Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism

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

Gilles Pokam, Cristiano Pereira
Intel
Santa Clara, CA
{gilles.a.pokam, cristiano.l.pereira}@intel.com

Abstract

Architectures for deterministic record-replay (R&R) of multi-threaded code are attractive for program debugging, intrusion analysis, and fault-tolerance uses. However, very few of the proposed designs have focused on maximizing replay speed — a key enabling property of these systems. The few efforts that focus on replay speed require intrusive hardware or software modifications, or target whole-system R&R rather than the more useful application-level R&R.

This paper presents the first hardware-based scheme for unintrusive, application-level R&R that explicitly targets high replay speed. Our scheme, called *Cyrus*, requires no modification to commodity snoopy cache coherence. It introduces the concept of an on-the-fly software Backend Pass during recording which, as the log is being generated, transforms it for high replay parallelism. This pass also fixes-up the log, and can flexibly trade-off replay parallelism for log size. We analyze the performance of Cyrus using full system (OS plus hardware) simulation. Our results show that Cyrus has negligible recording overhead. In addition, for 8-processor runs of SPLASH-2, Cyrus attains an average replay parallelism of 5, and a replay speed that is, on average, only about 50% lower than the recording speed.

Categories and Subject Descriptors C.0 [**General**]: Hardware/ Software Interfaces; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - MIMD Processors; D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming; D.4.0 [**Operating Systems**]: General; D.4.1 [**Operating Systems**]: Process Management - Threads

Keywords Deterministic Replay, Application-level Parallel Replay, Unintrusive Hardware-Assisted Recording, Source-only Recording, Backend Log Processing

1. Introduction

Deterministic Record-Replay (R&R) seeks to monitor the execution of a program (or a set of programs) and exactly reproduce it on a subsequent execution. R&R has broad uses in at least program debugging [1, 15, 33], where, for example, a concurrency bug can be reproduced, intrusion analysis [10, 14], where an intrusion can be traced back to an attacker's actions, and fault-tolerant, highly-available systems [9, 32], where a backup machine can resume where the primary failed. This paper focuses on R&R for multi-threaded applications on multiprocessor machines. In such a scenario, R&R typically involves recording all the non-deterministic events that occurred during the initial execution — i.e., application inputs and memory access interleavings. Then, during replay, logged inputs are provided to the application at the correct times, and the memory accesses are forced to interleave in the same manner as in the log.

There are several proposals of schemes for R&R of multi-threaded programs. On the one hand, there are those that do not require any special hardware [2, 5, 10, 11, 18, 27, 28, 32–34], typically relying on the OS, compiler and/or run-time libraries for recording and replaying. Being software-only solutions, these systems are relatively inexpensive to implement but tend to run slowly during recording. Other schemes record with the aid of some special hardware module [3, 7, 13, 23–26, 30, 31, 35–37]. These systems add negligible overhead during recording, but can be expensive to implement. In addition, some schemes R&R the whole machine's execution (e.g., [3, 13, 23]), while there are others that only R&R a single or a group of applications running on the machine (e.g., [24, 25, 33]). Typically, it is the latter (application-level R&R) that the users actually need rather than the former (whole-system R&R). In addition, recreating the whole machine state during replay is often very hard and, to work correctly, needs to deal with many non-portable operating system and hardware issues of the platform. Application-level R&R, in contrast, tends to be more portable and adds less overhead.

Different R&R schemes attempt to optimize different metrics. Traditionally, hardware-based R&R schemes have attempted to minimize log size requirements. Software-based schemes, instead, have focused on minimizing the overhead of recording — in some cases, even at the cost of potentially having to replay multiple times [2, 28]. Very few schemes have focused on maximizing replay speed — most notably DeLorean/Capo [23, 24], DoublePlay [34], and Karma [3]. All three use *parallel replay* mechanisms for this purpose.

Each of the three previous systems has shortcomings that could limit its practicality. Specifically, DeLorean/Capo uses transactional record and replay hardware, which requires a redesign of current commodity processor hardware. Karma provides whole-system R&R rather than application-level R&R. As indicated above, this is not what users typically need and, in addition, it is hardly portable. In addition, Karma requires augmenting the cache coherence protocol messages — which we want to avoid. Finally, DoublePlay is a software-based scheme, which requires modifying and recompiling the application, marking its synchronizations.

This is unfortunate, given that fast replay is a key enabling property for R&R systems. For example, debugging can be more productive if buggy executions can be quickly replayed to the point of the bug. Similarly, intrusion analysis can benefit from extensive on-the-fly analysis of how the attack is taking place. Finally, in fault tolerance, a backup machine has to quickly catch up with a failed one to provide hot replacement.

To attain effective low-overhead R&R, we believe that, in addition to providing fast parallel replay, the system needs to: (i) support *application-level* R&R, and (ii) rely on *unintrusive* hardware design. In particular, it should avoid system-level hardware changes such as any changes to the cache coherence protocol. We believe this is fundamental for acceptance of R&R hardware. Since most multiprocessors today use snoopy cache coherence, we require our design to be compatible with (and not modify) snoopy protocols.

In this paper, we make the following contributions:

• We present the first hardware-based approach for unintrusive, application-level R&R that explicitly targets high-speed replay. The approach, called *Cyrus*, requires no modification to commodity snoopy cache coherence.

• Cyrus introduces the concept of an on-the-fly software *Backend Pass* during recording which, as the log is being generated, consumes it and transforms it. This pass fixes-up the log, which has incomplete information due to our recording requirements of only application-level interactions and no cache coherence protocol changes. In addition, the backend pass exposes a high degree of parallelism for replay. Finally, as the backend pass produces the final log, it can also flexibly trade-off replay parallelism for log size.

• We modified the Linux kernel to control and virtualize a simulated version of the Cyrus hardware. Our results show that Cyrus adds negligible recording overhead, even with the backend pass. In addition, for 8-processor runs of SPLASH-2, Cyrus attains an average replay parallelism of 5 (in terms of the length of the critical instruction path), and a replay speed that is, on average, only about 50% lower than the recording speed.

The rest of the paper is organized as follows: Section 2 discusses background issues and challenges in R&R; Section 3 presents Cyrus' architecture; Section 4 describes implementation issues; Sections 5 and 6 evaluate Cyrus; Section 7 discusses related work; and Section 8 concludes the paper.

2. Background and Key Challenges

2.1 Background on Deterministic R&R

Deterministic Record-Replay (R&R) consists of monitoring the execution of a multithreaded application on a parallel machine, and then exactly reproducing the execution later on. R&R requires recording all the non-deterministic events that occur during the initial execution. They include the inputs to the execution (e.g., return values from system calls) and the order of the inter-thread communications (i.e., the interleaving of the inter-thread data dependences). Then, during replay, the logged inputs are fed back to the execution at the correct times, and the memory accesses are forced to interleave according to the log.

To accomplish application-level R&R, we leverage previous work, Capo [24], which describes how the OS virtualizes the R&R structures. A Replay Sphere is the single application (or group of applications) that we want to R&R in isolation from the rest of the system. Each sphere has an Input Log and a Memory Log.

The Memory Log collects the order of the data dependences between threads. To collect such orders, we can use a software-only solution that relies on the runtime or operating system. Alternatively, we can use a hardware-assisted scheme that relies on a special hardware module. This approach has the advantage of recording with negligible performance overhead, even for applications with frequent inter-thread data dependences. Hardware-assisted schemes typically use cache coherence transactions to detect inter-thread dependences.

To reduce the amount of state that needs to be collected in the Memory Log, most of the recent proposals of hardware-assisted schemes [3, 13, 23, 24, 30, 35] log the amount of work that a thread does between communications, rather than the communications themselves. Specifically, each entry in the log records the number of consecutive operations executed by a processor between inter-thread dependences. These groups of instructions or memory accesses are known as *chunks*, *episodes* or *blocks*. The entry also has information on what other block(s) in the log depend on this one. During replay, inter-thread dependences are enforced by executing these blocks in the proper order.

Most of the proposed block-based recording schemes encode the execution in a fairly serial form that can take away much of the parallelism that existed in the original execution. Specifically, some schemes log a total order of blocks [23, 24, 30], while others use Scalar Lamport Clocks (SLC) [17] to order blocks [13, 35]. In a basic design with SLCs, when a processor detects an inter-thread data dependence, it terminates its current block and assigns to it a scalar clock value (usually called *timestamp*). The recording mechanism guarantees that if an instruction in block *B2* depends on an instruction in block *B1*, then *B2* receives a timestamp strictly larger than that of *B1*. This way of using timestamps creates many unnecessary block ordering constraints which hide the original parallelism.

To recover lost parallelism, DeLorean uses speculative execution of blocks in parallel [23]. Karma [3], instead, records the block orders as a directed acyclic graph (DAG) of blocks. In the general case, each block has multiple predecessor and successor blocks, and parallel replay is possible.

Karma, however, is a whole-system R&R scheme that has been designed around a directory-based cache coherent system. It augments the coherence messages with timestamps and some new fields. It relies on the availability of explicit invalidation acknowledgements in directory protocols.

2.2 Key R&R Challenges

Our goal is to develop a useful and easy-to-implement R&R hardware scheme. Such a scheme needs to: (i) support application-level R&R rather than whole-system R&R, (ii) avoid changes to system-level multiprocessor hardware, especially changes to the snoopy cache coherence protocol, and (iii) enable highly-parallel replay.

In this section, we elaborate on the challenges that these three requirements pose. Our general approach matches current proposals [3, 13, 23, 24, 30, 35]: we use the cache coherence transactions to detect dependences between threads, and each entry in the log records the number of consecutive instructions executed by a processor between dependences.

2.2.1 Challenge 1: Application-Level R&R

The main difficulty in performing application-level R&R is that many of the cache coherence transactions observed in the machine are potentially unrelated to the application being recorded; they are due to the OS or to other applications. Consequently, an application-level R&R scheme has to identify these unrelated transactions and prevent them from inhibiting correct replay. This is unlike a whole-system R&R scheme.

Figure 1 shows this problem for a three-processor machine. As we run the application, we may observe coherence transactions between processors executing the application being recorded (e.g., transaction (1)). However, we may also observe transactions between the application being recorded and the OS (transaction (2)), between two OS threads (transaction (3)), and even between the application (or OS) and an application that is *not* being recorded (transactions (4) and (5)).

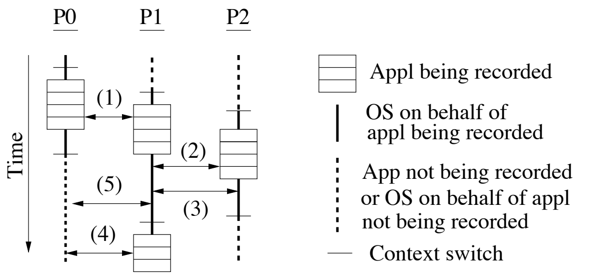

P0
P1
P2
Time
(1)
(2)
(5)
(3)
(4)
Appl being recorded
OS on behalf of appl being recorded
App not being recorded or OS on behalf of appl not being recorded
Context switch

Figure 1. Difficulties in capturing inter-thread dependences in application-level R&R.

Some of these communication events are unrelated to the application and can make the log inconsistent and cause replay divergence. They result from a variety of causes. One is interaction between OS and application threads, possibly through common buffers, and between OS threads. Another is the presence of hardware prefetchers, which may move unpredictable data and change its coherence state. Another effect is the processor issuing speculative loads, which access unpredictable data. In addition, the presence of context switches adds further uncertainty: a transaction may move data from a cache where the owner thread has been preempted. Should we record it? Finally, the Bloom filters [4] used in many R&R schemes to help detect dependences between threads [13, 23, 30] compound the problem: the events may be false positive dependences due to address aliasing in the signature.

2.2.2 Challenge 2: Unintrusive Hardware

The second challenge of an R&R scheme is the need to avoid changes to system-level hardware and, in particular, any changes to the cache coherence protocol. This paper focuses on snoopy coherent systems because they are the most commonly used approach today. In this environment, we must not augment the coherence protocol with new messages or even new fields in existing messages. This is because the timing of the messages is an integral part of the protocol design and any timing changes will require protocol re-validation.

A key consequence of this requirement is that the R&R scheme must record inter-thread dependences *from the dependence source only*. To see why, consider Figures 2(a)-(c). In these charts, processor *P1* initiates a request, which causes *P0* to supply a cached line and/or to invalidate it. These are dependences that need to be recorded in the R&R log. However, snoopy protocols provide incomplete information. Specifically, while the requesting processor (*P1*) includes its ID in the coherence transaction, the processor at the dependence source (*P0*) does not supply its ID in Figure 2(c) because there is no response message; in addition, *P0* may or may not provide its ID in Figures 2(a)-(b). This is unlike directory-based protocols, where there are explicit request and response messages that include the sender ID.

Hence, in any dependence, only the R&R module at the source processor (*P0*) knows about it and logs it; we have to assume that

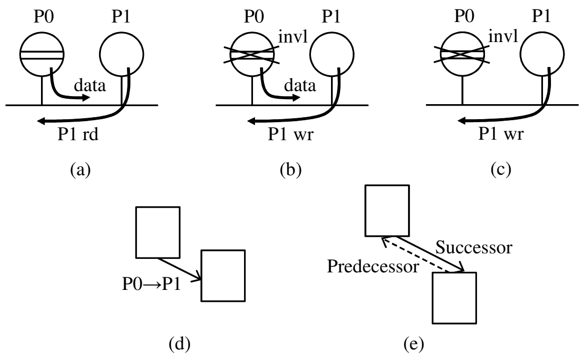


Figure 2. Keeping a snoopy cache coherence protocol unmodified requires recording dependences from the dependence source only.

the R&R module at the destination processor (*P1*) is completely unaware of the dependence. Any replay system must be able to reconstruct the execution from a log with only dependences of the form shown in Figure 2(d) instead of bidirectional successor and predecessor information as in Figure 2(e).

2.2.3 Challenge 3: Replay Parallelism

Parallelism is fundamental to high-speed replay, which in turn will enable new uses of this technology. To expose maximum parallelism, the log must encode the dependences between blocks across threads. Also, for each dependence, the source and destination block boundaries should be *as close as possible* to the dependence's source and destination references, respectively.

3. Unintrusive App-Level R&R for Replay Parallelism

3.1 Main Idea

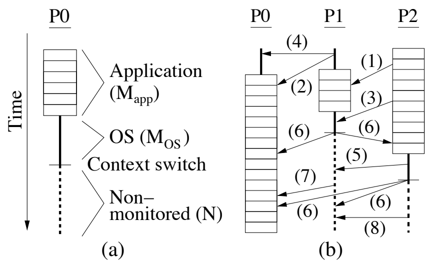
We propose a new general approach to address the previous challenges and deliver hardware-unintrusive, application-level R&R for replay parallelism. Our approach is called *Cyrus*. We use the mechanisms of Capo [24] to record input logs (Section 4.2). As for the memory log, to support application-level R&R, the hardware judiciously avoids logging certain types of interprocessor interactions. Moreover, to keep the cache coherence protocol unmodified, the hardware logs the dependence only on the source processor. The result of these two constraints is a log with some dependences that still need to be fixed-up or discarded, and with unidirectional dependence information only.

Consequently, as the log is being dumped into memory, an on-the-fly software *Backend Pass* consumes it and transforms it into the final log. This backend pass performs three actions: (i) fixes-up and discards some of the dependences to correctly implement application-level replay; (ii) transforms the unidirectional dependences into bi-directional ones for ease of replay; and (iii) produces a log that enables a high degree of parallelism during replay. In addition, the backend pass can flexibly produce a log with the desired tradeoff between degree of replay parallelism and log size.

Figure 3 shows the system. We consider this backend pass to be a fundamental enabler of a hardware-assisted R&R scheme that is hardware-unintrusive, supports application-level R&R and allows a maximum (and also settable) degree of replay parallelism. In the following, we show how Cyrus addresses each of the challenges.

3.2 Application-Level R&R

To understand which interprocessor dependences need to be recorded for application-level R&R, consider the model of Figure 4(a). At



Type of Interaction		Type of Dependence	Cyrus Action	Example
$srcM$	$srcM_{app} \rightarrow dstM_{app}$	Correct	Record	(1)
	$srcM_{OS} \rightarrow dstM_{app}$	Correct	Record	(2)
	$srcM_{app} \rightarrow dstM_{OS}$	Early or Unrelated	Record & Defer/Discard	(3)
	$srcM_{OS} \rightarrow dstM_{OS}$	Early or Unrelated	Record & Defer/Discard	(4)
	$srcM \rightarrow dstN$	Unrelated	Record & Defer/Discard	(5)
M-preemption in context switch		Serialization (Multiple)	Record & Defer/Discard	(6)
$srcN$	$srcN \rightarrow dstM$	Correct, Early or Unrelated	Ignore	(7)
	$srcN \rightarrow dstN$	Unrelated	Ignore	(8)

(c)

Figure 4. Characterizing the types of interprocessor interactions.

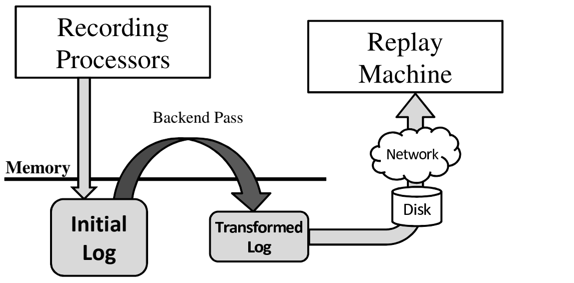


Figure 3. Overview of the Cyrus system.

any given time, a processor may run a process that is being recorded or one that is not. We call such times *M* and *N* for monitored and non-monitored, respectively. During the *M* time, the processor may run application code (M_{app} time) or OS code on behalf of the application (M_{OS} time).

In this environment, there are several types of interactions between a source and a destination processor. The destination processor (*dst*) is the one that initiates a coherence action and receives a response — e.g., it misses in its cache or sends an invalidation. The source processor (*src*) is the one that sends the response. Figure 4(c) shows the types of interactions possible, together with the corresponding type of dependence involved, the action taken by Cyrus, and a dependence example from Figure 4(b). In the next few paragraphs, we describe each of the interactions and how Cyrus handles them.

We start by describing the interactions where the source is a processor running a monitored process (*srcM*), as shown in the first group of entries in Figure 4(c). If the destination is a processor running monitored application code ($srcM_{app} \rightarrow dstM_{app}$ or $srcM_{OS} \rightarrow dstM_{app}$), this is a correct dependence within the recorded application — the recorded application misses in the cache or sends an invalidation. Therefore, Cyrus records it in the log.

If the destination is running the OS on behalf of a monitored application ($srcM_{app} \rightarrow dstM_{OS}$ or $srcM_{OS} \rightarrow dstM_{OS}$), two cases are possible. One is that the OS is accessing data that will later be accessed by the monitored application code (i.e., it is effectively prefetching the data); the other case is that the OS is accessing data that is unrelated to the monitored program and happens to be in the source processor's cache. In the first case, we must record this correct dependence that is detected *early*; in the second case, we must discard it. Since Cyrus does not know which case it is, it conservatively records it in the initial log. Later, the backend pass will find which case it is, and either set the destination of the dependence to be the next M_{app} block running on the destination processor (an action called "deferring the dependence"), or discard it. In Figure 4(c), we call this action "Record & Defer/Discard".

Finally, if the destination processor is running a non-monitored process ($srcM \rightarrow dstN$), the action pertains to unrelated data, and does not need to be recorded. However, for ease of implementation as we will see, Cyrus records it as in the previous case.

The next row in Figure 4(c) corresponds to a context switch where an *M* process is preempted. After the preemption, data left in the cache may be requested by other processors. To avoid having to log such interactions, in a context switch, Cyrus conservatively records one dependence from this processor to every other processor in the machine. This is called a *serialization* because it effectively serializes the last block of the current processor prior to the context switch before the current block of every other processor. Cyrus records such dependences in the initial log and the backend pass will defer or discard them. Specifically, any such dependence will be discarded if no monitored process ever runs on the destination processor.

The final two rows in Figure 4(c) correspond to when the source is a non-monitored process (*srcN*). In this case, a $srcN \rightarrow dstM$ interaction can be correct, early or unrelated, while a $srcN \rightarrow dstN$ interaction is unrelated. Cyrus ignores each of these dependences. This behavior is correct since any such dependence is guaranteed to be superseded by one of the serialization dependences described above.

Overall, as shown in Figure 4(c), to ensure correct application-level recording, Cyrus only needs to log events when a processor running the application or OS code of a *monitored* process is: (i) either the *source* of a dependence (i.e., at the request of another processor, it provides a line from its cache or invalidates a line from its cache) or (ii) suffers a context switch. Still, we need a later pass to fix or discard certain dependences.

3.3 Unintrusive Recording Hardware

The Cyrus hardware is shown at a high level in Figure 5. Each core has a *Race Recording Unit* (RRU) associated with the cache controller. For simplicity, we show the RRU for a system with a single-level cache hierarchy. In this design, the RRU observes the bus transactions, and is also informed of processor requests and cache evictions.

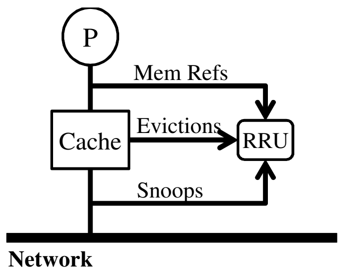


Figure 5. High-level view of the Cyrus hardware.

To keep the design unintrusive, we require that it does not change the cache coherence protocol in any way — including, for snoopy schemes, not adding new fields to messages. As explained

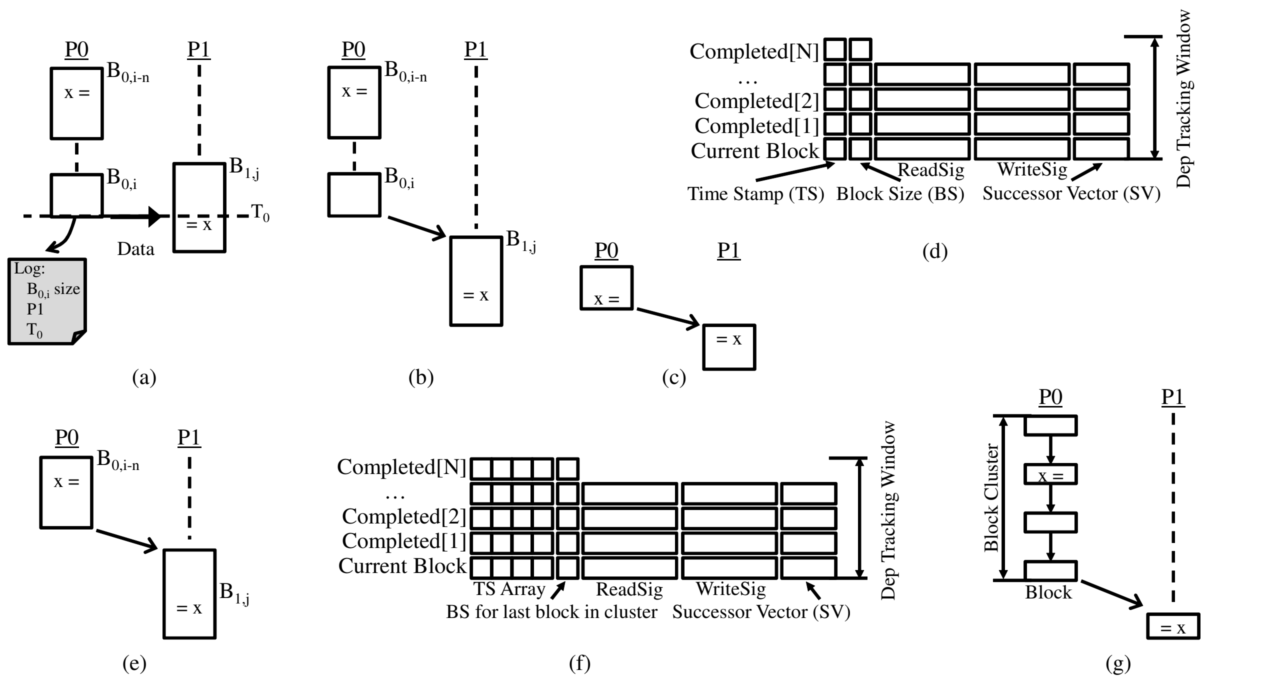


Figure 6. Recording dependences in Cyrus.

in Section 2.2.2, the implication for snoopy schemes is that, when an interprocessor dependence takes place, only the source processor knows about it and can record it.

Consequently, Cyrus operates as follows. When a processor (P_0) executing a block ($B_{0,i}$) of a monitored process observes a bus transaction to which its cache needs to respond (by invalidating a line and/or providing a line), the RRU hardware is signaled. The RRU terminates $B_{0,i}$ and (in a naive design) creates a local log entry composed of: $B_{0,i}$'s block size (BS) in number of instructions, the ID of the processor that initiated the transaction (the dependence's destination processor), and the current time. Cyrus counts time as the number of bus transactions so far, which is known by and is consistent across all processors. We call such number the Time Stamp (TS). The destination processor is unaware that a dependence has been recorded.

This information is all that Cyrus needs to log, and requires no modification to the coherence protocol. However, to ease the replay, we will need to have bidirectional dependence information as in Figure 2(e). Such information is generated from the initial log by the backend pass and is stored in the final log (Section 3.5).

3.4 Replay Parallelism

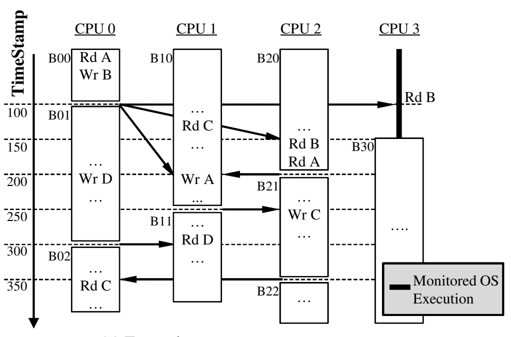
With the naive approach described, the log records an inter-thread dependence between the blocks that are running when the coherence action is detected. This approach enables only limited replay parallelism. For example, consider Figure 6(a), where processor P_0 writes to variable x in block $B_{0,i-n}$ and processor P_1 reads x in block $B_{1,j}$ at time T_0. The figure also shows the log entry. Since the coherence action occurs while P_0 is executing block $B_{0,i}$, the logged entry implies a dependence and a replay order between blocks $B_{0,i}$ and $B_{1,j}$ as in Figure 6(b) — even though the source of the dependence is much earlier, and the destination is deep inside the destination block. To extract maximum parallelism, we would like the log to represent the execution as in Figure 6(c), where processors P0 and P1 overlap their execution as much as possible.

To approach this ideal capability, Cyrus can be designed to use a small Maximum Block Size and to track multiple blocks at a time. The idea is for the RRU to keep information for the most recent *N* completed local blocks. These completed blocks plus the currently-running block form the *Dependence-Tracking Window*, from which dependence sources are tracked. Each of these blocks (except for the oldest one) has a read and a write signature register (ReadSig and WriteSig), which hash-encode with a Bloom filter [4] the addresses of the lines read or written by the block (Figure 6(d)). When the local cache responds to an incoming coherence request, the hardware checks the address of the request against the signatures in reverse order, starting with the ones for the currently-running block. When one of the signatures matches the address, we know that the corresponding block was the source of the dependence, and record it. This allows us to precisely place the source of the dependence in the right block. If none of the signatures matches the address, the oldest of the N completed blocks is assumed to source the dependence.

If the currently-running block is the source of the dependence, it is terminated. In this case, all the blocks are shifted up, the old one is written to the log, and a new one starts. With this support, the log records the example dependence as in Figure 6(e), where the source of the arrow is closer to the source access. This enables more replay parallelism. Karma [3] uses this approach for *N*=1.

Figure 6(d) shows other fields of each entry in the dependence-tracking window, which we will discuss later.

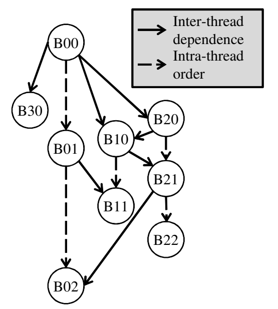
Unfortunately, even this enhanced approach has some shortcomings. To have a large dependence-tracking window, *N* needs to be high, which means that many pairs of costly signatures are needed. The alternative is to increase the block size, therefore needing a lower *N*. In this case, however, the source of the dependence may be far from the end of the source block, and the destination of the dependence may be far from the beginning of the destination block. In the worst case, the source and destination references are separated by twice the maximum block size.



(a) Execution

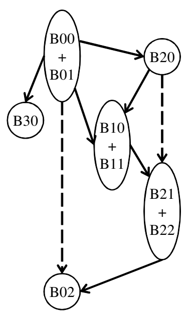
CPU	TID	TS	SIZE	Successor Vector			
0		100		-	200	150	100
0		300		-	300	0	0
0		370		-	0	0	0
1		250		0	-	250	0
1		360		0	-	0	0
2		200		0	200	-	0
2		350		350	0	-	0
2		390		0	0	-	0
3		385		0	0	0	-

(b) Initial Log



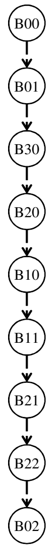
CPU	TID	SIZE	PTV				STV			
0			-	0	0	0	-	1	1	1
0			-	0	0	0	-	1	0	0
0			-	0	1	0	-	0	0	0
1			1	-	1	0	0	-	1	0
1			1	-	0	0	0	-	0	0
2			1	0	-	0	0	1	-	0
2			0	1	-	0	1	0	-	0
2			0	0	-	0	0	0	-	0
3			1	0	0	-	0	0	0	-

(c) MaxPar



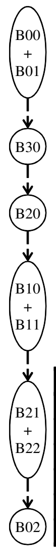
CPU	TID	SIZE	PTV				STV			
0			-	0	0	0	-	1	1	0
0			-	0	1	0	-	0	0	0
1			1	-	1	0	0	-	1	0
2			1	0	-	0	0	1	-	0
2			0	1	-	0	1	0	-	0
3			1	0	0	-	0	0	0	-

(d) Stitched



TID	SIZE

(e) Serial



TID	SIZE

(f) StSerial

Figure 7. Example of execution and resulting Cyrus logs. The table in (b) depicts the initial block data dumped by the processors, while the other tables show the results of the different backends, encoding the corresponding DAGs. In the tables, dashes indicate entries corresponding to dependencies to the processor itself. These are never used.

To address this problem, Cyrus introduces the concept of *Block Clusters*. Block clusters use the observations that: (i) to reduce the separation between the beginning of the destination block and the destination reference, we need small blocks; and (ii) to reduce the separation between the source reference and the end of the source block, we need a large dependence-tracking window which, to be cheap, needs large blocks. Hence, in block clusters, we use small blocks and *combine them to make them appear* as large blocks. In practice, reducing the separation in (i) is more important than in (ii). The reason is that any separation in (i) directly slows down the replay execution relative to the recorded execution.

With block clusters, we use a small block size, but we group multiple consecutive blocks into a cluster for the purpose of tracking dependence sources. The RRU's dependence-tracking window contains multiple block clusters. Each one has a *single* ReadSig and WriteSig signature pair that contains the addresses accessed by all the blocks in the cluster. If the address of an incoming coherence transaction matches the signature, then the source of the dependence is assumed to be the *last* block of the cluster.

Figure 6(f) shows the case of four blocks per cluster. When a block executes and exhausts its maximum size without recording a dependence, its termination time stamp is stored in TS[i] and the next block in the cluster starts. Note that for such blocks, Cyrus does not need to store the size explicitly because it is known to be the maximum block size. When a dependence is found in the running cluster's signatures, the running block is assumed to be the source; that block is terminated, its time stamp and size (BS) are saved, and the cluster is terminated. All the cluster information is shifted up and a new cluster is started. Future dependence sources found in the signatures of any cluster in the RRUs dependence-tracking window, are assigned to the last block in that cluster.

With this support, Cyrus provides a large dependence tracking window, and at the same time, reduces the distance between the beginning of the destination block and the destination reference. This is seen in Figure 6(g). The result is more replay parallelism.

3.5 Backend Software Pass

The initial log generated by the recorder has unidirectional dependence information only, and contains some dependences that need to be fixed-up or discarded for application-level R&R. To correct these issues, a backend software pass processes the log, creating a final log that is highly amenable to parallel replay. In addition, the backend pass can format the log for different tradeoffs between replay parallelism and log size.

3.5.1 Transforming the Log

Each entry of the initial log contains the following base information for one block: the ID of the CPU that executed the block and the block's termination time stamp (TS). In addition, if this is the last block of a cluster that sourced dependences, the information also includes the block size (BS) in number of instructions, and successor vector (SV). The latter has one entry for each of the other processors in the machine. SV entry *i* is either null or has the TS when the current cluster sourced a dependence to processor *i*. If the cluster sourced multiple dependences to processor *i*, SV[i] has the TS of the earliest one — which is the most conservative.

Figures 7(a) and (b) show an example execution with 4 processors and the resulting initial log, respectively. Each row in Figure 7(b) indicates a block dumped by the corresponding processor. In all of the tables in Figure 7, TID is the ID of the thread to which the block belongs. It is provided by the OS driver that controls the RRU. The hardware itself is oblivious to the notion of threads (Section 4.2).

For simplicity, we assume one block per cluster and two clusters. In Figure 7(a), we can see that, as soon as a processor sources a dependence for a datum accessed in the current block, it terminates the block. If the dependent datum has not been accessed in the current block but in past blocks, the current block is not terminated. For example, at TS=200, CPU1 performs "Wr A". Since this access does not conflict with block B01 of CPU0, B01 is not cut and the dependence is assigned to B00, instead.

In Figure 7(a), all the dependences are Correct ones except for the one from CPU0 to CPU3, which is an Early or an Unrelated one. In the figure, CPU3 is initially executing the OS on behalf of the monitored process. The OS accesses variable *B*, creating a dependence with processor 0, which terminates its block B00. According to Cyrus's operation, it has to record the dependence, and rely on the backend pass to either defer it or discard it. Since, as shown in the figure, CPU3 later executes block B30 of the monitored application, the backend pass sets the destination of the dependence to be B30 — i.e., defers the dependence. This is required for correctness, as block B30 could next silently access variable *B*. If, instead, CPU3 never executes any block of the monitored application, the backend pass discards the dependence.

Similarly, if the OS preempts a monitored thread (i.e., on a context switch), it uses the programming interface of the RRU (Section 4.2) to create Serialization dependences with all other processors; they are eventually deferred or discarded by the backend pass.

As the backend pass processes each entry of the initial log, fixing up and discarding dependences, it also records, in each dependence's destination block, which other block is the source. Encoding such bidirectional dependence information will enable parallel replay. Hence, it incrementally builds the dependence DAG that captures all the necessary ordering of blocks for a deterministic replay.

To encode the resulting DAG in the final log, we adopt and generalize the representation used by Karma [3], in which, instead of representing dependences as source-destination block pairs, we use a token-based representation. Assume a dependence between block B1 of processor P1 to block B2 of processor P2. To enforce the dependence during the replay, the log will have B1 send a token to P2 after its execution, and B2 wait for a token from P1 before starting. Both source and destination are processor numbers rather than block numbers.

Our baseline backend pass algorithm and the resulting transformed log are called *MaxPar* because they expose maximum replay parallelism obtainable from the initial log. An entry in the MaxPar log contains the following information for a block: the IDs of the CPU and thread that executed it, its size, the Successor Token Vector (STV), and the Predecessor Token Vector (PTV). The STV is a bit vector with as many bits as other processors. Bit *i* is set if a successor of the block is in processor *i*. The PTV is an array of counters with as many entries as the STV. Entry *i* counts the number of predecessors that the block has in processor *i*. For our example, the resulting MaxPar log and execution DAG are shown in Figure 7(c).

With the MaxPar log, replay will involve processors executing in parallel, synchronizing only on dependences — figuratively passing tokens between them. In the following, we outline the MaxPar algorithm and then consider other algorithms.

3.5.2 MaxPar: Algorithm for Maximum Parallelism

In this discussion, we call a block *open* while it still has unresolved successors or predecessors, and *resolved* otherwise. After a block becomes resolved, the backend can write it to the transformed log as soon as all of the previous blocks of the same processor are written. After writing, we say the block is *retired*.

Figure 8 shows the high-level pseudocode of the algorithm. Each processor is represented by a proxy object. A proxy keeps track of its open blocks in a chronologically ordered list. Also, it keeps a data structure (called *waitingList*) for blocks of other proxies that, according to their Successor Vectors (SV), have unresolved successors in this proxy.

The algorithm processes blocks in batches of consecutive blocks from the same processor. When a new block is added to proxy *P*, its SV is checked, and for each successor, the block is added to the *waitingList* of the proxy for that successor. Next, since a new batch has been added to *P*, blocks in the *waitingList* of *P* are checked to see if their dependences can be resolved. To resolve a dependence that was recorded at time *t*, the open blocks of *P* are binary-searched to find the first block whose time stamp is larger than *t*. This block is the destination of the dependence. The dependence is recorded by setting the appropriate entry in the STV of the source block and incrementing the corresponding entry in the PTV of the destination block. As dependences are resolved, periodically, a trimming pass is run to retire the resolved blocks from the proxies by writing them to the transformed log. Before writing a block to the log, MaxPar tries to merge it with the previous block of the same processor, if that does not reduce the recorded parallelism. Specifically, assume that the previous block is B_0 and the current block is B_1. If B_0 has no successors (other than B_1) and B_1 has no predecessors (other than B_0), merging B_0 and B_1 will not change the recorded parallelism.

```
AddBatch($batch$, $proxy$):
  for each block $b$ in the batch
    for each valid successor processor $s$ in $b.sv$
      /* call $sp$ the proxy for processor $s$ */
      add $b$ to $sp.waitingList[proxy]$
  foreach other proxy $op$ in the system:
    foreach block $b$ in $proxy.waitingList[op]$
      find $dep$
      /* $dep$ is the block in proxy that is the successor of $b$ */
      if ($dep$ is not NULL)
        remove $b$ from $proxy.waitingList$
        mark this dependence as resolved in $b$
        update $b.STV$ and $dep.PTV$
  if enough time has passed since last trimming
    Trim()
Trim():
  for each proxy $p$ in the system
    for each block $b$ in $p$
      if $b$ is old enough and all its predecessors are retired
        write $b$ to the transformed log
        remove $b$ from $p$
MaxPar():
  while (there are batches)
    batch ← next batch
    AddBatch(batch, proxies[batch.cpu])
```

Figure 8. High-level description of the MaxPar algorithm.

There are some details that are not shown in Figure 8. One difficulty is how to tell whether all the predecessors of a given block have been seen and resolved. Processors dump their block data independently and in batches (not one by one) and it is quite possible that when a block is dumped, some of its predecessors are still in their respective processors.

The solution here is to make sure block data do not indefinitely reside in processor buffers and will be dumped if they have been around for a preset amount of time, called *Maximum Silence Period* (or MSP) — e.g., 100000 timestamp units. Consider block B_i of proxy P_i. With the above guarantee, which we call the *Bounded Silence* guarantee, if the maximum timestamp of all the blocks dumped so far is larger than $B_i.ts + MSP$, then we know for sure that all the predecessors of B_i have also been dumped and their dependencies have been resolved (please recall that the predecessors of B_i have smaller timestamps than B_i itself). At this time, we can consider B_i to be *old enough* (Figure 8) to be retired.

Another important question concerns Early or Unrelated dependencies (See the table in Figure 4). Assume B_i records P_j as a successor but, since P_j is not running any monitored threads, it will not dump any blocks for a long time (or maybe forever). How should the recorded dependence be resolved?

To handle this case, again, we use the Bounded Silence guarantee. If B_i recorded the dependence at time t and P_j dumps no blocks before time $t + MSP$, then it is guaranteed that no monitored block existed on P_j at time t. Hence, it suffices to attribute the dependence to the next dumped block of P_j, or it can be safely discarded if P_j never dumps. The token-based representation enables an efficient implementation in this case. Each proxy, P_j in this example, has a vector of counters which count the number of Early dependences from other processors (P_i in this case). Let us call this vector the Early Token Vector. At time $B_i.ts + MSP$, B_i can consider its dependence resolved and send an Early token to P_j by incrementing entry i of the Early Token Vector in P_j. When the next block of P_j is dumped, the counters in the Early Token Vector are added to the PTV of that block and then they are reset.

The MSP-based techniques described above imply that, at time $B_i.ts + MSP$, B_i is *old enough* (term used in Figure 8) to be retired, since all of its predecessors and successors are guaranteed to have been resolved by that time.

3.5.3 Trading-off Replay Parallelism for Log Size

As the backend pass generates the final log, it can transform it in ways that affect the size of the log and the potential for parallelism in its replay. This provides substantial flexibility (Figure 9). For example, when R&R is used for on-line intrusion analysis or fault tolerance, it typically requires high-speed replay. In this case, the log format should be such that it enables highly-parallel replay. Such format is MaxPar. On the other hand, when R&R is used for (off-line) software debugging, replay speed is less important. Hence, we likely prefer a format that needs less log space at the expense of offering less parallelism. Such format is called *Stitched*. This format is also suitable for intrusion analysis or fault tolerance when the application is I/O- or memory-intensive. This is because, in such scenarios, replay is typically faster than recording. Finally, when small log size is paramount, even at the expense of replay speed, the *Serial* or *StSerial* formats should be used. In the following, we discuss these formats.

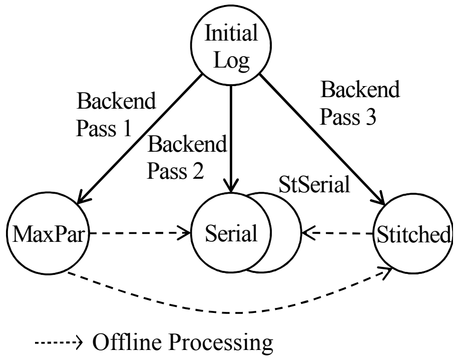


Figure 9. Flexibility of the backend pass.

Stitched: Reduced Parallelism. The Stitched format uses less log space than MaxPar, but it offers less parallelism for replay. Compared to MaxPar, Stitched merges consecutive blocks of an application thread into a *Stitched* block as long as this process does not introduce cycles in the graph.

Accurately detecting cycles on-the-fly can be computationally intensive. There are conservative techniques that can be used instead. One such technique involves using Lamport Scalar Clocks. Specifically, each block is assigned a clock that should be strictly larger of those of its predecessors. While merging a sequence of consecutive blocks into a stitched block *SB*, the algorithm watches the clock values of the all the predecessors of the blocks in *SB*. As long as all of these predecessors have clock values not larger than that of the first block in *SB*, no cycle can be created and we can safely merge the blocks. If, however, one of the predecessors of the next block to stitch violates this condition, we stop merging and start a new block sequence.

Figure 7(d) shows the Stitched execution DAG and log for the example in Figure 7(a). Compared to the MaxPar algorithm (Figure 7(c), we have combined blocks into bigger blocks, hence reducing the number of log entries but also decreasing the parallelism of the DAG available to the replayer.

Serial: Sequential Replay. When having a very small log is very important, even at the expense of any parallelism in the replay, we use the Serial format. In this case, we create a total order of blocks. This format is generated with a simple topological sort on the dependence DAG. It can be generated either on-the-fly in the backend or off-line after the MaxPar log has been created. Each serial log entry only needs to contain the thread ID and the size of a block — the rest of the information is unnecessary.

Figure 7(e) shows the Serial execution DAG and log for the example in Figure 7(a). Compared to the MaxPar algorithm (Figure 7(c)), we have created a total order of blocks, disabling any parallel replay, but substantially reducing the log size.

StSerial: Stitched Sequential Log. Finally, we can reduce the log size even more if we apply the Serial algorithm to a DAG generated by Stitcher. The result is called StSerial. Compared to MaxPar, we reduce both the number of log entries and the size of each of them. The replay is also serial. Figure 7(f) shows the StSerial execution DAG and log for the example in Figure 7(a).

Advanced Uses. Other flexible uses of the backend pass are possible. One use is for the backend to dynamically change the format of the log at different phases of a program's execution. This scenario may be useful when R&R is used in online-replay scenarios (e.g., fault tolerance) where a secondary server follows the execution of a primary one. To reduce the bandwidth required to transfer the log from the primary to the secondary server, the backend may usually use the Stitched format. However, in sections of the program when fast replay is needed (perhaps because the execution becomes compute intensive), the backend may switch to MaxPar.

Once can also think of a transformation to reduce the number of processors in the log. The transformation involves combining the entries from two or more processors into one. In practice, this transformation is unlikely to be useful since (i) it does not change the number of log entries and just reduces the size of PTV and STV, and (ii) the replay can already be done with fewer processors than used for recording, even with an unmodified log; we only need that a replay processor execute blocks from multiple recording ones.

4. Implementation Issues

4.1 Race Recording Unit (RRU) Design

The Cyrus hardware consists of a *Race Recording Unit* (RRU) associated with the cache controller of each processor (Figure 5). When a processor is executing a monitored process, if its cache observes a bus transaction that induces a dependence with data previously accessed by the processor, the RRU records that dependence.

Figure 10 shows the hardware inside the RRU. It has four components: Tracked Block-Cluster Buffer, Block-Cluster Buffer, Time Counter, and Eviction Signature. The Time Counter is the global clock, obtained by counting the number of coherence transactions on the bus. It has the same value in all the cores.

The Tracked Block-Cluster Buffer (TBCB) implements the dependence-tracking window described in Section 3.4. It contains information about several block clusters: the currently-running one

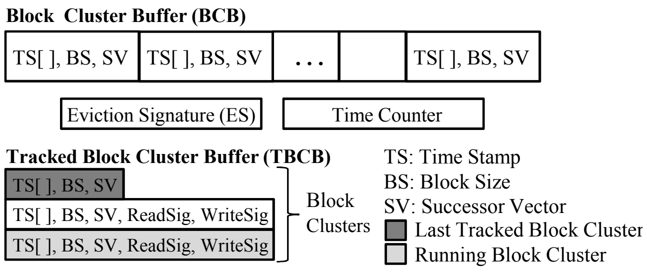


Figure 10. Race Recording Unit (RRU) design.

(*Running* in Figure 10) and the N most recently-completed ones. Of these, the earliest one is called *Last Tracked* in Figure 10. All block clusters in the TBCB except for the Last Tracked one have read and write signature registers. These registers hash-encode and accumulate with a Bloom filter [4] the addresses of all the lines read or written by all the blocks in the corresponding block cluster.

To understand how the TBCB works, assume first that there is no cache overflow; we consider cache evictions later. When a request on the bus *hits in a cache*, the cache's RRU checks the requested address against the signature registers in its TBCB — a bus write is checked against read and write signatures, while a bus read only against write signatures. The checks are performed in order, starting with the signatures of the Running cluster and proceeding to older clusters. The goal is to find which cluster is the latest source of the dependence.

If there is a hit in the signatures of the Running cluster, the current time stamp is saved in the cluster's Successor Vector (SV) entry for the requesting processor. Moreover, the cluster terminates and the current block size is saved in the cluster's BS field. In addition, the whole TBCB is shifted upward, pushing the contents of the Last-Tracked cluster into the Block-Cluster Buffer and a new Running block cluster begins.

If, instead, there is a hit in the signatures of an older cluster in the TBCB, we save the current time stamp in that cluster's SV entry for the requesting processor. Finally, if instead, the request does not hit in any signature, we conservatively assume that the Last Tracked cluster is the source of the dependence. In this case, we save the current time stamp in that cluster's SV entry for the requesting processor. In all cases, if the corresponding SV entry is already set to a smaller timestamp, we do not update it.

The case when processor P0 writes a variable, then P1 reads it and then P2 reads it, correctly triggers the logging of a dependence P0→P2 (in addition to P0→P1). The reason is that, although the P2 read does not induce any coherence operation on P0's cache, P0's cache hits and, as a result, P0's write signatures are checked. If the P0 write occurred during its Last Tracked cluster, there is no signature, but Cyrus still records a dependence by default. This would be conservative (although correct) if P0 had only read, not written. Fortunately, recording conservative dependences so far in the past is not expected to hurt replay parallelism.

If the current block in the Running cluster reaches its maximum size without sourcing a dependence, it terminates, saving the current time stamp in the corresponding TS field of the Running cluster. Then, a new block starts. If all the blocks of the Running cluster have been exhausted, the cluster terminates, and the whole TBCB is shifted upward. As information on an old cluster is displaced from the tail of the TBCB, it is dumped into the Block Cluster Buffer (Figure 10). When the Block Cluster Buffer is about to fill up, its contents are appended to the tail of an in-memory buffer provided by the operating system (Section 4.2).

4.1.1 Eviction Signature

Caches suffer line evictions. In the design presented, when a cache evicts a line, its RRU loses the ability to record inter-processor dependences on data from that line. Indeed, future bus transactions on that line would not find the data in the cache and, therefore would not trigger checks in the dependence-tracking window.

To eliminate this problem, when a clean or dirty line is evicted from a cache, Cyrus hash-encodes and accumulates its address into the RRU's Eviction Signature (ES) (Figure 10). Then, when a transaction is observed on the bus, Cyrus checks if the address hits in the cache or in the ES. If it hits in at least one, Cyrus proceeds with checking the block clusters in the TBCB.

The ES should be regularly cleared to avoid collecting many addresses that could cause address aliasing. Fortunately, every time that the OS preempts a thread that is being monitored, the local RRU records a Serialization dependence with all other processors in the system (Section 3.2). At that point, the ES can be cleared. Also if, at any time, the ES contains too many addresses, Cyrus simply terminates the current block, records a dependence from it to all the other processors, and clears the ES.

4.2 OS Design for R&R

Figure 11 shows the overall architecture of our R&R system, where the dashed boxes indicate the Cyrus extensions. We base our design on that of Capo [24]. The Replay Sphere Manager (RSM) is the basic kernel module that controls R&R. We organize the OS extensions according to the sources of non-determinism. Hence, we have two components: one for input non-determinism and one for memory-interleaving non-determinism.

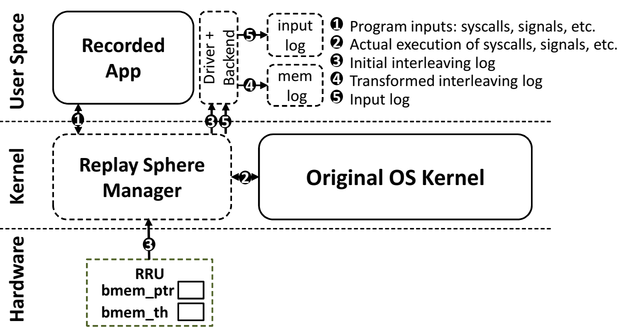

User Space
Kernel
Hardware
Recorded App
Driver + Backend
input log
mem log
❶ Program inputs: syscalls, signals, etc.
❷ Actual execution of syscalls, signals, etc.
❸ Initial interleaving log
❹ Transformed interleaving log
❺ Input log
Replay Sphere Manager
Original OS Kernel
RRU
bmem_ptr
bmem_th

Figure 11. Overall architecture of our R&R system, where the dashed boxes are the Cyrus extensions. The numbers correspond to the generation of logs during recording.

We use a driver program to launch a R&R sphere to perform record or replay. In record mode, the RSM generates the input log, while the RRUs generate the memory-interleaving log. The data transfers proceed as shown with numbers in the figure. As the initial memory interleaving log is generated, the backend pass runs on a dedicated processor and transforms it. In replay mode, the driver reads the input and memory interleaving logs and passes them to the RSM, which consumes them. In addition, the RSM uses performance counters to detect block termination, as we will see.

4.2.1 Input Non-Determinism Module

This module is similar to Capo's [24]. There are four different sources of input non-determinism that Cyrus handles: system calls, data copied to/from the user address space, signals, and non-deterministic processor instructions. Unlike Capo, which uses `ptrace`, we have implemented this component as a Linux kernel module to improve the performance and make it easier to integrate with the memory-interleaving module. Since this module uses per-thread data structures, it is easy to support multiple replay spheres simultaneously.

4.2.2 Memory-Interleaving Non-Determinism Module

Using the RRU at Record Time. As the RRU generates the log, it dumps it into an OS-allocated block of memory called *bmem*. The RRU offers a minimal interface for the OS to manage and virtualize the hardware during recording. This interface contains: (i) a pointer to bmem (*bmem_ptr*), and (ii) a threshold register that indicates the point at which bmem is about to overflow (*bmem_th*). When bmem_th is reached, an interrupt is triggered, and a new bmem is allocated.

The OS manages the per-thread bmem areas and virtualizes these hardware registers, so that different threads can use the hardware without mixing up their data. In particular, this involves making sure that a valid bmem_ptr is configured before recording begins, allocating a fresh bmem when the previous one is full, and ensuring that, on a context switch, all the recorded data is dumped into the bmem and a Serialization dependence is recorded. This is done by writing to a RRU-specific control register. Also, the OS appends to each bmem buffer the ID of the thread to which the blocks in the buffer belong. Thus, the RRU itself does not need to know about threads.

Enforcing the Recorded Interleavings during Replay. The OS is able to recreate the recorded interleavings by allowing each block to start its execution only after all of its predecessors have executed. For this, it uses mechanisms to detect block termination and to synchronize predecessor/successor blocks.

To detect block termination, Cyrus uses performance counters similar to those available in commodity processors. The replaying thread configures the counter so that an interrupt is triggered when the number of instructions executed equals the needed block size. Cyrus assumes synchronous and precise interrupts for this, i.e., the interrupt is generated just before the first instruction of the next block is executed.

To synchronize predecessor/successor blocks, Cyrus uses a software solution. When a block finishes, it should send tokens to its successors, and before the next block starts, it should wait for enough tokens from its predecessors. Cyrus implements this in the RSM (i.e., in the OS and without modifying the application code) using software semaphores. There is a semaphore for each (P_i, P_j) pair of different record-time processors. This semaphore represents tokens sent from P_i to P_j. After a block terminates, the OS first sends tokens to the appropriate semaphores for its successors. It then reads the next block from the memory-interleaving log and for each recorded predecessor, it uses the appropriate semaphore to wait until enough tokens are received from that predecessor.

5. Evaluation Setup

For our evaluation, we augmented the Linux 3.0.8 kernel with a Replay Sphere Manager (RSM). The OS drives and virtualizes the Cyrus architecture modeled with the Simics [22] full-system simulator. The OS changes include the input non-determinism module and the memory-interleaving non-determinism module that manage the two logs.

We use Simics to model an x86-based chip multiprocessor with a single level of private caches that are kept coherent using a snoopy-based MESI cache coherence protocol. Table 1 shows the parameters of the architecture. Unless explicitly specified, we perform the parallel record and replay runs with our applications running on 8 processors. The backend pass uses one additional processor. The baseline RRU configuration uses blocks with at most 4K instructions. It has 2 tracked block clusters (N = 1), and hence

Processor and Memory System Parameters	
Chip multiprocessor	Bus-based with snoopy MESI protocol 8 proc. for application; 1 for backend
Processor	Single issue, x86 ISA 2GHz clock
L1 Cache	64KB size, 64B line, 4-way assoc. 1 cycle hit, 2-bit LRU replacement
L1 Cache Miss Latency	10-cycle round-trip to another L1 100-cycle round-trip to memory

Cyrus Parameters	
Read & write signature	4×512bits & 4×256bits H3 Bloom filter
Eviction signature	4×512bits H3 Bloom filter
# Tracked block-clusters	2
# Blocks per block cluster	16
Maximum block size	4K instructions
Block cluster buffer (BCB)	8 entries

Table 1. Parameters of the simulated hardware.

we only need one pair of signatures per RRU. We use 16 blocks per cluster. We execute 10 applications from the SPLASH-2 suite, which we run from beginning to end.

6. Evaluation

6.1 Recording & Backend Overhead

We first examine the initial log size and whether it can become a bottleneck for Cyrus. Figure 12 shows the growth in the rate of the *initial log* generation as the number of processors increases. This is a temporary log and, hence, it is not compressed. The time unit in this figure is 1K cycles of total execution time (i.e., 0.5 μsec assuming a 2GHz clock). On a system with 8 processors, this means 8K-cycles worth of instruction execution. As seen in the figure, on average, the logging rate grows about linearly with the number of processors. However, a simple calculation shows that even with 8 processors, the average log generation rate is less than 29 MByte/sec. This is far less than the bandwidth of the system bus in current machines (which is typically on the order of several GByte/sec). For this reason, it is not likely that the initial log generation can become a bottleneck.

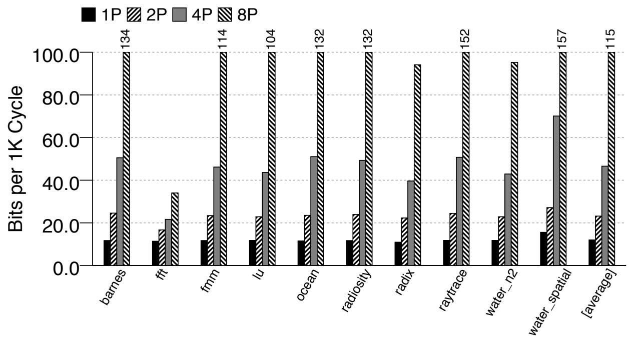


Figure 12. Initial log size for different numbers of processors, shown in terms of the number of bits generated per 1K cycles of total execution time.

Figure 13 examines the overhead of recording with and without the backend. The figure compares the execution time of the benchmarks when the Cyrus hardware is not enabled (*NoMemLog*), when Cyrus records the memory-interleaving log (*MemLog*), and when, in addition, the backend pass runs (*MemLog+Backend*). In all cases, the RSM is recording the input non-determinism log. For each benchmark, the bars are normalized to *NoMemLog*. The applications execute with 8 processors.

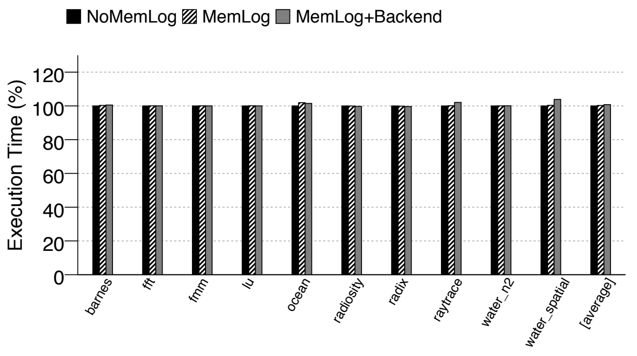


Figure 13. Overhead of recording with and without the backend pass for 8-processor runs.

The figure shows that the overhead of recording the memory-interleaving log, either with or without the backend pass, is negligible. The backend pass induces little overhead because it uses a dedicated processor. While this fact increases the system cost, it allows Cyrus' R&R to be non-intrusive to the hardware.

6.2 Comparing Different Backend Pass Algorithms

We now compare the different Cyrus' backend pass algorithms. We compare the available replay parallelism and the log size of the Serial, StSerial, Stitched, and MaxPar formats (Figures 14 and 15). To estimate the available replay parallelism, we use the Normalized Inverse Critical Path Length (NICPL) of the dependence graph in the log. To measure the NICPL of a benchmark, we start by computing the length of the longest chain of dependences (in terms of number of instructions) in the log. This is the critical path length. Then, we divide the critical path length obtained with a fully-serial log like Serial or StSerial (which is the number of instructions in the benchmark) by the critical path length obtained with a given log. The result is the NICPL of the log. Thus, a higher NICPL value indicates more parallelism in the recorded dependence graph.

Figure 14 compares the NICPL values for the Serial, StSerial, Stitched, and MaxPar formats. We can see that, on average, MaxPar and Stitched provide a replay parallelism of 5 and 3, respectively. Most of the applications can benefit considerably from MaxPar and, to a lesser extent, from Stitched.

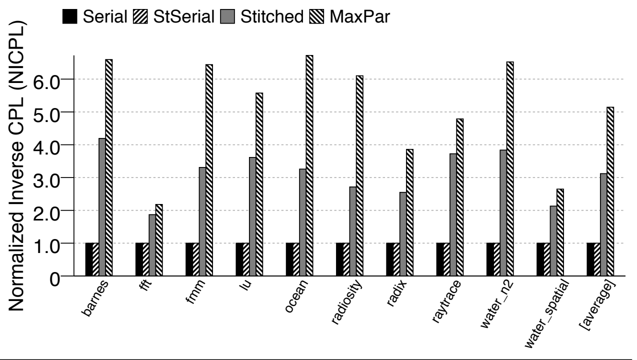


Figure 14. Normalized Inverse Critical Path Length (NICPL).

Figure 15 shows the resulting size of the logs. We compressed the logs with bzip2 and report the number of bits used per 1K instructions. We see that, on average, MaxPar and Stitched generate about 2 bits/Kinstruction, while Serial and StSerial generate 1 bit/Kinstruction. Stitched is not capable of considerably reducing the log size over MaxPar's because, as mentioned in Section 3.5.2, MaxPar already merges many of the recorded blocks while retaining maximum parallelism. On the other hand, StSerial is only slightly more space efficient than Serial. Finally, the figure shows that water_spatial produces large log files. This is because it synchronizes frequently, which creates many small blocks.

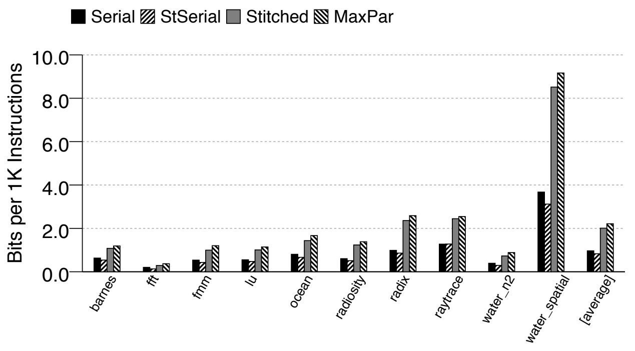


Figure 15. Log size in bits per 1K instructions.

Overall, comparing MaxPar to Serial, we conclude that, with a 2x bigger log, MaxPar delivers a 5x higher parallelism. This is likely to be a very good tradeoff in some R&R applications. On the other hand, Stitched is not a desirable design point. With a 2x bigger log than Serial, it only delivers a 3x higher parallelism. StSerial is only slightly better than Serial.

6.3 Replay Execution Time

We now compare the replay execution time of the benchmarks under a variety of scenarios. We start with the MaxPar log with different block sizes. Figure 16 shows the replay execution time for maximum block sizes equal to 1K, 4K, 16K, and 64K instructions. In all cases, there are 2 block clusters and 64K instructions per cluster. Thus, there are 64, 16, 4, and 1 blocks per cluster, respectively. The plot is normalized to the execution time of recording with 64K-instruction blocks (the recording time for all the other scenarios is practically the same). We can see that, in general, replay execution time is comparable to recording time, even for these communication-intensive benchmarks. On average, with 1K blocks, replay takes only 50% longer than recording, while with 4K blocks, it takes only 60% longer. As we increase the block size, replay time increases. This is largely because there is less replay parallelism with big blocks.

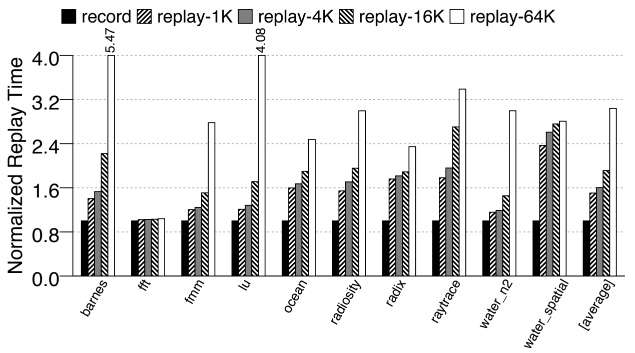


Figure 16. Replay execution time with the MaxPar log for different block sizes.

Figure 17 shows how the logs of the different backends (MaxPar, Stitched, StSerial, and Serial) affect the replay execution time. For this experiment, we use the baseline RRU configuration of Section 5. As usual, the figure shows the replay times normalized to the recording execution time. As expected, the less-parallel logs cause an increase in the replay execution time. On average, we see that with MaxPar and Stitched, it takes about 60% and 100% longer, respectively, to replay than to record. Replaying in StSerial and Serial takes, on average, about 7 and 12 times longer, respectively, than recording. The relative speeds of MaxPar, Stitched, and StSerial largely match the parallelism numbers provided by the NICPLs in Figure 14. Serial, however, is much slower. The reason is that, in the current implementation, Serial does not try to merge blocks before writing them to the log, since this may cause replay deadlocks (we omit the discussion of why this is the case in the interest of space). Hence, it has a high overhead passing tokens, which is done with semaphores.

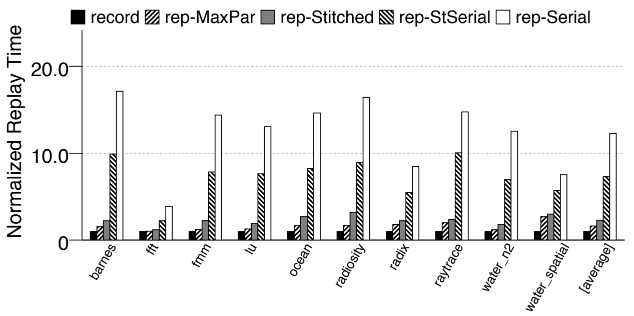


Figure 17. Replay execution time with logs from different backends for a 4K block size.

An important feature of application-level R&R is the ability to replay on machines that have different architectures and, in particular, different processor counts than the recording machine. In Figure 18, we show the effect of using fewer processors to replay than were used to record. The recording run used 8 processors, while the replay executes a MaxPar log on 8, 4, 2 or 1 processors. In each case, the 8 threads of the application have to be multiplexed over the available number of processors and synchronize by passing tokens around. The figure compares the execution times normalized to the recording time. As shown in the figure, the replay becomes progressively slower, but not exceedingly so. The amount of slowdown is a function of the number of replay processors as well as the parallelism that existed in the original execution and was captured in the MaxPar log.

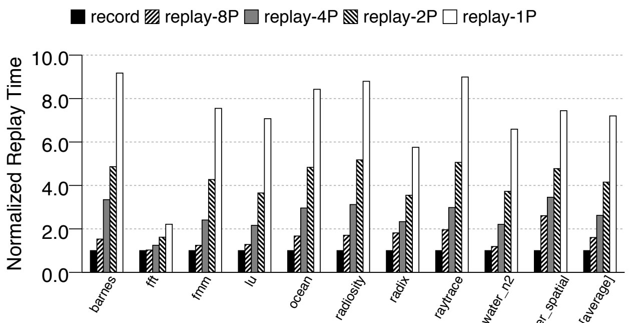


Figure 18. Replay execution time with a lower processor count than during recording.

Finally, Figure 19 breaks down the execution time of replay with the MaxPar log. For each benchmark, the bars are normalized to 100 and broken down into: time spent executing user mode instructions (*user*), time when the OS is executing on behalf of the application, such as servicing system calls (*kernel*), overhead associated with handling the input log (*input log overhead*), overhead associated with handling the memory interleaving block log (*block log overhead*), time spent waiting for tokens from predecessor blocks (*wait for pred*), and time that could not be classified as one of the above (*other*). The latter is mostly application-level load imbalance — e.g., when some application threads are waiting on a barrier while other threads have not yet arrived at that barrier.

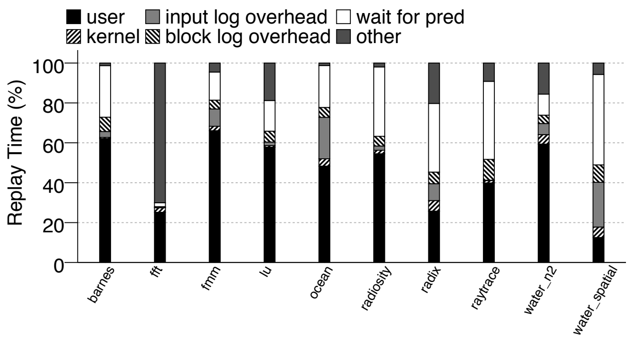


Figure 19. Breakdown of the replay execution time with the MaxPar log for 4K blocks.

The figure shows that the benchmarks exhibit very different behaviors. However, if we focus on the overheads — i.e., the categories other than *user* and *kernel* — we see that *other* and *wait for pred* are often dominant. The *other* category typically implies that there is load imbalance, and little can be done. For example, FFT has a long initialization phase that causes load imbalance. The *wait for pred* category appears in many benchmarks. They suffer considerable slowdowns just because of waiting for tokens. This suggests focusing on recording more parallelism, possibly with more aggressive techniques. Also, utilizing special hardware support (rather than using semaphores and an all-software solution) for token passing during replay may significantly reduce this overhead for some applications.

6.4 Dependence-Tracking Window Analysis

In this section, we compare different organizations of the dependence-tracking window, using replay parallelism (i.e., NICPL) as our metric. Figure 20 shows the NICPLs for different designs, represented as $I \times J \times K$, where I is the number of block clusters (including the last one, with no signatures), J is block size, and K is the number of blocks per block cluster. Recall that the tracked window size equals $I \times J \times K$.

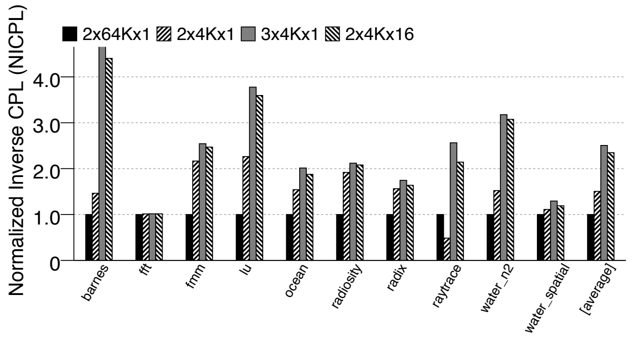


Figure 20. Effect of different organizations of the dependence-tracking window on parallelism. $I \times J \times K$ stands for I block clusters, J block size, and K blocks per cluster.

In the figure, higher bars are better. The figure is normalized to the NICPL of $2 \times 64K \times 1$, which can track a window of 128K. If we reduce the block size to 4K (second bar), we increase the NICPL even though the window is now only 8K. This design is better because small blocks improve parallelism. If we increase the number of clusters to 3 while keeping the block size to 4K (third bar), we improve NICPL significantly because the tracked window increases to 12K, but at the cost of an extra set of signatures. Finally, if we keep the clusters to 2 and the block size to 4K but use 16 blocks per cluster (last bar), we have nearly the same NICPL. We have a less precise tracking with only 2 clusters, but have a larger tracked window size. Moreover, we need no extra signatures. This is the most competitive design and we use it as default.

7. Related Work

Deterministic R&R has been the subject of a vast body of research work. Software-based solutions assume no special hardware support. Instead, they rely on modified runtime libraries, compilers, operating systems and virtual-machine monitors to capture sources of non-determinism. Russinovich and Cogswell [32] propose to modify the OS scheduler to record thread interleaving in a uniprocessor. Recap [27] takes a compiler-based approach, where the compiler inserts some code before every read operation that may touch shared data. LeBlanc and Mellor-Crummey [18] use Reader-Writer locking for shared memory accesses to log an execution. Agora [12] uses write-once memory and a history log for maintaining the correct state of a program. DejaVu [8] records the scheduling decisions of a Java Virtual Machine to enable deterministic replay of multi-threaded Java applications on uniprocessors. Bressoud and Schneider [5] and ReVirt [10] use a modified hypervisor to replay single processor virtual machines. Dunlap et al. [11] extend ReVirt to multiprocessor virtual machines. Flashback [33] focuses on application-level R&R for debugging, and captures only the interactions between the application being debugged and the operating system, like system calls and signals, instead of logging everything. PinPlay [29] is another software-only approach based on the Pin dynamic instrumentation system. It uses Pin for both recording and replay. Scribe [16] uses the virtual-memory page-level access control mechanisms to capture the interleaving of shared memory accesses. Although the paper reports reasonable overheads for system applications with low levels of sharing, it is not clear how it will perform for sharing- and synchronization-intensive programs.

These software-based approaches are either inherently designed for uniprocessor executions or suffer significant slowdown when applied to multiprocessor executions. DoublePlay [34] made efforts to make replay on commodity multiprocessors more efficient. It timeslices multiple threads on one processor and then runs multiple time intervals on separate processors. Hence, it only needs to record the order in which threads in each time interval are timesliced on the corresponding processor. This technique eases logging by only requiring the logger to record the order in which the time slices are executed within a time interval. However, DoublePlay uses an additional execution to create checkpoints off which multiple time intervals can be run in parallel. It also needs to use modified binaries (in particular, a modified libc) for efficient execution.

ODR [2] and PRES [28] are probabilistic replay techniques for reproducing concurrency bugs. The idea is to record only a subset of non-deterministic events required for deterministic replay (to reduce the recording overhead) and use a replayer that searches the space of possible executions to reproduce the same application output or bug, respectively. Respec [20] targets online replay scenarios. It records a subset of non-deterministic events and uses the online replay run to provide external determinism. The idea is to retry the execution from the last checkpoint when a divergence happens. Like DoublePlay [34], it needs to use modified binaries.

Hardware-based solutions usually use hardware to record memory races to reduce the overhead of R&R for multiprocessor executions. Past work on memory race recording has largely concentrated on directory-based schemes (e.g., [3, 13, 23, 24, 36, 37]). There is work on snoopy protocols [26, 30, 31], but the designs require modifications to the cache coherence protocol hardware. Specifically, Strata [26] requires that all the processors agree to start a new stratum (or logging epoch) at regular intervals. This is done by augmenting the messages with a Log Stratum bit, which can be set by the processor initiating the miss or by a processor that provides the data. Strata uses a recording approach that requires that all processors record an entry in their logs at the same time, which does not scale well with the processor count. In the MRR [30] and CoreRacer [31] systems, every time that a block commits, the event is broadcasted with a bus transaction. The design uses Lamport clocks to order the blocks and does not provide much parallelism.

FDR [36] and RTR [37] are among the very first race recording techniques proposed. They record dependences between pairs of instructions and, thus, can record parallel dependence graphs. However, they are full-system techniques and rely on modified directory protocols. Also, recording dependences between pairs of instructions can produce large logs and increase associated overhead. To reduce this overhead, block-based techniques [7, 13, 23, 24, 30, 31, 35] have been proposed, but they are not designed for parallel replay and require changes to the coherence protocol. DeLorean [23] and Capo [24] are block-based techniques that use speculative multithreading hardware to achieve replay parallelism.

Karma [3] is the first block-based R&R technique that explicitly targets replay parallelism without relying on speculative hardware. It is a whole-system (rather than application-level) R&R scheme for directory protocols. It records bidirectional dependences between source and destination blocks and, hence, makes some modifications to the cache coherence messages. The design allows two blocks to be tracked, an idea we build on in this paper. Karma allows the blocks to grow beyond conflicts, similar to the Stitched logs presented in this paper. The paper reports replay speeds within 19%-28% of vanilla runs (i.e., without R&R). The authors make, however, several simplifying assumptions about the mechanisms used for recording non-deterministic input events, and for handling memory logs, which need care in a realistic, OS-aware implementation of R&R. It is also difficult to extrapolate their results to an application-level scheme like Cyrus. It is also unclear how their replay mechanism can be extended to application-only replay, and to cases where the recording and replaying machines have different numbers of processors.

BugNet [25] records user processes by storing the result of load instructions in a hardware-based dictionary. This is enough to handle both input and memory-interleaving non-determinism and allows each thread to be replayed independently. However, BugNet still needs a solution to record inter-thread dependences, for which it uses FDR [36]. Lee et al. [19, 21] augment this technique by using offline symbolic analysis to reconstruct the inter-thread dependences. This technique is mostly suitable for debugging, since the analysis is, in general, a slow process.

8. Concluding Remarks

This paper presented Cyrus, the first hardware-based approach for unintrusive, application-level R&R that explicitly targets high replay speed. It introduces the concept of an on-the-fly software backend pass during recording which, as the log is being generated, transforms it. This backend pass fixes-up the log, which has incomplete information due to our recording requirements of only application-level interactions and no coherence protocol changes.

It also exposes the recorded parallelism and can flexibly trade-off replay parallelism for log size. Cyrus had negligible recording overhead. In addition, for 8-processor runs of SPLASH-2, it attained an average replay parallelism of 5, and a replay speed that was, on average, only about 50% lower than the recording speed. The backend pass was highly flexible and added negligible overhead.

While it is conceivable that a single-threaded backend can become a bottleneck as the number of processors increases, we did not find it to be so in our experiments (as evidenced by Figure 13). Also, when such a backend becomes a bottleneck, we can parallelize it and allocate more than one processor to it. This should only be necessary for considerably large systems and workloads that actively use many processors and generate many blocks. It should be emphasized that compute- and sharing-intensive applications such as SPLASH2 programs represent the worst case for Cyrus. Less demanding *system* workloads, such as databases and web servers, that create smaller memory logs, are considerably easier to handle.

Although we motivated Cyrus using bus-based snoopy systems, it can be easily adapted to other coherence schemes. Cyrus uses source-only recording, which is crucial for parallel application-level R&R. Moreover, it uses the number of bus transactions as the time source. The same time source can be used in any coherence scheme where all the processors see all of the requests in the same order (e.g., address-broadcast tree of Sun's Starfire [6] or the ring-based design of Intel's SandyBridge processors [38]). Alternatively, in many modern CMPs, there are chip-level-consistent clock sources that can be used as the time stamp (e.g., the uncore clock that synchronizes the on-die interconnect of recent Intel systems).

For directory-based designs, where such time sources are not available, the local time stamp of the requesting processor can be piggybacked on the request message. Then, the sourcing processors can save this time stamp in their successor vectors — hence, implementing source-only recording. This requires some changes in the format of the coherence requests. Fortunately, compared to bus-based designs, such changes are relatively easy to make in directory-based systems.

Acknowledgements

This work was supported in part by NSF under grants CCF-1012759, CNS-1116237, CNS-0834738, and CNS-0831212; Intel under the Illinois-Intel Parallelism Center (I2PC) and the ISTC for Secure Computing; and AFOSR MURI grant FA9550-09-01-0539.

References

[1] H. Agrawal et al. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3), May 1991.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Symposium on Operating Systems Principles*, 2009.

[3] A. Basu et al. Karma: Scalable deterministic record-replay. In *Int. Conference on Supercomputing*, 2011.

[4] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7), July 1970.

[5] T. Bressoud and F. Schneider. Hypevisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1), Feb 1996.

[6] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1), Jan. 1998.

[7] Y. Chen et al. LReplay: A pending period based deterministic replay scheme. In *Int. Symposium on Computer Architecture*, 2010.

[8] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Symposium on Parallel and Distributed Tools*, 1998.

[9] B. Cully et al. Remus: High availability via asynchronous virtual machine replication. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[10] G. W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium on Operating Systems Design and Implementation*, 2002.

[11] G. W. Dunlap et al. Execution replay of multiprocessor virtual machines. In *Int. Conference on Virtual Execution Environments*, 2008.

[12] A. Forin. Debugging of heterogeneous parallel systems. In *Workshop on Parallel and Distributed Debugging*, 1988.

[13] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *Int. Symposium on Computer Architecture*, 2008.

[14] S. T. King and P. M. Chen. Backtracking intrusions. In *Symposium on Operating Systems Principles*, 2003.

[15] S. T. King et al. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[16] O. Laadan et al. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS Int. Conference on Measurement and Modeling of Computer Systems*, 2010.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4), Apr. 1987.

[19] D. Lee et al. Offline symbolic analysis for multi-processor execution replay. In *Int. Symposium on Microarchitecture*, 2009.

[20] D. Lee et al. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[21] D. Lee et al. Offline symbolic analysis to infer Total Store Order. In *Int. Symposium on High Performance Computer Architecture*, 2011.

[22] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), February 2002.

[23] P. Montesinos et al. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Int. Symposium on Computer Architecture*, 2008.

[24] P. Montesinos et al. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[25] S. Narayanasamy et al. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Int. Symposium on Computer Architecture*, 2005.

[26] S. Narayanasamy et al. Recording shared memory dependencies using strata. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[27] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Workshop on Parallel and Distributed Debugging*, 1988.

[28] S. Park et al. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symposium on Operating Systems Principles*, 2009.

[29] H. Patil et al. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Int. Symposium on Code Generation and Optimization*, 2010.

[30] G. Pokam et al. Architecting a chunk-based memory race recorder in modern CMPs. In *Int. Symposium on Microarchitecture*, 2009.

[31] G. Pokam et al. CoreRacer: A practical memory race recorder for multicore x86 TSO processors. In *Int. Symposium on Microarchitecture*, 2011.

[32] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Programming Language Design and Implementation*, 1996.

[33] S. M. Srinivasan et al. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, 2004.

[34] K. Veeraraghavan et al. DoublePlay: Parallelizing sequential logging and replay. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[35] G. Voskuilen et al. Timetraveler: Exploiting acyclic races for optimizing memory race recording. In *Int. Symposium on Computer Architecture*, 2010.

[36] M. Xu et al. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Int. Symposium on Computer Architecture*, 2003.

[37] M. Xu et al. A regulated transitive reduction (RTR) for longer memory race recording. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[38] M. Yuffe et al. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Int. Solid-State Circuits Conference*, 2011.