalid. Once initialized, the first instruction specified by the program counter is executed. Since our system treated an instruction read as a load, its machine code can be found in the recorded log. An instruction is replayed as follows. If the instruction is a non-memory operation, it is executed by reading the input values from the emulated register states, and then writing back the result to the emulated 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 emulated memory state is updated with that value. If the instruction is a store, its input values are read from the emulated register state, its effective address is computed, and the emulated memory state is updated with the store value (so that later loads to the same location can get their values). 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 at every instruction replayed.

Recording every load value (including instruction reads) is expensive in terms of log size. But, BugNet logs a load, only if that load is the first memory access to the location that it accesses. Such loads are called as *first-loads*. The values of non-first-loads need not be logged, as they can be read from the emulated memory state.

Figure 1 shows a sample execution. Assume that all the instructions in the example access the same memory location. Consider just the first four instructions executed by the processor `P1` for now. `R1` is the first-load (with a return value `1`), and 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 using the emulated memory state.

To begin logging a program's execution, the operating system first creates a checkpoint by recording the *context header* and turns on logging for the processor core. The context header contains the initial register state, a process identifier and the value of the timestamp counter of the processor core. To detect and log first-loads we need processor support.

BugNet [22] used a bit per cache word in the private cache of a processor core to determine if that location has been logged for the program or not. We use an even simpler design, where we just log the cache block fetched on a (load or store) cache miss, because any first access to a location would result in a compulsory cache miss. In the case of a store miss, the data recorded for the cache block are the values before executing the store.

In BugNet, a memory location's value need not be logged if the first access to it is a store. Because, any store, including the first-store, can be deterministically replayed using the emulated register and memory states. However, we log the cache block fetched on a store miss, because it is possible that later on, the program could execute a first-load to a different word in the same cache block. This might slightly increase the log size when compared to the BugNet design. But it avoids the need to use a bit per cache word, and also helps our offline analysis explained in Section 4. `W2, W5, and W7` operations (denoted with cross marks) in Figure 1 are store misses and are logged in our design.

The data logged for a memory access consist of the instruction count of the memory operation that causes the cache miss, and the data of the cache block fetched. To simplify the design, we choose to not use any additional local log buffers. Instead, we directly write-back the cache block to the log space allocated in the main memory. Note that any read from an uncacheable memory-mapped location would always be logged as it will always result in a cache miss. 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.

A checkpoint for a program is created first when logging is turned on for that program. Thereafter, a new checkpoint is created at regular intervals. The checkpoint interval length is defined based on the available memory space for logging similar to the original BugNet architecture [22]. To create a new checkpoint, the operating system flushes the data in the private caches of the processor, logs the checkpoint header, and then continues to log the data of every cache block fetched on a cache miss.

3.2 Handling System Events

The previous section assumed a uni-processor system, and also that there are no system events that affect a program's execution. We now relax the latter constraint. Unlike BugNet [22], we choose to record the execution of the full system including the operating system code. An interrupt, a system call, or another program can context switch a program executing on a processor core.

On a context switch, the operating system terminates the current log by logging the current instruction count for the processor core (so that the replayer would know when to context switch during replay). It then logs a context header for the new program that is context switched in.

We assume physically tagged private caches. Therefore, on a context switch, private cache blocks are not flushed. This could cause an issue for our logging scheme, as the first access to a virtual memory location by the newly context switched in program might not result in a cache miss. We solve this problem by tracking an additional log-bit per cache block, which is reset on a context switch or when a new checkpoint is created. Either when a memory access results in a cache miss, or when the log-bit is not set for the cache block accessed, we log the physical address of the cache block and set the log-bit. This additional information allows us to emulate the physical address space for the memory state during replay. During replay of the full system, when a program accesses a virtual address for the first time, we can determine its equiv-

alent physical address from the log. Thus, we can establish a map between the virtual and physical addresses for a program during replay and emulate the physical address space.

The mapping between physical addresses and virtual addresses could change after a page fault. We solve this problem by just flushing the private cache blocks on a page fault.

Replaying by emulating physical address space also allows our offline analysis to correctly determine shared-memory dependencies between multiple processes (and of course threads) that concurrently run on different cores.

To replay a checkpoint interval, the replayer starts from the first context header and continues to emulate the register state and 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 proceeds normally.

The above approach ensures replay of the full system execution on a processor core for an interval. We can replay on any operating system as long as we have a tool that emulates the ISA (Instruction Set Architecture) of the recorded processor. 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.

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 as well). 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 in the figure 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 has to first gain exclusive permission to that cache block. This results in invalidation of cache blocks privately cached in all the other nodes. As a result, when a processor core tries to read a value that was last written by another processor core, it triggers a cache miss. `W7` and `W5` shown in the figure are examples. Thus, our logging mechanism implicitly captures the new values produced by remote processors. 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 also reproduce the `old` value for a memory operation, which would be the value of the memory location before it was modified by the memory operation.

Thus, without any additional support for a multi-processor system, just by 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 reproduced after replaying the execution of the two processor cores. 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.

The operating system also records the final memory system state at the end of recording (similar to the core dump collected after a system crash). In the example, the final state of `x` is `3`. In Section 4, we discuss how all this information can be used to determine 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 and reset the log-bits (a log-bit per cache block is not required if we simply choose the flush the private caches on a context switch). 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. Also, log the physical address of a memory access if the log-bit for the cache block accessed is not set. 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 [21, 23] exploits this property to enable cross-platform architectural simulation. Capo [17] 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 why a load-based checkpoint is sufficient for deterministically replaying an execution of a processor core without having to concurrently replay executions in the other cores. However, a programmer would still to replay shared-memory dependencies between concurrent executions in different cores to understand and debug a multi-threaded program.

In this section, we describe an algorithm based on an SMT solver [9] for determining shared-memory dependencies offline by replaying the execution of each processor core using the load-based checkpoint logs. We also describe processor support for recording light-weight hints during recording in order to bound the complexity of our offline analysis.

4.1 Problem Statement and Offline Analysis Overview

An interval of a multi-processor execution is recorded using a load-based checkpoint log for each processor core. Using this information, a replayer can deterministically replay the execution of each core and produce a trace of memory operations for that execution. The trace contains the physical address, the old value and the new value for every memory operation that is replayed.

Figure 2 presents two example traces collected for two different multi-processor executions. Given a trace for a multi-processor execution, the goal of an offline analyzer is to determine the shared-memory dependencies between concurrent executions of threads on different cores. That is, it needs to determine a total order for the memory accesses. The total order should satisfy the following two constraints.

The first constraint is that the derived order should obey the load-store semantics. That is, a memory operation Q can be dependent on another operation P in the derived total order, only if Q's old value is equal to the new value of P (in Figure 2, `y2` can only follow `y6`). The new value of the last access to a memory location in the derived total order should be equal to the final state of that location in the core dump.

The second constraint is that the derived total order should preserve the program order between all the memory accesses executed in a processor core. This second constraint is necessary to ensure that the derived memory order is valid for a concurrent execution on a sequentially consistent memory model (which is what we assume in our work). By relaxing this constraint we might be able to support relaxed consistency models.

The two constraints can be determined for each memory operation from the traces that are obtained by replaying the execution of each processor core. These constraints are then encoded as a first-order logic formula, which is then solved using an SMT solver. Section 4.4 describes this in detail.

4.2 Replay Guarantees

Using load-based checkpoint logs we can deterministically replay a processor core's execution to reproduce the exact same control flow and sequence of instructions along with their input and output values. In addition, the offline analysis produces a valid total order for all the memory operations that satifies the two constraints mentioned earlier. Thus, we can reproduce an execution that has the same input/ output and final program state as the recorded execution.

However, it is possible that for a recorded execution there are multiple valid memory order that satisfy our constraints. For the example in Figure 2(a), there are multiple valid total orders that satisfy our constraints. Total orders `y6->x1->y2->x2->x3->x4->y7` and `y6->x1->y2->x3->x4->y7->x2` are both valid. Note that for accesses to location `y`, the only valid order is `y6->y2->y7`. For `x`, however, `x5` can be ordered between `x1` and `x3`, or as the last access to the location `x`.

If multiple memory orders can satisfy the above constraints for a recorded execution (which is rare), our tool produces all of them, including the memory order seen during recording. Running a data-race detector for each of those interleavings is guaranteed to find the data-race that exists in the recorded execution. In fact, producing multiple interleavings that produce the same incorrect state is equivalent to producing multiple proofs for the same bug, which could be more valuable to a programmer.

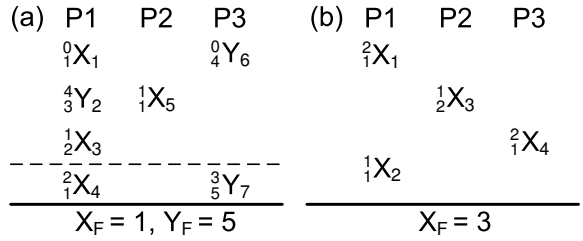

Figure 2: Two trace examples after replaying the execution of each processor core independently using load-based input logs. The dotted line represents a Strata hint.

4.3 Recording Strata Hints

A multi-processor execution for a checkpoint interval could contain millions or even billions of memory accesses. This leads to an astronomical 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 [19]. 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`. 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 [19]. A stratum in our design simply bounds the window of our offline analysis. Whereas, in previous work [19] 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 [19] also has the same complexity issues that we described in Section 2 for designs like ReRun [10] and DeLorean [16].

In Section 5 we analyze how frequently we need to log a stratum and show that logging a stratum every ten thousand processor cycles is sufficient for most applications, and the space overhead is more than 4 times less than the race logs recorded in ReRun [10].

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 shown 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 to 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 an 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 right 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.

4.4 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 being either local or read-only. We also eliminate read/write accesses to uncacheable memory locations.

4.4.1 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 [9] SMT solver.

The algorithm to encode the constraints for a set of memory operations in a strata region is shown in Algorithm 1. We introduce one symbolic variable for each memory event. The variable is called the Event Order (EO) variable represented as O_i, whose values give us a total order for all the memory operations and ensures the program order for memory events of a processor core. 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 EO variables is $[1..|E|]$. Thus for the left example in Figure 2, we have $O_1, \ldots, O_7 : [1..7]$.

Lines $1-4$ in Algorithm 1 encode the program order constraints. For our example, the program order constraints are $(O_1 < O_2 < O_3 < O_4) \wedge (O_6 < O_7)$. Lines $5-9$ in the algorithm specify the uniqueness constraint for the EO variables. This is necessary for EO variables to ensure that we get a total order for memory events.

Lines $10-17$ encode the load-store semantic constraints. For each event e_i we define two sets: Follower to the Same Memory Location ($FSML_i$) and Follower to the Same Memory Location with the Same Value ($FSML_i^{val}$). Let $e_i = (v, val_{old}, val_{new})$ be an event in the processor core p, $e_j = (v', val'_{old}, val'_{new}) \in FSML_i$ iff (1) $v = v'$, and (2) if e_j is also an event in p, e_j is the immediate follower to e_i that access v. $FSML_i^{val} \subseteq FSML_i$ and for any $e_j = (v', val'_{old}, val'_{new}) \in FSML_i^{val}$, $val_{new} = val'_{old}$. A memory event $e_i \in E_p$ can have an empty $FSML_i$ or $FSML_i^{val}$ if e is the last memory access to a location v in the processor core p.

We provide the load-store constraints for memory events e_1 and e_7 for the example on the left in Figure 2. For e_1 that accesses $x1$, we have $FSML_1 = FSML_1^{val} = \{e_3, e_5\}$. Its constraint is $(O_1 < O_3 \wedge (O_5 < O_1 \vee O_3 < O_5)) \vee (O_1 < O_5 \wedge (O_3 < O_1 \vee O_5 < O_3))$, which can be simplified to $(O_5 < O_1 \vee O_3 < O_5) \vee (O_1 < O_5 \wedge O_5 < O_3)$ because $O_1 < O_3 \equiv true$. For e_7 that accesses y_7, we have $FSML_7 = \{e_2\}$ and $FSML_7^{val} = \emptyset$. Its constraint is $O_2 < O_7$, because y's final state value matches the new value of $y7$ (lines 13-14).

The SMT formula is then given to the Yices SMT solver [9]. 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.

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 [10], which is the state-of-the-art hardware solution for recording shared-memory dependencies.

5.1 Evaluation Methodology

We use Simics [15] as the front end for full system functional simulation, and our cycle accurate simulator as the back end. We analyze four processor configurations: 2,4,8, and 16 cores. We model MESI coherence protocol. We model a ring network for the 2,4,8-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

Algorithm 1 ENCODING(STRATASET E, FINALSTATE D)

1: **for** $p = 1; p \leq P; p++$ **do**
2: Let $E_p = \langle e_m, e_{m+1}, \ldots, e_n \rangle$;
3: add constraint $O_m < O_{m+1} < \ldots < O_n$;
`//Program order constraint`
4: **end for**
5: **for all** $(e_i, e_j) \in E \times E$ **do**
6: **if** $processor(e_i) \neq processor(e_j)$ **then**
7: add constraint $O_i \neq O_j$;
`//Uniqueness constraint`
8: **end if**
9: **end for**
10: **for all** $e_i = (x, i_{old}, i_{new}) \in E$ **do**
11: let $FSML_i$ be the FSML set of e_i
12: $\mathcal{C}_i = \bigvee_{e_f \in FSML_i} (O_i < O_f \wedge \bigwedge_{e_k \in FSML_i^{val} \wedge e_f \neq e_k} (O_k < O_i \vee O_f < O_k))$;
`// Load-store semantic constraint`
13: **if** $x = i_{new} \in D$ and e_i is the last access to x in $p = processor(e_i)$ **then**
14: $\mathcal{C}_i = \mathcal{C}_i \vee \bigwedge_{e_f \in FSML_i} (O_f < O_i)$;
15: **end if**
16: add constraint $\mathcal{C}_i$
17: **end for**

Table 1. We picked a representative set of benchmarks from various benchmark suites, which are listed in Table 2. All the following experiments are performed on 8 core configuration, unless not explicitly mentioned.

5.2 Strata Region Length

Our first analysis is to determine the appropriate length for the Strata regions to bound the search space of offline analysis. As discussed in Section 4.4, we eliminate local, read-only, and uncacheable memory accesses in each Strata region, and only analyze *unfiltered* memory accesses that is left after filtering. Filtering the local accesses and read-only accesses within a Strata region eliminates over 99% of memory accesses from offline analysis. Figure 3(a) shows this result for a configuration where the Strata regions are constructed when 10,000 cycles has elapsed. Here, `Swim` show the most portion of unfiltered accesses, which was 0.4%.

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 some threshold. Figure 3(b) shows the time taken (y-axis uses a log scale) to analyze Strata regions with different numbers of unfiltered accesses. This includes the execution of all test applications where the Strata regions are constructed with rough cycle bound. The result shows the exponential increase of offline analysis cost. Therefore, we would like to make most of Strata regions have less than 500 unfiltered accesses and prevent Strata regions from containing more than 1000 unfiltered accesses.

Based on this observation, we experimented with four different processor cycle bounds. Figure 4 shows the distribution of Strata intervals for each bound. Each Strata interval would have different number of unfiltered memory accesses depending on the program characteristics, and the intervals are classified over four ranges. Figure 4 shows that for programs like `Fmm` 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 `Swim` 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 the offline analysis overhead (the result is discussed later in Section 5.3). Since we filtered out local and read-only accesses, the number of unfiltered accesses reflects how many read/write sharing has been occurred. Moreover, the number of cache misses within a Strata includes the effect of capacity misses. Therefore, downgrade counts can work better than the cache miss counts. Figure 6(a) and Figure 6(b) plot the number of unfiltered accesses versus the cache miss count and downgrade count respectively for all applications with the cycle bound of 10,000. 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 5 shows results for downgrade bound approach for four different d values. For example, for `barnes` and `swim`, the downgrade bound approach reduce the portion of intervals with greater than 1000 unfiltered accesses.

5.3 Strata Log Size and Symbolic Analysis Performance

Our second analysis is to see how each cycle or downgrade bound affects Strata hint log size and offline analysis overhead. 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). Figure 7 shows the log size in mega-bytes and offline analysis time in seconds per one second of program execution with varying bound configurations on applications `Barnes, Fmm, Wupwise` and `Apache`. The average bars include the result of all test applications including these four. On average, we can see the expected pattern that with lower cycle bound the log size increases, but offline analysis cost decreases. Moreover, with lower downgrade bound, the log size also increases, but offline overhead decreases. The result was anticipated in that reducing such bounds will generate the Strata region with less number of unfiltered accesses, leading to make symbolic analyzer easier to find a valid total order.

However, there are more interesting points on the Figure 7. First of all, `Fmm` with cycle bound of 100,000 and 10,000 shows that offline analysis overheads are almost same. There are two reasons. As can be seen in the Figure 4, there is no big difference in Strata distribution between cycle bound of 100,000 and 10,000. Furthermore, with smaller cycle bound, we need to analyze more number of Strata, which offsets the benefit of analyzing smaller and simpler Strata. Second, `Barnes` shows the effectiveness of downgrade bound compared to cycle bound. Figure 4 show that with cycle bound of 100,000 on `Barnes`, around 30% of Strata contain more than 1000 unfiltered accesses, which causes high overhead on offline analysis. Here, we can reduce the cycle bound or apply the downgrade bound. Figure 7 shows that cycle bound approach reduce offline overhead by 20x at the cost of 10x log size, whereas downgrade bound of 10 reduces offline overhead by 800x while increasing the log size only by 7x. Third, `Wupwise` shows

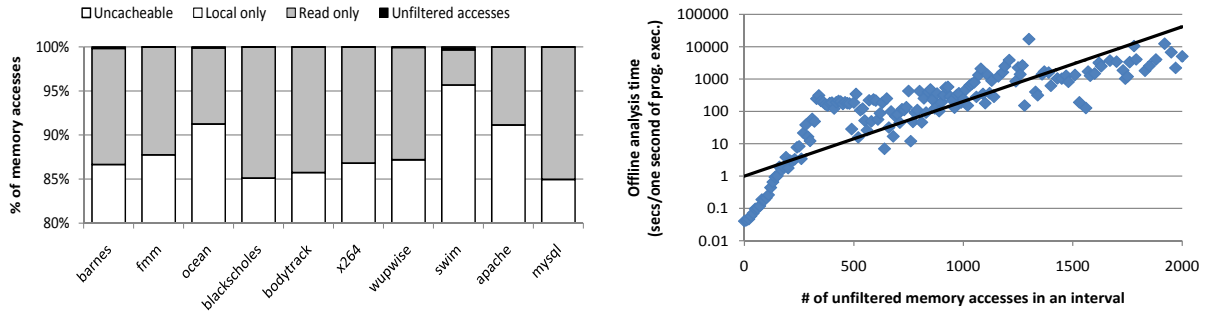

Figure 3: Effectiveness of filtering local and read-only accesses & Scalability of offline analysis.

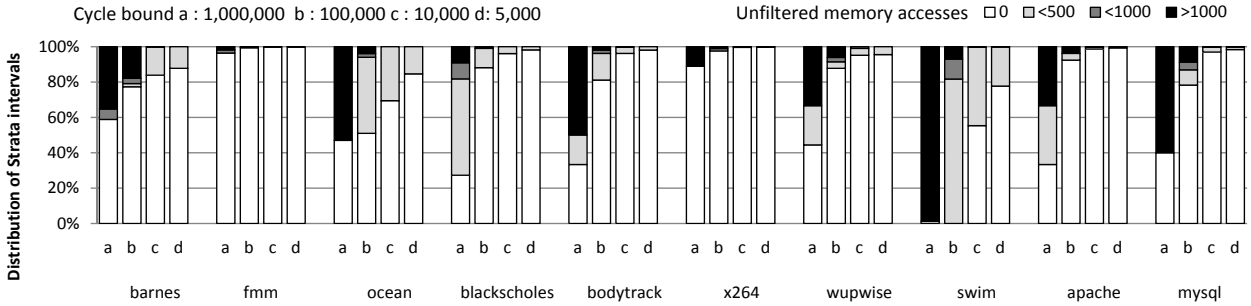

Figure 4: Distribution of unfiltered memory events in a Strata interval for cycle bounds.

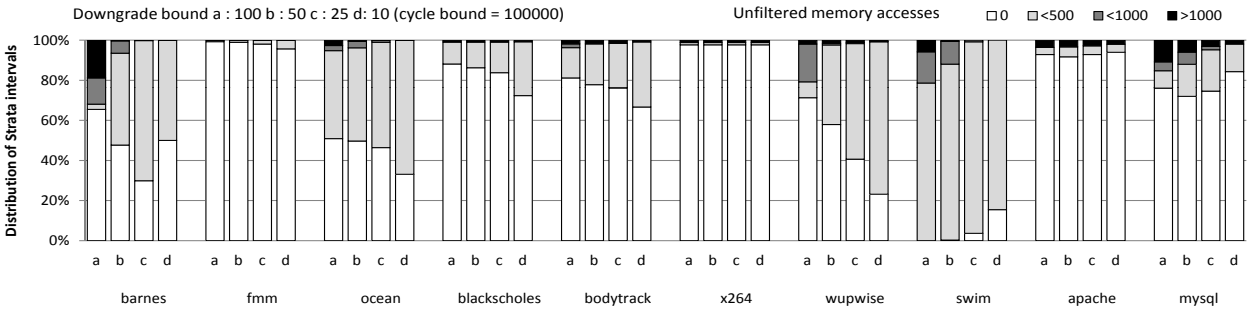

Figure 5: Distribution of unfiltered memory events in a Strata interval for downgrade bounds.

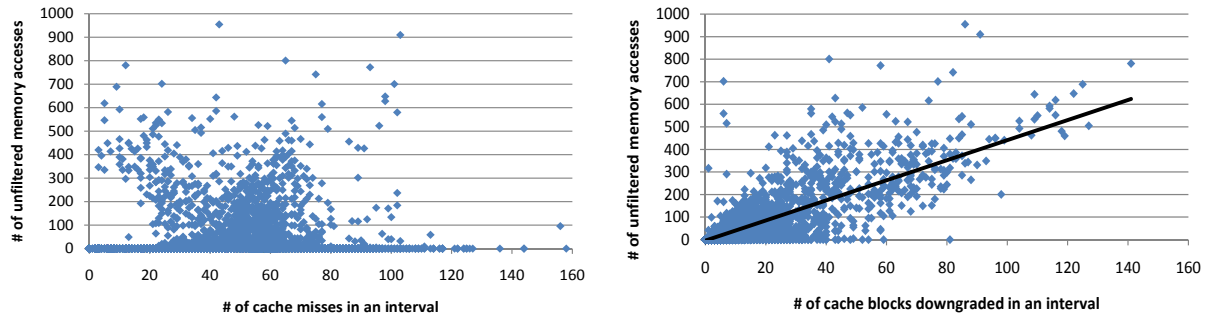

Figure 6: Correlation between the number of unfiltered accesses and cache miss counts (on the left), and downgrade counts (on the right)

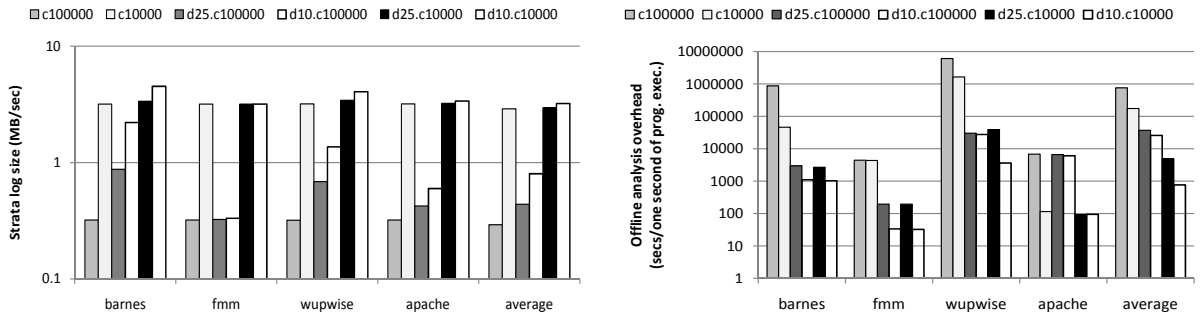

Figure 7: Strata log size and offline analysis overhead for different bounds

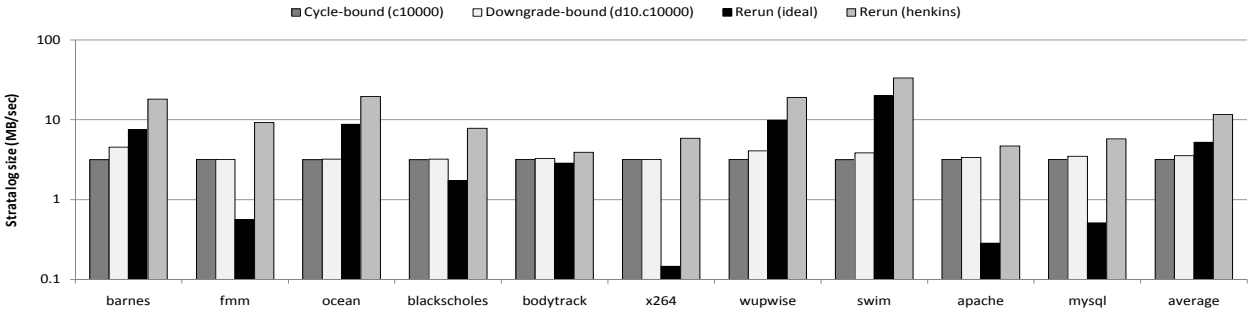

Figure 8: Strata log size

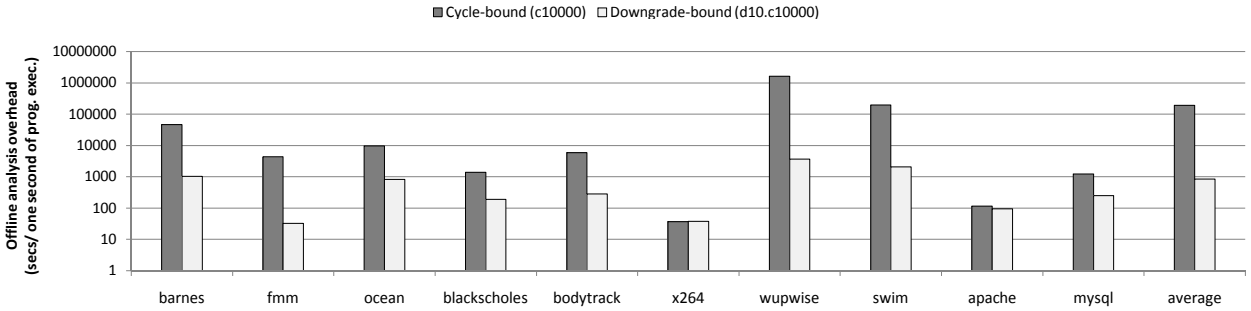

Figure 9: Offline analysis overhead

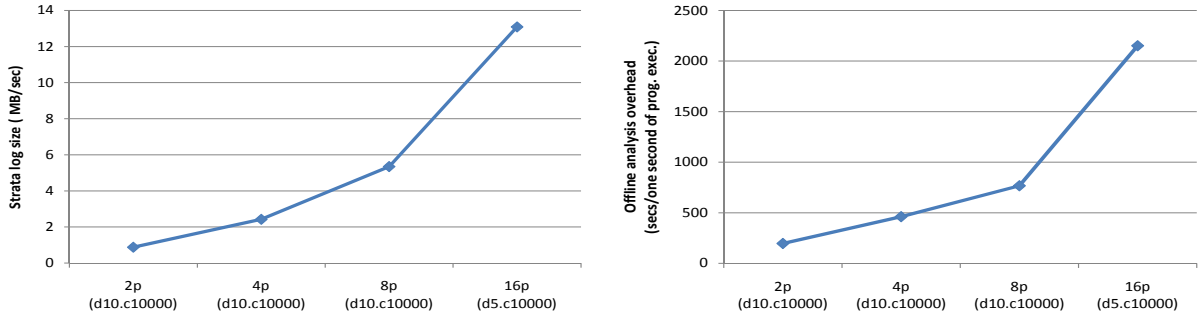

Figure 10: Strata log size and offline analysis overhead for different number of cores

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

Programs	Description
SPLASH 2	SPLASH is a suite of parallel scientific workloads. We consider `barnes, fmm, ocean` from this suite. Each benchmark executes same number of worker threads as number of cores. Fast forwarded up to second barrier synchronization point. Trace collected for 500 million instructions.
Parsec 2.0	Parsec is a new benchmark suite for CMP. We present the results from `blackscholes, bodytrack` and `x264`. Each benchmark runs same number of worker threads as number of cores. Fast forwarded up to second barrier synchronization point except x264. Fast forwarded up to the middle of encoding for x264. Trace collected for 500 million instructions.
SPEComp	We use SPEComp2001 as another representative workload. We present the results from `wupwise` and `swim`. OMP_NUM_THREADS is set to the same number of cores. Fast forwarded up to second OMP parallelization point. Trace collected for 500 million instructions.
Apache	We use Apache 2.2.13 with MPM worker (multi-threaded) configuration. We use SURGE [4] to generate web requests to the repository of 20,000 files (totaling 480 MB). We simulate 400 clients, each with 25 ms think time between requests. Trace collected for 500 million instructions.
MySQL	We use MySQL 5.1.37 with –with-pthread configuration to force pthread based parallelization. We create a table with one million records and use SysBench [2] to generate database requests. We test on OLTP mode with 16 threads and allow every type of queries. Trace collected for 500 million instructions.

Table 2: Benchmarks

the highest overhead on offline analysis among all test applications. Upon further investigation, we found that `Wupwise` has more fine-grained sharing than other benchmarks. Most Strata consist of multiple variable sharing with read/write patterns, leading to high downgrade rates. Those fine-grained multiple variable sharing causes offline analyzer to meet lots of conflicts when finding a valid total order assignment. Since the downgrade rates are high, similar to `Barnes`, we could see 1600x offline performance improvement on downgrade bound approach. More in detail, we also could observe that d25 and d10 does not make a big difference with cycle bound of 100,000. In `Wupwise` there was a Strata where one producer updates the value once and multiple consumers (here we have seven consumers) read the value, which give rise to the downgrade count to be one for each core, which is smaller than the downgrade bounds. This Strata contains fairly large number of unfiltered memory accesses and dominates overall offline analysis time. Forth, for some applications like `Apache`, cycle bound effectively generates Strata with small number of unfiltered accesses. `Apache` shows different pattern from `Wupwise` in that it shows better offline performance on cycle bound method, but also shows the same pattern on downgrade bound approaches as `Wupwise`.

Figure 8 and Figure 9 show the Strata log size and symbolic analysis overhead over all test applications with cycle bound and downgrade bound. We need 2.89 and 3.21 mega-bytes respectively to record Strata hints for one second of program execution in the 8-processor configuration. We also show the memory race log size for ReRun [10], 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 Henkins' hash function. ReRun requires a log that is 4 times larger. But, more importantly, our approach is complexity-effective. Offline analysis takes about 1.2 days to analyze a second of program execution on average for cycle bound of 10000. This cost is paid only once before replay. Once analyzed, the replayer need not incur this analysis cost, and therefore can be interactive. On most applications except `Wupwise` and `Swim`, the results show that the simpler cycle-bound approach is sufficient.

All the previous analysis have been done with 8 core configuration. Figure 10 shows the results from different hardware configurations. With more number of cores, we need proportionally increased amount of Strata log because every core should keep the memory count when a Strata is created. Moreover, in order to bound the search space of offline symbolic analysis, it is required to assign the lower bound for Strata creation. The results show that our approach can be tuned for more number of cores.

5.4 Input Log Size and 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 which the packets that write-back recorded logs are given a lower priority in the routers. Table 3 shows input log size and the performance degradation on 8 core configurations. On average, we need 292 mega-bytes of memory to record the memory count and cache blocks fetched on misses for one second of program execution. `Swim` shows the highest cache miss rates and requires largest input log size. The worst degradation is also for `Swim` (1.57% slowdown). The priority optimization reduces the overhead to 1.48%. On average, the non-prioritized scheme incurs 0.35% slowdown, whereas prioritized scheme incurs 0.29% overhead.

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

application	input log size (MB/sec)	baseline average IPC	slowdown without priority	slowdown with priority
barnes	505.08	10.44	0.43%	0.33%
fmm	219.84	13.04	0.01%	0.01%
ocean	432.91	4.31	0.21%	0.2%
blackscholes	198.13	12.61	0.93%	0.59%
bodytrack	122.66	11.56	0.17%	0.15%
x264	147.21	13.71	0.03%	0.03%
wupwise	186.31	13.48	0.03%	0.02%
swim	884.26	1.74	1.57%	1.48%
Apache	77.62	13.34	0.04%	0.03%
MySQL	142.86	13.59	0.03%	0.02%
average	291.69	10.78	0.35%	0.29%

Table 3: Input log size and recording performance

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.

Acknowledgments

We thank Reetuparna Das for helping us with the architectural simulator and workload setup. We also thank Brad Calder and Madan Musuvathi for the valuable discussions. This work was funded in part by NSF with grants CCF-0916770, CNS-0905149, CCF-0811287, a Microsoft gift and an equipment grant from Intel.

References

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

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

[3] 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*, pages 194–206. ACM Press, 1991.

[4] 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*.

[5] 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.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ISCA*, pages 278–289, 2007.

[7] J. D. Choi, B. Alpern, T. Ngo, and M. Sridharan. A perturbation free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4):471–482, 1987.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] S. Narayanasamy, C. Pereira, and B. Calder. Software profiling for deterministic replay debugging of user code. In *5th International Conference on Software Methodologies, Tools and Techniques (SoMET)*, Oct 2006.

[21] 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.

[22] 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.

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

[24] 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*, pages 111–122. IEEE Computer Society, 2002.

[25] 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*, pages 123–134, 2002.

[26] 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*, pages 29–44, 2004.

[27] 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.

[28] 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*, pages 49–60, 2006.

[29] 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*.