QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs*

Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich
Intel Corporation
{gilles.a.pokam, klaus.danne, cristiano.l.pereira, rolf.kassa, tim.kranich, shiliang.hu, justin.e.gottschlich}@intel.com

Nima Honarmand, Nathan Dautenhahn, Samuel T. King, Josep Torrellas
University of Illinois at Urbana-Champaign
{honarma1, dautenh1, kingst, torrella}@illinois.edu

ABSTRACT

There has been significant interest in hardware-assisted deterministic Record and Replay (RnR) systems for multithreaded programs on multiprocessors. However, no proposal has implemented this technique in a hardware prototype with full operating system support. Such an implementation is needed to assess RnR practicality.

This paper presents *QuickRec*, the first multicore Intel Architecture (IA) prototype of RnR for multithreaded programs. QuickRec is based on *QuickIA*, an Intel emulation platform for rapid prototyping of new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and *Capo3*, a full software stack for managing the recording hardware from within a modified Linux kernel.

This paper's focus is understanding and evaluating the implementation issues of RnR on a real platform. Our effort leads to some lessons learned, as well as to some pointers for future research. We demonstrate that RnR can be implemented efficiently on a real multicore IA system. In particular, we show that the rate of memory log generation is insignificant, and that the recording hardware has negligible performance overhead. However, the software stack incurs an average recording overhead of nearly 13%, which must be reduced to enable *always-on* use of RnR.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - MIMD Processors; C.4 [**Performance of Systems**]: Design Studies; C.0 [**General**]: Hardware/software interfaces

Keywords

Deterministic Record and Replay, Shared Memory Multiprocessors, Hardware-Software Interface, FPGA Prototype.

*This work is supported in part by the Illinois-Intel Parallelism Center (I2PC).

ISCA'13 Tel Aviv, Israel

1. INTRODUCTION

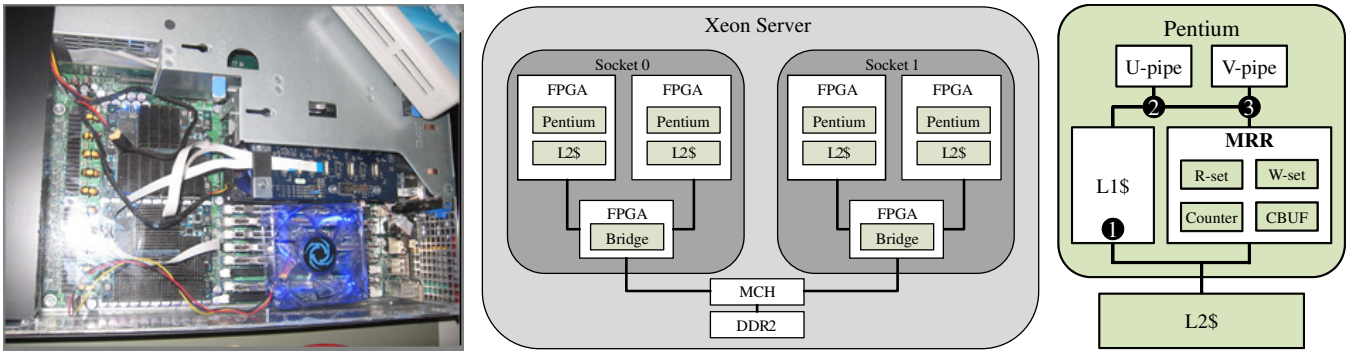
Deterministic Record and Replay (RnR) of multithreaded programs is an appealing mechanism for computer systems builders. RnR can recreate past states and events, by recording key information while a program runs, restoring to a previous checkpoint, and replaying the recorded log to force the software down the same execution path. With this mechanism, system designers can debug applications [1, 4, 6, 17, 32, 34, 41], withstand machine failures [5], and improve the security of their systems [15, 16].

To replay a program, an RnR system must capture all sources of non-determinism. For multithreaded programs running on multicores, there are two key sources of non-determinism. The first is the inputs to the execution, such as effects and return values of system calls or occurrence of signals. The second is the order of the inter-thread communications, which manifests as the interleaving of the inter-thread data dependences through the memory system. While the first source of non-determinism can be captured in software with relatively low overhead, doing the same to record the second source typically imposes significant slowdowns.

To record memory access interleaving with low overhead, researchers have proposed several hardware assisted RnR designs (e.g., [3, 7, 12, 13, 23, 24, 25, 26, 30, 31, 36, 39, 40]). These proposals have outlined RnR systems that have negligible overhead during execution recording and can operate with very small log sizes. To evaluate these systems, the authors typically implement their techniques in software-based simulators. In addition, they typically run their simulations without an operating system that manages and virtualizes their special hardware. The exceptions are LReplay [7], which extends and simulates the RTL (Register Transfer Level) model of a chip multiprocessor and does not discuss system software issues, and Capo [24] and Cyrus [12], which use an RnR-aware operating system on top of simulated hardware.

Although this evaluation approach helps assess the efficacy of the proposed algorithms, it ignores practical aspects of the design, such as its integration with realistic cache coherence hardware, coping with relaxed memory models, and virtualizing the recording hardware. In addition, promoting RnR solutions into mainstream processors requires a co-design with the system software that controls the hardware, and omitting software effects from the evaluation presents only part of the overall performance picture.

To evaluate the practical implementability of hardware-assisted RnR, we have built *QuickRec*, the first multicore IA-based prototype of RnR for multithreaded programs. QuickRec is based on *QuickIA* [37], an Intel emulation platform for rapid prototyping of



(a) (b) (c)

Figure 1: Photograph of the QuickRec prototype with FPGAs in CPU sockets (a); architecture of the QuickIA processor-emulation platform (b); and architecture overview of the extended Pentium core in QuickRec, where circled numbers identify the main CPU *touch points* required to enable recording (c).

new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and *Capo3*, a full software stack for managing the recording hardware from within a modified Linux kernel.

This paper focuses on identifying and characterizing RnR-related implementation issues. Specifically, we describe how QuickRec records the memory access interleaving of threads, and how to integrate this support into a commodity IA multicore. We discuss subtle issues related to capturing the ordering of instructions with multiple memory accesses, and the interaction with the memory consistency model. We also discuss how Capo3 records the inputs to processes, manages the replay logs, and virtualizes the hardware components. We provide data characterizing QuickRec's recording performance and log parameters. Overall, our evaluation demonstrates that RnR can be practical for real IA multicore systems.

This effort has led to some lessons learned, as well as to some pointers for future research directions. In particular, we find that the main challenge of RnR systems is to take into account the idiosyncrasies of the specific architecture used, such as single instructions producing multiple memory transactions. Further, we find that the software stack has a dominant role in the overall system performance, as it manages the logs. Based on these experiences, we suggest focusing future research on recording input events efficiently, and on replay techniques that are tolerant of the micro-architectural details of the system.

The main contributions of this work are the following:

- The implementation of the first IA multicore prototype of RnR for multithreaded programs. The prototype includes an FPGA design of a Pentium multicore and a Linux-based full software stack.
- A description of several key implementation aspects. Specifically, we show how to efficiently handle x86 instructions that produce multiple memory transactions, and describe the elaborate hardware-software interface required for a working system.
- An evaluation of the system. We show that the rate of memory log generation is insignificant, given today's bus and memory bandwidths. In addition, the recording hardware has negligible performance overhead. However, the software stack incurs an average recording overhead of nearly 13%, which must be reduced to enable always-on use of RnR.

This paper is organized as follows: Section 2 introduces the QuickRec recording hardware; Section 3 describes the Capo3 system software; Section 4 characterizes our prototype; Section 5 discusses using replay for validation; Section 6 outlines related work; Section 7 describes lessons learned; and Section 8 concludes.

2. QuickRec RECORDING SYSTEM

The QuickRec recording system prototyped in this work is built on a FPGA processor-emulation platform called QuickIA. This section introduces QuickIA and then describes the changes we added to support RnR. Figure 1a shows a picture of the QuickRec recording system testbed.

2.1 QuickIA Processor Emulation Platform

The QuickIA processor emulation platform [37] is a dual-socket Xeon server board in which Xeon CPUs are substituted with FPGA modules from XtreamData [38]. Each such FPGA module is composed of two Compute FPGAs and one Bridge FPGA, as shown in Figure 1b. Each Compute FPGA implements a second-generation Pentium core with private L1 and L2 caches. The Bridge FPGA implements the interconnect between the two Compute FPGAs and the Intel Front Side Bus (FSB), which connects the two CPU sockets to the Memory Controller Hub (MCH) on the platform. This allows both CPU sockets to be fully cache coherent, with full access to memory and I/O. The QuickIA system implements a MESI coherence protocol with L2 as the point of coherence.

The Pentium cores used in the QuickIA emulation platform are fully synthesizable. Each core features a dual-pipeline in-order CPU with floating-point support. In addition, each core is extended with a set of additional features to reflect the state of the art of modern processors. These changes include L1 cache line size increase to 64 bytes, Memory Type Range Registers, physical address extension, and FSB xAPICs.

The four emulated Pentium cores run at 60MHz. While this clock frequency is low, the memory bandwidth is also low (24MB/s), which means that the ratio between CPU speed and memory bandwidth is similar to that of today's systems. The QuickIA system includes 8GB of DDR2 memory and basic peripherals (network, graphics card and HDD), and can boot a vanilla SUSE Linux distribution. The basic platform parameters are shown in Table 1.

2.2 Recording Interleaving Non-Determinism

To record the non-determinism of memory access interleaving, the RTL of the synthesizable Pentium core is augmented to capture the order of memory accesses. This support includes mechanisms to break down a thread's execution into *chunks* (i.e., groups of consecutive dynamic instructions), and then order the chunks across cores. A significant effort was invested in integrating this support into the Pentium core without adding unnecessary complexity. Some of the main challenges we faced include dealing with the IA memory model, and coping with x86 instructions with multi-

Cores	4 Pentium cores
Clock	60MHz
L1 data cache	32KB, private, WB, 8-way assoc, 64B line size, 1-cycle latency
L2 cache	512KB, private, WB, 16-way assoc, 64B line size, 4-cycle latency
Coherence	MESI
Memory	8GB DDR2, 24MB/s bandwidth (measured by STREAM [22]), 90-cycle round-trip latency

Table 1: QuickIA platform parameters.

ple memory accesses. The extended Pentium core is then synthesized and downloaded into FPGAs to boot up the QuickRec emulation platform. A high-level overview of the extended Pentium core is shown in Figure 1c. In the figure, the *Memory Race Recorder (MRR)* box implements the functionality for recording memory access interleaving, while the circled numbers indicate the CPU *touch points* required to enable it.

2.2.1 Capturing and Ordering Chunks

The QuickRec recording system implements a mechanism similar to the Intel MRR [30] to divide a thread's execution into chunks. It adds Bloom filters next to the L1 cache to capture the read and write sets of the memory accesses in a chunk (*R-set* and *W-set* in Figure 1c). The line addresses of the locations accessed by loads and stores are inserted into their respective set at retirement and at global observation time, respectively. A thread's chunk is terminated when the hardware observes a memory conflict (i.e., a data dependence) with a remote thread. Conflicts are detected by checking the addresses of incoming snoops against addresses in the read and write sets. When a conflict is detected, a counter (*Counter* in Figure 1c) with the current chunk size is logged into an internal chunk buffer (*CBUF* in Figure 1c), along with a timestamp that provides a total order of chunks across cores. The chunk-size counter counts the number of retired instructions in the chunk. After a chunk is terminated, the read and write sets are cleared, and the chunk-size counter is reset.

In addition to terminating a chunk on a memory conflict, QuickRec can be configured to terminate a chunk when certain system events occur as well, such as an exception or a TLB invalidation. A chunk also terminates when the 20-bit chunk-size counter overflows. Additionally, the addresses of lines evicted from L2 are looked up in the read and write sets and, in case of a hit, the chunk also ends. This is done because the read and write sets would not observe future coherence activity on these evicted lines. Further information on chunk termination is provided in Section 2.3.

Figure 1c shows the main CPU touch points required to enable the chunking mechanism described above. The first CPU touch point is hooked-up to the external L1 snoop port to allow snoops to be forwarded to the MRR for address lookups. The second and third CPU touch points are hooked-up to the U and V integer execution pipelines of the Pentium core. They provide diverse functionalities, such as forwarding load and store line addresses to the MRR for insertion into the read and write sets, and forwarding the instruction retirement signal to the MRR to advance the chunk-size counter.

One of the complexities we encountered when integrating the chunking mechanism into the Pentium core was keeping updates to the read and write sets within one cycle, so that they can be performed in parallel with a cache access. The problem is that only the lower bits of the addresses are available at the beginning of a cache cycle, as the upper bits (tag bits) are provided usually late in the cycle, after a DTLB access. To preserve a single cycle for the read and write set update, addresses (tag plus set bits) are buffered into a latch stage before they are fed to the Bloom filter logic. To compensate for the delayed update of the read and write sets, these buffers are also looked-up on external snoops, at the cost of additional comparators for each address buffer.

2.2.2 Integration into the IA Memory Model

The IA memory model allows a load to retire before a prior store to a different address has committed, hence effectively ordering the load before the prior store in memory. This memory model is called Total Store Order (TSO). In this situation, using the retired instruction count is not sufficient to guarantee that loads and stores are ordered correctly during replay. This is because, during replay, instructions are executed in program order. Hence, regardless of when the store committed to memory during the recorded execution, the store is evaluated before the load during replay. To address this problem, QuickRec implements a solution similar to the one proposed in CoreRacer [31] to handle TSO. The idea is to track the number of pending stores in the store buffer awaiting commit and, at chunk termination, append the current number to the logged entry. This number is called the Reordered Store Window (RSW) count. The MRR is hooked-up to the memory execution unit to enable this functionality.

2.2.3 Instruction Atomicity Violation

In the x86 ISA, an instruction may perform multiple memory accesses before completing execution. For instance, a split cache line access, which is an access that crosses a cache line boundary, requires more than one load or store operation to complete. In addition, some complex instructions require several memory operations. For example, the increment instruction (INC) performs a load and a store operation. At the micro-architecture level, these instructions are usually broken down into multiple micro-operations or μops. An Instruction Atomicity Violation (IAV) occurs if an event causes the QuickRec recording system to log a chunk in CBUF in the middle of such an instruction execution. An example of such an event is a memory conflict. Because software is usually oblivious of split cache line accesses and μop execution, IAVs make it difficult for software to deterministically reproduce a program execution.

Figure 2 shows an example. Thread T0 executes instruction INC A, which increments the value in memory location A. The instruction breaks down into the three μops shown in the figure: a read from A into user-invisible register r_{tmp}, the increment of r_{tmp}, and the store of r_{tmp} into A. At the same time, thread T1 writes A. Suppose that the operations interleave as shown in the time line.

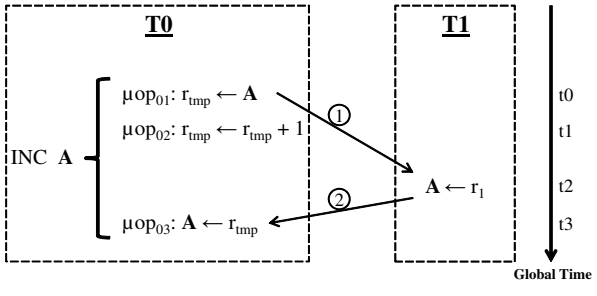


Figure 2: Instruction atomicity violation (IAV) example.

When the store in T1 executes at time t2, a conflict with T0 is detected, since μop_{01} has read from the same address at t0. Therefore, QuickRec terminates the chunk in T0 and logs an entry in T0's CBUF. This chunk is ordered before the store in T1. However, since the INC instruction has not yet retired, INC is not counted as belonging to the logged chunk. Then, when the INC instruction executes μop_{03} and retires at t3, a conflict with T1 is detected. This causes QuickRec to terminate the chunk in T1 and log an entry in T1's CBUF that contains the store. The logged chunk is ordered before the currently-executing chunk in T0, which is assumed to include the INC instruction. Consequently, in this naive design, the replay would be incorrect. Indeed, while during recording, μop_{01} occurred before the store in T1, which in turn occurred before μop_{03}, during replay, the store in T1 will be executed before the whole INC instruction.

This problem occurs because the INC instruction suffers an IAV. Although the instruction has performed some memory transactions during the earlier chunk in T0, since the instruction has not retired when the chunk in T0 is logged, the instruction is counted as belonging to the later chunk in T0.

The QuickRec recording system solves this problem by monitoring the retirement of the multiple memory accesses during the execution of the instruction. Specifically, it uses a dedicated IAV counter to count the number of retired memory transactions for a multi-line or multi-operation instruction (Figure 3). The IAV counter is incremented at every retired memory transaction, and is reset when the instruction retires. At chunk termination, if the IAV counter is not zero, the current instruction has not retired, and an IAV has been detected. In this case, QuickRec saves the value of the IAV counter in the log entry of the terminated chunk. Since, during replay, we know exactly the number (and sequence order) of the memory transactions that need to occur in a given instruction, by reading the IAV counter and examining the RSW count (Section 2.2.2), we know how many memory operations of the subsequent instruction need to be performed before completing the current chunk. In our actual implementation, the IAV counter is incremented by 1 for each access in a split cache line reference, and by 2 for any other access. With this design, an odd counter value indicates that the chunk terminated between the accesses of a split cache line reference.

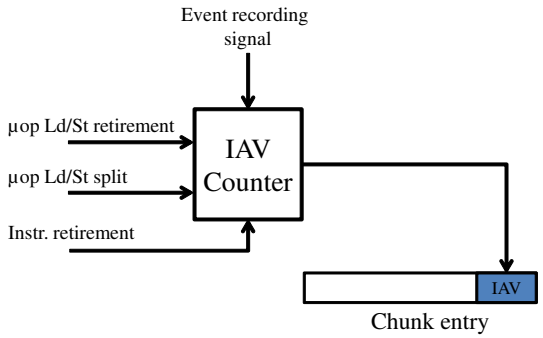


Figure 3: IAV counter mechanism.

Consider again the example of Figure 2. When T1 executes the store at time t2 and a conflict is detected in T0, the INC instruction has not yet retired. The IAV counter in T0 is 2, since the only retired access is that of μop_{01}. Therefore, an IAV is detected. The QuickRec recording system terminates the chunk in T0 and, as it logs the chunk, appends to it the value of the IAV counter. This log entry conveys to the replayer the information that an IAV has occurred in the chunk and that only the first memory µop had retired at the time of chunk termination.

Instruction atomicity violation was first introduced in [29] and then described in [31]. The main difference with [31] is that we log the number of retired memory transactions instead of the number of transferred bytes. The advantage of logging memory transactions over transferred bytes is the reduction in the log size.

2.2.4 *Log Management*

CBUF is organized into four entries, where each is as large as a cache line. When a chunk terminates, a 128-bit chunk packet is stored in CBUF. When a CBUF entry is full, it is flushed by hardware to a dedicated memory region called CMEM. To minimize the performance impact, this is done lazily, during idle cycles, by bypassing the caches and writing directly to memory. Occasionally, however, the chunking mechanism must stall the execution pipeline to allow CBUF to drain to CMEM to avoid overflow.

There are two main packet types inserted into CBUF, namely the timestamp packet (TSA) and the chunk packet. Both are very conservatively sized as 128-bit long. Once a TSA is logged for a thread, subsequent chunk packets for that thread only need to log the timestamp difference (TSD) with respect to the last TSA. The TSA is then logged again when the value in TSD overflows. Note that this also causes a chunk termination. Figure 4 shows the format of these two packets. The chunk packet contains the TSD, chunk size (CS), and RSW and IAV counts. It also contains a *Reason* field, which indicates why the chunk was terminated — e.g., due to a RAW, WAR or WAW conflict, an exception, or a chunk-size overflow. Table 2 lists the main reasons for terminating chunks.

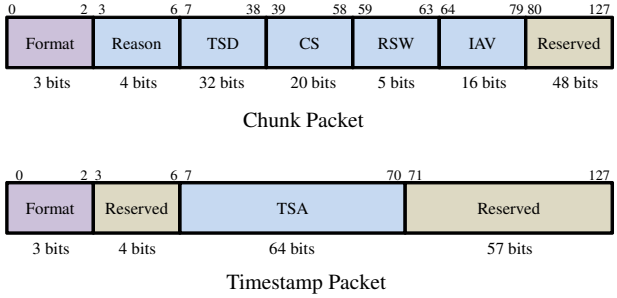


Figure 4: Packet formats in QuickRec.

Type	Reason
RAW	RAW conflict between chunks
WAR	WAR conflict between chunks
WAW	WAW conflict between chunks
WAB	Both WAR and WAW conflicts between chunks
EXCEPT	Exception, interrupt, far call, or far return
EVICT	Line eviction from L2 that hits the R-set or W-set
CS_OVERFLOW	Chunk size overflow
TLBINV	TLB invalidation
XTC	Explicit chunk termination instruction

Table 2: Main reasons for terminating chunks. WAB (Write-After-Both) is when a write in one chunk hits in both the read and the write set of another chunk.

2.3 Programming Interface

The QuickRec recording system contains a set of registers to configure and program the hardware. For instance, using these registers, the hardware can be programmed to record memory non-determinism for user-level code only, or for both user- and system-level code. It can also be programmed to terminate a chunk under certain conditions only, such as a specific type of conflict or exception. Privileged software can also specify where in memory the logs are written for each recorded thread. The QuickRec recording system also has a status register that is updated at chunk termination time to capture the state of the machine at that point. Among other information, it captures the reason for the chunk termination. Some of its information is copied to the Reason field of the logged chunk packet. A more detailed discussion of the programming interface, and how the system software uses it to manage the QuickRec hardware is provided in Section 3.3.

QuickRec extends the ISA with two new instructions: one that terminates the current chunk (*XTC*), and one that terminates the current chunk and flushes CBUF to memory (*XFC*). The use of these two instructions is restricted to privileged software. Examples of their use are discussed in Sections 3.4 and 3.6.

2.4 Other Issues

Because the main purpose of this work is to demonstrate the feasibility of hardware-assisted RnR, this prototype only addresses the issues that are critical to support RnR for the majority of programs. For instance, the prototype only supports Write-Back (WB) memory [14], which constitutes the majority of memory accesses in current programs. Memory accesses to Uncacheable (UC) or Write-Combining (WC) memory are not tracked, and cause the system to terminate a chunk. Chunking is resumed when the next access to WB memory occurs.

In some cases, the IA memory model allows accesses to WB memory to have different ordering semantics than TSO. For instance, in fast string operations, a store to WB memory can be reordered with respect to a prior store. To ensure that QuickRec's RSW and IAV support work properly, we disable this feature, so that all loads and stores obey TSO semantics.

Although we do not discuss how to extend our mechanisms to support Hyperthreading, the changes required to do so are minimal. In modern IA cores, there already exist mechanisms for detecting conflicts between the different hardware thread contexts sharing the same cache. Therefore, in order to enable RnR on a Hyperthreaded core, one would only need to replicate certain resources for each hardware thread context (e.g., the read and write sets).

3. Capo3 SYSTEM SOFTWARE

To manage the QuickRec hardware, we built a software system called *Capo3*. Capo3 draws inspiration and borrows many of the concepts and principles from Capo [24], a system designed for hardware-assisted RnR. However, Capo3 must run on real hardware, and as such, we encounter several issues that were abstracted away in Capo due to using simulated hardware. In this section, we compare Capo3 with Capo, describe its architecture, and focus on several of its key aspects.

3.1 Comparing Capo3 with Capo

Capo3 uses some of the basic ideas introduced by Capo, including the *Replay Sphere* and the *Replay Sphere Manager* (RSM). The Replay Sphere abstraction is the single application (or a group of applications) that should be recorded/replayed in isolation from the rest of the system. The Replay Sphere Manager is a software component that is responsible for correctly capturing non-deterministic input and memory access interleaving.

Capo3 also uses the same basic techniques as Capo to record program inputs, including interactions between the operating system and processes (e.g., system calls and signals), and non-deterministic instructions (i.e., *rdtsc* and *cpuid*). Recording these input events guarantees that, during replay, the same data can be injected into the user-mode address space. However, some system calls also affect the kernel-mode data structures of the program. Hence, to ensure that their effects are deterministically recreated during replay, we re-execute these system calls during replay.

To correctly capture kernel state, like in Capo, the RSM enforces a *total order* of input events during recording. The same total order is enforced during replay. This total order has major performance and correctness implications, as shown in Sections 3.6 and 4.

Capo3 uses a different software architecture than Capo. Specifically, it places the bulk of the RnR logic in the kernel — whereas Capo used *ptrace* to capture key events with user-mode logic. Moreover, since Capo3 must virtualize real hardware, its design must support a hardware/software interface to enable context switches, record memory access interleaving when the kernel is running with interrupts enabled, and manage subtle interactions between QuickRec hardware and Capo3 software.

3.2 Capo3 Architecture

Capo3 implements the RSM as an extension to the Linux kernel. To record an execution, a driver program initializes a Replay Sphere using the RSM-provided interface. The RSM then logs the input events, sets-up the MRR hardware to log the memory access interleaving, and makes all these logs available to the driver program that is responsible for the persistent storage and management of the logs. Figure 5 shows the high-level architecture of the Capo3 software stack.

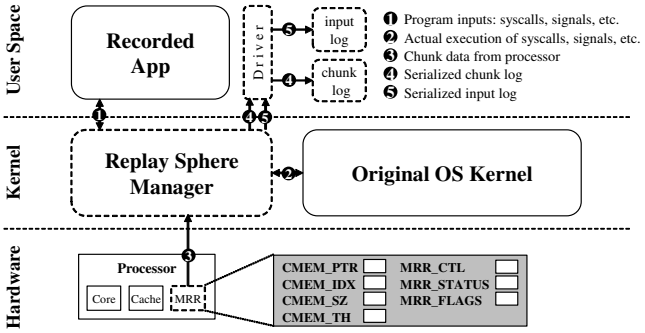


Figure 5: Overall architecture of Capo3. Dashed boxes indicate QuickRec-specific components.

Our decision to use a kernel-based implementation was driven by the observation that the Linux kernel has well-defined places to enable the kernel to interpose on processes. As a result, Capo3 only requires the kernel to be augmented in a few key places, so it can interpose on all system calls, signals, and memory copies between processes and the kernel. These changes also allow Capo3 to virtualize the QuickRec hardware by saving/restoring QuickRec state upon a context switch. Overall, our kernel-based implementation consists of roughly 3.4K lines of code, where the bulk of the code is dedicated to managing the logs, and is well isolated from the rest of the kernel.

There are four different sources of input non-determinism that the RSM captures: system calls, data copied to user-mode address

spaces, signals, and non-deterministic processor instructions. To bind these recorded events to their corresponding threads, the RSM assigns a unique R-Thread ID to each recorded thread. During replay, each thread is guaranteed to get the same R-Thread ID. These R-Thread IDs are also used to associate chunks recorded by the QuickRec hardware with their corresponding threads.

3.3 Virtualizing the QuickRec Hardware

To virtualize the QuickRec hardware, the RSM uses the programming interface outlined in Section 2.3. The main components of this interface are the seven registers shown in the lower level of Figure 5. Specifically, the Chunk Memory Pointer (*CMEM_PTR*) points to CMEM, which is the in-memory buffer that contains the logged chunk data. Each thread gets its own CMEM. The Chunk Memory Index (*CMEM_IDX*) indicates the location in CMEM where the next CBUF entry is to be written. This register is updated by hardware as CBUF entries are written to memory. The Size Register (*CMEM_SZ*) indicates the size of CMEM. The Threshold Register (*CMEM_TH*) indicates the threshold at which a CMEM overflow interrupt is generated. The Control Register (*MRR_CTL*) enables and disables chunking under certain conditions, while the Status Register (*MRR_STATUS*) provides the status of the hardware. These last two registers were described in Section 2.3. Finally, the Flags Register (*MRR_FLAGS*) controls kernel-mode recording and is discussed later.

It is the RSM's responsibility to manage the CMEM buffers and virtualize these hardware registers so that different threads can use the hardware without having their chunk data mixed-up. In particular, this involves: (i) ensuring that a valid CMEM pointer is configured before recording begins, (ii) allocating a new CMEM buffer when the previous one fills-up, and (iii) writing to CMEM any contents remaining in the CBUF before a thread is pre-empted.

When a CMEM buffer reaches its capacity, Capo3 writes it to a file. Because there may be multiple full CMEM buffers in the system waiting to be written to the file, the RSM serializes this write operation using a work queue handled by a dedicated thread. This work queue provides an effective back-pressure mechanism when the buffer completion rate of the recorded threads exceeds the speed of the thread that empties the queue. Specifically, when the work queue becomes full, the RSM puts the recorded threads to sleep until the work queue can catch up. This mechanism preserves correctness, although it may negatively impact recording performance.

3.4 Handling Context Switches

On a context switch, the RSM first executes an XFC instruction to ensure that the current chunk terminates, and that all the residual data in the processor's CBUF are flushed to CMEM. This is needed to avoid mixing the log of the current thread with the next thread.

Once this has been performed, the RSM saves and restores the values of the registers in the MRR. Specifically, for the current thread, it saves the registers that the hardware may have modified during execution. They are the CMEM_IDX and MRR_FLAGS registers. Then, before the next thread can execute, the RSM restores the thread's prior CMEM_PTR, CMEM_IDX, CMEM_SZ, CMEM_TH, MRR_CTL, and MRR_FLAGS values, enabling it to correctly resume execution.

3.5 Recording in Kernel Mode

Certain parts of the kernel can interact with a process' address space, creating the potential for the kernel to have races with user-level instructions. The *copy_to_user* family of functions in the Linux kernel is an example of such code. Hence, in order to record all the memory access orderings that can affect the execution of an application during replay, the QuickRec hardware must also capture the execution of these kernel-level memory accesses.

QuickRec provides a flag that, if set, allows the MRR to record kernel instructions as well as user-mode instructions. Hence, to record sections of the kernel such as *copy_to_user()*, our initial approach was to set that flag prior to entering *copy_to_user()* and reset it after returning from *copy_to_user()*. The problem with this approach is that an asynchronous interrupt (e.g., from a hardware device) or a page fault can occur during the execution of *copy_to_user()*. In this case, since the flag is still set, QuickRec would incorrectly record the interrupt or page fault handler code.

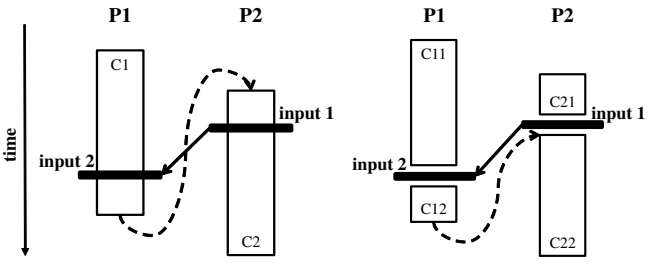
Our solution to this problem is to have an MRR_FLAGS register, where the least significant bit (LSB) acts as the previously-mentioned flag. On entry to *copy_to_user()*, we set the LSB, while on returning from it, we reset it. Moreover, the register operates as a shift register. When an exception is taken, the register automatically shifts left with a 0 being inserted into the LSB, which disables recording. Upon returning from the exception handler (as indicated by the *iret* instruction of x86), the register shifts right, restoring the previous value of the LSB. If the exception has happened in the middle of a *copy_to_user()*, this design disables recording as soon as the exception is taken, and resumes it as soon as the execution returns to *copy_to_user()*.

3.6 Handling Input/Chunking Interactions

The RSM component that records the input log and the one that manages the chunking log proceed almost independently from each other, each creating a total order of their events. However, in our initial implementation, we observed a subtle interaction between the two components that resulted in occasional deadlocks.

The problem occurs if a chunk includes instructions from both before and after and input event. In this case, the dependences between chunks and between inputs may intertwine in a way that causes deadlock.

As an example, consider Figure 6a, where chunks C1 and C2 execute on processors P1 and P2. Suppose that C2 first executes an input event that gets ordered in the input log before an input event in C1. Then, due to a data dependence from P1 to P2, C1 is ordered in the chunking log before C2. We have recorded a cyclic dependence, which makes the resulting logs impossible to replay and, therefore, causes deadlock.



(a) Deadlock in replay (b) Deadlock avoided

Figure 6: Examples of dependences between input events (solid lines) and between chunks (dashed lines).

To avoid this problem, Capo3 does not let a chunk include instructions from both before and after an input event. Instead, before an input event is recorded, the RSM executes the XTC instruction — therefore terminating the current chunk. With this approach, the situation in Figure 6a transforms into the one in Figure 6b. In this case, there are four chunks and the cyclic dependence has been eliminated. Both input and chunk dependences are satisfied if we replay the chunks in the C11, C21, C12 and C22 order.

Another issue related to the interaction between the two logs is how the replayer can match the input log entries and the chunk log entries generated by the same thread. Fortunately, this is easy, since the RSM assigns a unique R-Thread ID to each thread (Section 3.2). As the logs are generated, they are augmented with the R-Thread ID of the currently-running thread. In particular, as the RSM writes the CMEM buffers to the log, it attaches the current R-Thread ID to the buffer's data.

4. PROTOTYPE CHARACTERIZATION

4.1 Experimental Setup

We evaluate the QuickRec system by collecting and analyzing both log data and performance measurements for a set of SPLASH-2 benchmarks (Table 3). We execute each benchmark to completion, and show results for a default configuration of 4 threads running on 4 cores. In addition, we also assess the scalability of QuickRec by analyzing runs with 1, 2, 4, and 8 threads. For our experiments, we pin each application thread to a particular core. Thus, in the default case, we assign each thread to its own core and, in the 8-threaded case, we assign two threads to each core. We implement Capo3 as a kernel module in Linux 3.0.8.

Benchmark	Input Size	# of Instruc. (B)
Barnes	nbody 8000	3.4
FFT	-m 22	3.7
FMM	-m 30000	5.3
LU	-n 1024	3.0
LU-NC	-n 1200	4.7
Ocean	-n 1026	7.5
Ocean-NC	-e1e-16	2.2
Radix	-n 10000000	2.3
Raytrace	teapot.env	0.3
Water	1000 molecules	5.4

Table 3: Characteristics of the benchmarks. The last column shows the total number of instructions executed in the 4-threaded run in billions. *Water* refers to Water-nsquare.

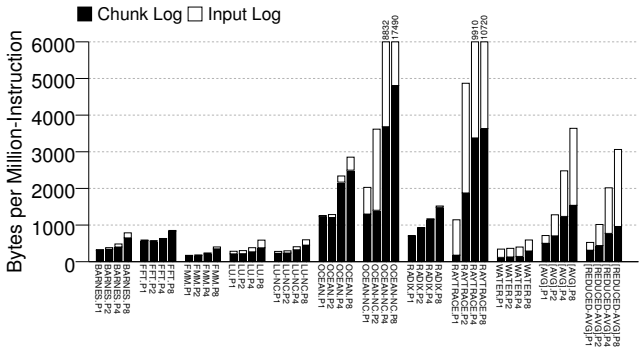
4.2 Log Analysis

In this section, we analyze the size and bandwidth requirements of the logs generated during the recorded execution. In addition, for the chunk log, we perform a detailed characterization. In all cases, we consider logs without data compression.

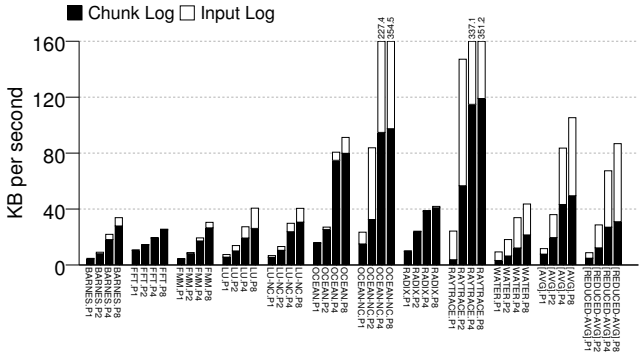
4.2.1 Log Sizes and Bandwidth

Figure 7a shows the *uncompressed* size of the input and chunk logs for each of the benchmarks and for the average case (*AVG*). For each benchmark, we show data for 1-, 2-, 4-, and 8-threaded runs. The size is given in bytes per million instructions. From the bars, we see that the average log size produced by QuickRec for 4 threads is 1,224 and 1,235 bytes per million instructions for input logs and for chunk logs, respectively. These are small numbers. However, the *Ocean-NC* and *Raytrace* benchmarks generate notably larger logs for 4-8 threads. This effect is mainly due to the increased use of synchronization in the benchmarks, which involves frequent calls to the *futex()* system call. As a result, the input log size increases substantially. Also, since Capo3 terminates the running chunk before recording an input event (Section 3.6), the chunk log also grows substantially.

The average log sizes that we measure are in line with sizes reported in previous work. For example, the log sizes reported for



(a) Uncompressed log sizes



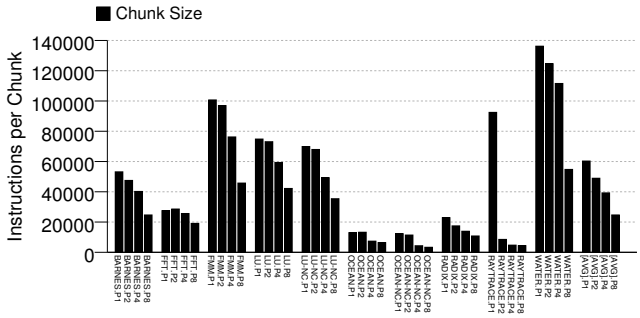
(b) Memory bandwidth requirements

Figure 7: Analyzing the log sizes without data compression and the resulting memory bandwidth requirements.

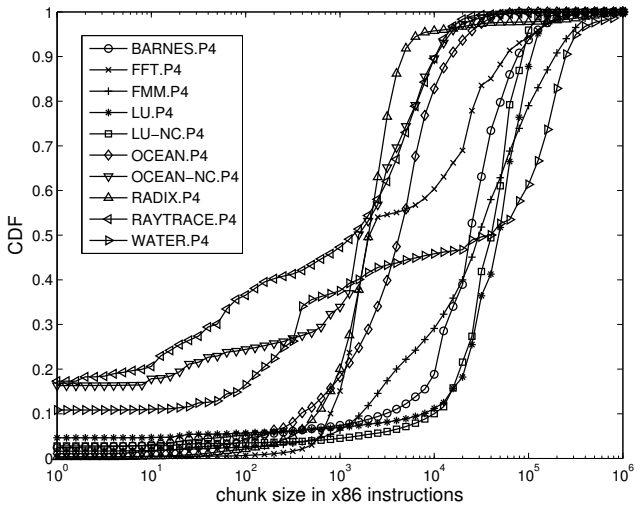
Cyrus [12], DeLorean [23], Rerun [13], and LReplay [7] are all within approximately 0.5x–2x of ours. We also note that our numbers correspond to a simple, unoptimized RnR implementation, and can easily be improved. As a simple example, consider the log entry for a chunk in QuickRec (Figure 4). Of the 128 bits, in most cases, only 80 bits are used for RnR. The remaining bits are mostly used for debugging and characterization of the hardware. If we eliminated them, we would get the average log sizes labeled REDUCED-AVG in Figure 7a. Further log size reductions can be attained with improved bit encoding.

Figure 7b shows the memory bandwidth requirements of logging. The figure is organized as the previous one and shows bandwidth in KB per second. From the average bars, we see that the bandwidth for 4 threads is 40 KB/s and 43 KB/s for input and chunk logs, respectively. These numbers, when combined, represent only 0.3% of the 24 MB/s bandwidth available in our prototype (Table 1). Hence, the effect of logging on bus and memory contention is very small. If we use the 80-bit chunk entries for the log (bars labeled REDUCED-AVG in Figure 7b), the bandwidth requirements are slightly lower.

To reason about the bandwidth requirements of QuickRec's logging on modern computers, consider the following. A modern multicore computer cycles at a higher frequency than our prototype, but it also has higher memory bandwidth. To understand the impact of these changes, we recompiled and ran our benchmarks on a dual socket Xeon server with 2.6 GHz E5-2670 processors. We measured the elapsed time (and speedup over our prototype) of the 4-threaded applications and scale the bandwidth numbers accordingly. Assuming the 80-bit log entry per chunk, we obtained an average bandwidth consumption across the benchmarks of 17.9 MB/s (and 61.1 MB/s for *Ocean-NC*, which is bandwidth-intensive). Given that the E5-2670 processor provides a memory



(a) Average chunk size



(b) Cumulative distribution of chunk size

Figure 8: Chunk size characterization.

bandwidth of up to 6.4 GB/s per core, the logging accounts for only 0.07% on average (and 0.23% in *Ocean-NC*) of the available bandwidth of 4 cores. Based on these estimates, we conclude that the bandwidth usage is negligible and will not have a negative impact on the performance of real systems.

If we compress the logs using gzip's default DEFLATE algorithm, we attain an average compression ratio of 55% for chunk logs and 88% for input logs. Hence, the average 4-threaded benchmark can be recorded for almost three days before filling up a terabyte disk.

Finally, Figure 7a and Figure 7b also suggest that both the log sizes and the bandwidth requirements scale reasonably as the number of threads increases from 1 to 8.

4.2.2 Chunk Characterization

Figure 8a shows the average size of the chunks in terms of retired x86 instructions. Figure 8b shows the distribution of chunk sizes for 4-threaded runs. On average, the size of a chunk for 4-threaded runs is $39K$. However, Figure 8b shows that, while many chunks are large (e.g., more than 80% of the chunks in *Barnes*, *LU*, and *LU-NC* are larger than 10,000), there are many chunks with fewer than 1,000 instructions. For three benchmarks, there is a significant fraction of zero-sized chunks, which mostly result from explicitly terminating a chunk unconditionally at input events. This effect can be avoided by changing Capo3 or the hardware.

Figure 9 details the chunk termination reasons, using the categories shown in Table 2, except that exceptions, chunk-size overflows, and TLB invalidations are grouped together in *Other*. From the figure, we see that the largest contributor to chunk termination is cache line evictions. In the QuickRec hardware, a chunk must be terminated if a line that is evicted from the L2 hits the read set or the write set in the same core. This is because subsequent snoop requests to that line are not delivered to the MRR; they are filtered out by the L2. Techniques to mitigate this behavior will contribute to reducing the number of chunks.

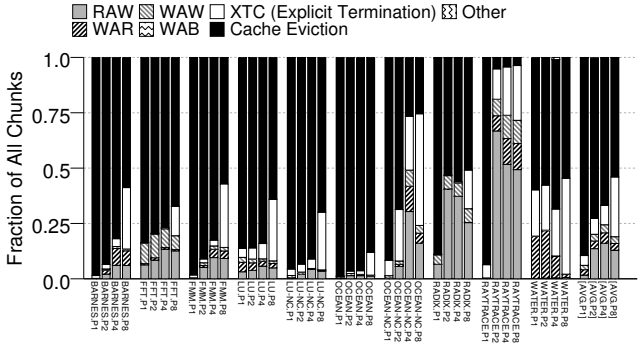


Figure 9: Chunk termination reasons.

Conflicts due to WAR, RAW, WAW and WAB are the second most prevalent reason of chunk terminations. Another frequent reason is explicit chunk termination with XTC. This termination reason is common when we have more threads than processors (i.e., in the 8-threaded runs). In this case, there are many context switches which use XTC. This reason is also common if the benchmark has numerous input events, such as signals or system calls, which require explicit use of XTC to obtain a total order of events. For example, this is the case for *Raytrace* and *Ocean-NC*, which, as shown in Figure 8b, have a large number of zero-sized chunks.

To deal with instruction reordering and instruction atomicity violations, QuickRec appends RSW and IAV information to chunk entries. Figure 10 displays the fraction of chunks that are associated to non-zero RSW and/or IAV values. The figure reveals that such chunks are common. For 4-threaded runs, an average of 16% of the chunks are RSW or IAV chunks. In fact, both RSW-only and IAV-only chunks are common. One interesting case is that of *Radix*, where the fraction of IAV chunks is over 40%. The reason is that *Radix* has a long-running tight loop with several multi-memory-operation instructions. *FFT* has many RSW-only chunks, which result from executions where loads and stores are interleaved. Overall, RnR systems must be designed to handle these cases.

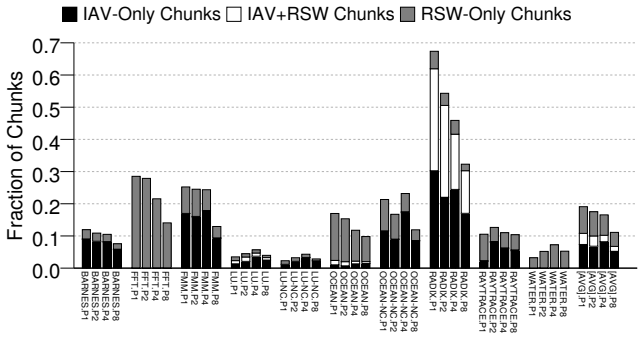


Figure 10: RSW and IAV characterization.

4.3 Performance Measurements

To measure the overhead of QuickRec's different components, we ran each benchmark in five different configurations. First, *na-*

tive is the normal execution with no recording. Second, in *hw-only*, the MRR hardware is enabled and writes chunk data to main memory, but otherwise no other component of the system is enabled. This configuration measures the overhead of the extra memory traffic generated by the MRR. Third, in *input*, the RSM only logs the sources of input non-determinism described in Section 3.2 and the MRR is disabled. Fourth, *chunk* augments the *hw-only* configuration by having the RSM dump the CMEM buffers to a file; no input is recorded. Finally, *combined* is a full recording run where both input and chunk data are processed by the RSM. To reduce the OS-induced noise, each configuration is run five times and the results are averaged. Each run executes with four threads.

Figure 11 shows the execution time of each configuration normalized to the execution time of *native*. The figure shows that, in most benchmarks, recording both input and chunk logs only incurs a 2–4% overhead. The main exceptions are *Ocean-NC* and *Raytrace*, which suffer an overhead close to 50%. As indicated in Figure 7a, these two benchmarks perform substantial synchronization, which involves frequent calls to the *futex()* system call and, often, results in putting threads to sleep. On average across all of the benchmarks, the recording overhead is 13%.

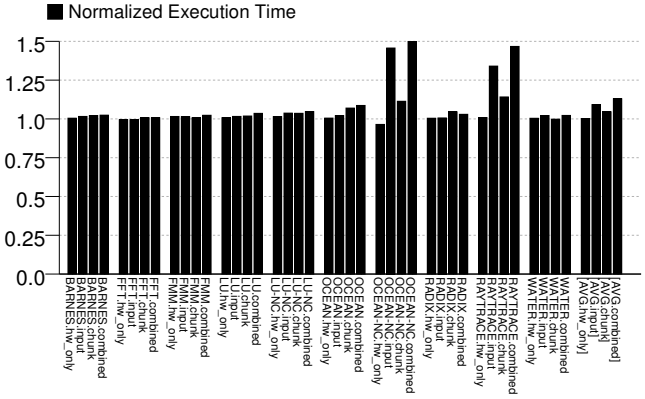


Figure 11: Execution time with each recording configuration for four-threaded executions. The bars are normalized to the execution time of *native*.

Interestingly, the recording overhead is entirely due to the software stack. Indeed, the hardware overhead, as shown in *hw-only*, is negligible. We also see that the software overhead is primarily due to input logging, rather than chunk logging. Overall, future work should focus on optimizing the software stack and, in particular, input logging — specifically, removing the serialization in the recording of input events.

Figure 12 shows the processor time (the time processors spend doing useful work for the applications) separated into user and system time. For each benchmark, we show three bars: one for the recorded application itself (*App*), one for the driver that reads the input log from memory and writes it to disk (*Input*), and one for the driver that reads the chunking log from the memory and writes it to disk (*Chunking*). For each benchmark, the bars are normalized to the processor time of the application.

The figure shows that most of the processor time is spent running the application. On average, the drivers add little overhead. Only the two benchmarks with large logs in Figure 7a spend noticeable time in the drivers. Finally, most of processor time in these applications is user time.

To understand the sources of overhead in QuickRec, Figure 13 breaks down the total processor cycles into four categories. First, *App time* are the cycles spent executing instructions not resulting from Capo3 overhead. Second, *Input overhead (working)* are the

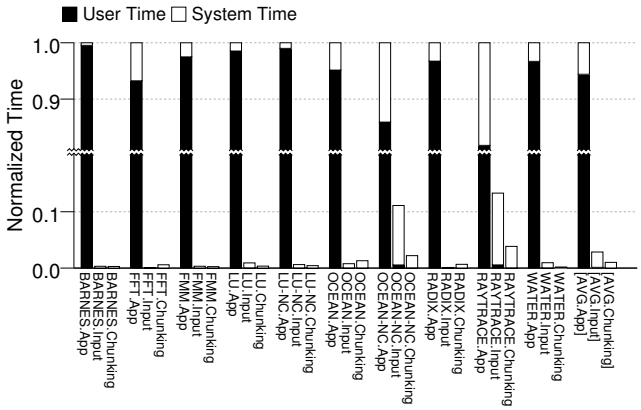


Figure 12: Total time that the processors spend working on the applications divided into user and system time.

cycles spent in Capo3 code managing the input events. Third, *Input overhead (sleeping)* are the cycles spent in Capo3 waiting on synchronization in order to enforce a total order of input events. Finally, *Chunking overhead* are the cycles spent in Capo3 code managing the chunking log. The figure shows the breakdown for different thread counts. As the figure indicates, for 4- and 8-threaded runs, the main overhead of Capo3 is due to enforcing a total order of input events. We are looking into optimizations and/or alternative designs for this component.

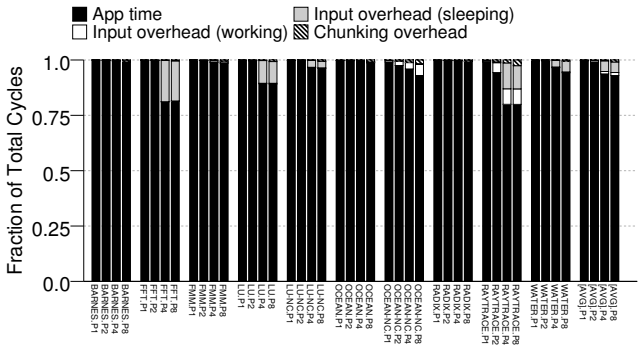


Figure 13: Breakdown of the total processor cycles for different thread counts.

Figures 14 and 15 present detailed breakdowns of the input and chunking overheads, respectively, for different thread counts. In each figure, the overheads are normalized to the overhead of the 1-threaded execution for the given benchmark.

Figure 14 divides the overhead of input recording and management into the contributions of system calls, copy to user (CTU), and other events. In each case, the figure separates working and sleeping overheads. The figure shows that the sleeping overhead resulting from serializing the system calls is by far the largest component for 4- and 8-threaded runs. In particular, *FFT*'s normalized overhead for 4- and 8-threaded runs is high. The reason is that *FFT* has minimal overhead with 1 thread and has many synchronization-induced *futex()* calls with 4 or more threads.

Figure 15 depicts a similar breakdown for the chunk-management overhead. The overhead is divided into execution of XTC instructions (*Chunk term*), execution of XFC instructions (*CBUF flush*), allocation of a new CMEM buffer (*Buffer allocation*), putting a CMEM buffer in the work queue (*To workqueue*) and *Other*. The latter is dominated by the overhead of saving and restoring MRR registers in a context switch. We see that *Buffer allocation* and *Other* dominate.

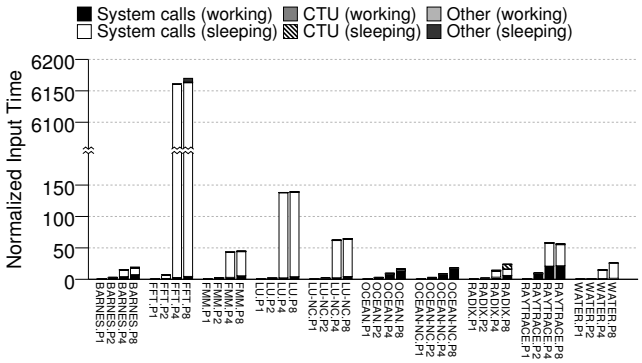


Figure 14: Breakdown of the normalized overhead of input recording and management. CTU stands for Copy To User.

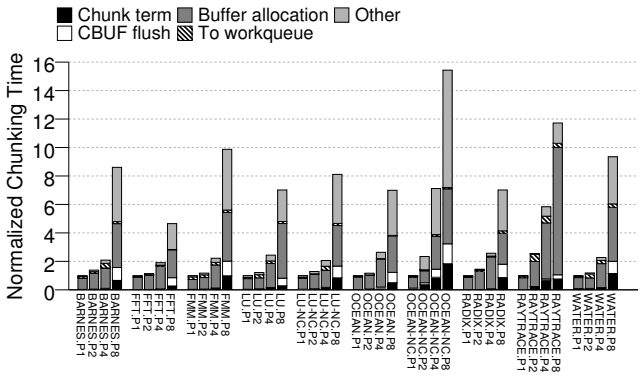


Figure 15: Breakdown of the normalized chunk-management overhead.

5. VALIDATION USING REPLAY

A critical aspect of the design and implementation of a recording system is to validate it with replay. Replaying recorded logs enables full assurance that the recording system captures the correct and complete information. Therefore, in this section we discuss the replayer from the perspective of its validation of QuickRec.

We implemented the replayer using the Pin [20] binary instrumentation framework. We chose this approach for three reasons. First, user-level binary instrumentation is operating-system independent (similar to PinPlay [28]), which enables replay to occur on a machine that is independent from the QuickRec system. Second, Pin operates at speeds faster than existing instruction-set simulators, while maintaining an acceptable level of observability. Third, using Pin, we can extend the replayer by integrating other analysis tools, such as race detectors [2, 33] and debuggers [19].

5.1 High-Level Implementation Description

To correctly replay a recorded execution, the replayer requires the executed code (binary and libraries, including self-modified code), and the program inputs and shared-memory access interleaving experienced during the recorded execution. Prior to replay, the static code is extracted from the log files. Self-modified code, which is not present in the log files, is re-generated by the replayed execution. Non-deterministic inputs are made deterministic by injecting the appropriate recorded data into the replayed execution at appropriate execution points. For most system calls (e.g., *read()*), this operation involves emulating the system call, by: (i) injecting the logged data into the program if there is a logged *copy_to_user()* entry, and (ii) setting the return values as defined in the input log. However, there are a few system calls, such as thread creation and termination, that are re-executed to recreate the proper kernel state.

Chunk ordering is accomplished by counting instructions as they are replayed, and stopping when the counter reaches the logged chunk size. In addition, the replayer enforces the logged chunk order, based on the recorded timestamps.

5.1.1 Chunks with Non-Zero RSW or IAV Counts

To handle the IA memory model correctly, the replayer needs to take into account the values of the RSW and IAV counts. Specifically, to support TSO, the replayer simulates a thread-local store buffer. On a store operation, the replayer writes the address and value of the store to the local store buffer — instead of committing the store to the global memory. On a load operation, the replayer first checks the local store buffer. If the address is not found, it loads the value from the global memory. Then, at the end of the chunk, the replayer drains the stores from the local store buffer, except for a number equal to the RSW count of the chunk, and commits their values to the global memory. The stores remaining in the local store buffer are committed as part of the next chunk.

To handle non-zero IAV counts, the replayer needs to know the number of memory transactions involved in the execution of each instruction. When the replayer finds a chunk whose IAV is non-zero, after executing the chunk, it emulates the execution of the memory transactions of the first instruction after the chunk, one at a time. The replayer stops when the number of memory transactions is equal to the IAV count. The remaining memory transactions of the instruction are emulated at the beginning of the next chunk.

5.2 Validating the Complete System

Prior to full-system tests, we developed multiple levels of system validation. We began with RTL simulations to validate the MRR hardware without software, while we used Simics [21] simulations to validate Capo3. Next, we integrated Capo3 with QuickRec and developed tests to independently exercise the recording functionalities of input non-determinism and shared-memory interleaving. Last, we tested the complete system with our benchmarks.

When bugs were found during full-system tests, the major challenge was pinpointing their origin. In QuickRec, bugs can originate from either the replayer, the recording hardware, or the recording software; distinguishing between the three is usually non-trivial. In our experiments, the most common type of bug manifestation was a divergence between the memory state or the control flow of the recorded and replayed executions. There are many reasons why a divergence can occur, and being able to pinpoint the root cause of such a divergence is critical.

The most obvious location to check for divergent executions is where non-deterministic input events are logged. This is because, during recording, Capo3 saves the contents of the processor registers at the entry of system calls. Hence, the replayer can compare the state of the processor registers before a system call to the recorded state. This provides a clear detection point of divergence. Moreover, a system call should result in a chunk termination and, therefore, should be the last instruction of the chunk it belongs to. This provides another divergence check.

Unfortunately, non-deterministic input events are infrequent and, therefore, insufficient to detect the root cause of most divergences — the source of divergence can be thousands of instructions before the system call. Therefore, a more fine-grained mechanism to detect divergences was needed.

For this purpose, we added a branch-tracing module in the FPGA hardware. It collects the history of branches executed — like the Branch Trace Store of today's IA processors. With this informa-

tion, the replayer can compare the control flow of the recorded execution with that of the replayed execution. This is a powerful method to detect divergences, since if either the record or replay system has a bug, then the replayed execution typically leads to a different control flow. Also, with branch traces, the detection point of a divergence tends to be close to its source.

5.2.1 Hardware Instruction Counting Bug

With branch tracing, we found one particularly noteworthy hardware bug. In the *water* benchmark, we found that a system call was not aligned with the end of the chunk during replay, indicating a bug in the system. The replayer was encountering a system call two instructions prior to the expected end of the chunk. At first, the problem appeared to be a control-flow divergence manifesting as different instruction counts between the log and replayed execution. However, the branch traces revealed no control-flow divergence. Further investigation showed that the hardware was miscounting instructions when handling floating-point exceptions. Without a confirmation from the branch traces regarding no control-flow divergence, it would have been very difficult to pinpoint this bug.

6. RELATED WORK

RnR systems can be classified into software-only and hardware-assisted. Software-only RnR systems (e.g., [5, 8, 9, 10, 11, 18, 27, 28, 32, 34]) run on commodity hardware and use modified runtime libraries, compilers, operating systems or virtual-machine monitors to capture sources of non-determinism. These software-based approaches are either inherently designed for uniprocessor executions or suffer significant slowdown when applied to multiprocessor executions. DoublePlay [35] attempts to make replay on commodity multiprocessors more efficient. To capture memory non-determinism, it timeslices a multithreaded execution into separate epochs and re-executes each epoch sequentially on a single processor. Hence, for each epoch, it only needs to record the order in which threads are scheduled in the second execution. However, DoublePlay cannot capture all data races and, therefore, cannot be used as a general solution for concurrency debugging. In addition, it requires an extra execution to record thread ordering. Finally, it needs to use modified binaries (in particular, a modified *libc*).

Hardware-assisted solutions use hardware to record memory access order. Some approaches modify coherence transactions in conventional directory-based protocols (e.g., [3, 13, 23, 24, 26, 39, 40]) and some are based on snoopy protocols (e.g., [12, 25, 30, 31]). Some approaches (e.g., [39, 40]) record dependences between pairs of instructions. This strategy can produce large logs and increase associated overhead. To reduce this overhead, chunk-based techniques have been proposed (e.g., [7, 12, 13, 23, 24, 30, 31, 36]). DeLorean [23] and Capo [24] are chunk-based schemes that use speculative multithreading hardware to achieve replay parallelism.

In terms of the hardware, QuickRec resembles CoreRacer [31] the most. While the chunking and the instruction reordering are handled similarly, the main differences are on the implementation of instruction atomicity violation, and on the integration of input recording and chunking. LReplay [7] extends a multiprocessor system with a pending period-based mechanism for recording thread interleaving, and uses large CAM structures to deal with instruction reordering. LReplay is evaluated using RTL simulation and does not discuss issues related to system software.

All of these hardware-assisted approaches have only been modeled using simulation, and often without considering the necessary software support. As such, they have generally ignored practical aspects of RnR systems. The QuickRec system is the first work to evaluate RnR across the entire stack using real hardware.

7. LESSONS LEARNED

The main lessons we learned from this effort are:

• Clearly, to maximize the chance that RnR is considered for adoption, it is critical to minimize the number of touch points that it requires on current processor hardware. QuickRec demonstrates that chunk-based recording can be implemented with low-enough implementation complexity and few-enough touch points to make it attractive to processor vendors.

• By far the biggest challenge of implementing RnR is dealing with the idiosyncrasies of the specific architecture used, as they fundamentally permeate many aspects of the hardware and software. Examples of idiosyncrasies are the memory consistency model and the CISC nature of the architecture.

• The design of the deterministic replayer must account for the micro-architectural details of the system, if it is to reproduce the execution exactly. This was altogether neglected by prior replay work. In fact, such micro-architectural details substantially increase the replayer's complexity, in turn impacting the usage models and potentially the ability to create non-proprietary replay tools.

• A new research direction is to investigate replay techniques that reduce or abstract away the complexity mentioned. Such techniques may hinge on commodity hardware, or may require hardware extensions to enable replay software.

• The design of the recording software stack can considerably impact the hardware design, as well as the overall performance. For instance, to properly record kernel-mode instructions (e.g., *copy_to_user()* calls), we had to make non-trivial changes to the hardware-software interface (Section 3.5). Also, the software stack is responsible for practically all of the QuickRec recording overhead.

• The main performance overhead in QuickRec is in the software layer collecting and managing the input logs. A seemingly unimportant issue such as the serialization of input-event processing has become our most obvious bottleneck. Recording input events very efficiently is an area were further work is needed.

• The performance analysis clearly suggests that, with a slightly-improved software stack, RnR can be used in *always-on* manner, enabling a potentially-large number of new RnR uses. Additional features may need to be added, such as checkpointing and log compression to reduce log file sizes in long-running programs.

• Finally, full-system prototyping is required to understand RnR issues related to architecture idiosyncrasies, hardware-software interaction, and true performance bottlenecks.

8. CONCLUSIONS AND FUTURE WORK

RnR of multithreaded programs on multicores has high potential for several important uses: debugging applications, withstanding machine failures, and improving system security. To make RnR systems practical, this paper has contributed in three ways.

First, we presented the implementation of QuickRec, the first multicore IA-based prototype for RnR of multithreaded programs. The prototype includes an FPGA instantiation of a Pentium multicore and a Linux-based full software stack.

Second, we described several key implementation aspects in QuickRec. We showed how to efficiently handle x86 instructions that produce multiple memory transactions, and detailed the elaborate hardware-software interface required for a working system.

Third, we evaluated QuickRec and demonstrated that RnR can be provided efficiently in real IA multicore machines. We showed that the rate of memory log generation is insignificant, given today's bus and memory bandwidths. Furthermore, the recording hardware had negligible performance overhead. However, the software stack

induced an average recording overhead of nearly 13%. Such overhead must come down to ensure always-on use of QuickRec.

Based on this work, we suggest focusing future research on several directions. First, to reduce the software stack overhead, it is important to record input events very efficiently — specifically, in a partially-ordered manner. This will reduce recording overhead, and truly enable always-on RnR.

Second, much emphasis should be placed on the replay aspect of RnR. We need approaches that are tolerant of, and abstract away, the micro-architectural details of the recording platform. Otherwise, proprietary details will stifle the development of replay support. We need creative ways of combining hardware and software support for replay.

Finally, we need to develop and demonstrate many uses of the RnR technology that solve real problems of multicore users. The areas of parallel program development tools and security-checking aids seem particularly ripe for development.

9. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, May 1991.

[2] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling Data Race Detection in the Intel Thread Checker. In *STMCS*, March 2006.

[3] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In *ICS*, June 2011.

[4] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *PLDI*, June 2000.

[5] T. Bressoud and F. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Transactions on Computer Systems*, 14(1), February 1996.

[6] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, August 2001.

[7] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: A Pending Period Based Deterministic Replay Scheme. In *ISCA*, June 2010.

[8] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SPDT*, August 1998.

[9] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, December 2002.

[10] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE*, March 2008.

[11] A. Forin. Debugging of Heterogeneous Parallel Systems. In *PDD*, May 1988.

[12] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism. In *ASPLOS*, March 2013.

[13] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, June 2008.

[14] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2002. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *SOSP*, October 2005.

[16] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP*, October 2003.

[17] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX Annual Technical Conference*, April 2005.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comp.*, April 1987.

[19] G. Lueck, H. Patil, and C. Pereira. PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation. In *CGO*, 2012.

[20] 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 *PLDI*, 2005.

[21] P. 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*, February 2002.

[22] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter*, pages 19–25, December 1995.

[23] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, June 2008.

[24] P. Montesinos, M. Hicks, S. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, March 2009.

[25] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, October 2006.

[26] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, June 2005.

[27] D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *PDD*, May 1988.

[28] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO*, April 2010.

[29] C. Pereira, G. Pokam, K. Danne, R. Devarajan, and A.-R. Adl-Tabatabai. Virtues and Obstacles of Hardware-Assisted Multi-Processor Execution Replay. In *HotPAR*, June 2010.

[30] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-Based Memory Race Recorder in Modern CMPs. In *MICRO*, December 2009.

[31] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, H. Jungwoo, and Y. Wu. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *MICRO*, 2011.

[32] M. Russinovich and B. Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *PLDI*, May 1996.

[33] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *WBIA*, December 2009.

[34] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Ann. Tech. Conf.*, June 2004.

[35] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, March 2011.

[36] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *ISCA*, June 2010.

[37] Q. Wang, R. Kassa, W. Shen, N. Ijih, B. Chitlur, M. Konow, D. Liu, A. Sheiman, and P. Gupta. An FPGA Based Hybrid Processor Emulation Platform. In *FPL*, August 2010.

[38] XtreamData. http://www.xtreamdata.com.

[39] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, June 2003.

[40] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *ASPLOS*, 2006.

[41] M. V. Zelkowitz. Reversible Execution. *Communications of the ACM*, 16(9):566, September 1973.