Hardware-Assisted Replay of Multiprocessor Programs

David F. Bacon, University of California, Berkeley and IBM Watson Research Center
Seth Copen Goldstein, University of California, Berkeley

Abstract

Shared-memory parallel programs can be highly non-deterministic due to the unpredictable order in which shared references are satisfied. However, deterministic execution is extremely important for debugging and can also be used for fault-tolerance and other replay-based algorithms. We present a hardware/software design that allows the order of memory references in a parallel program to be logged efficiently by recording a subset of the cache traffic between memory and the CPU's. This log can then be used along with hardware and software control to replay execution.

Simulation of several parallel programs shows that our device records no more than 1.17 MB/second for an application exhibiting fine-grained sharing behavior on a 16-way multiprocessor consisting of 12 MIP CPU's. In addition, no probe effect or performance degradation is introduced. This represents several orders of magnitude improvement in both performance and log size over purely software-based methods proposed previously.

1 Introduction

Debugging parallel programs can be very difficult. Unlike sequential programs, truly parallel programs are highly non-deterministic due to interleaved access to shared memory. As a result, a bug may not reappear when the program is run with the same inputs, and may disappear when timings are changed due to insertion of monitoring code. A summary of difficulties and potential techniques is presented in [11].

One of the most fundamental techniques for debugging sequential programs is the ability to re-run the program with the same inputs and have it fail in the same way. This paper describes a mechanism for providing this ability on a shared-memory multiprocessor, even for programs which exhibit very fine-grained sharing behavior.

We present a design for a combined hardware/software subsystem that allows fine-grained parallel executions to be replayed without introducing any probe effect and without recording an impossibly large amount of data. This is done without requiring modifications to the source code.

Since our methods build considerably on previous work, we begin by surveying replay techniques for sequential and multiprocessor architectures. Section 3 describes the class of parallel architectures we are targeting. Section 4 discusses our methods for recording event histories, and Section 5 describes the hardware that is used to realize these methods.

Section 6 describes experimental data from several simulations, which is used to evaluate the size required by our trace data, the speed required by our hardware data collection device, and a number of other parameters. Finally we conclude with a summary of the strengths and weaknesses of our approach.

Space constraints have required us to omit some details, particularly of parallel replay with the partial order logger and the hardware design; the interested reader is referred to [1] for more details.

Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 1991. SIGPLAN Notices Vol. 26, No. 12, December 1991, pp. 194-206. Extended version available as technical report UCB/CS 91/624, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

2 Previous Work

Debugging by replaying program execution has been of interest to the research community for over 20 years [2]. However, the cost of recording the necessary trace data has significantly hindered this approach [19].

Sequential programs are generally deterministic for long periods of time; non-determinism occurs infrequently due to input data, results of system calls, and interrupts. The first two can be logged by simply recording the input or result.

Interrupts are asynchronous and therefore to record precisely when they occurred requires knowing precisely which instruction they followed. This has been solved with a device known as an *instruction counter*, which is decremented by one for every instruction executed. When an interrupt occurs, the instruction counter can be saved, giving the precise point in the program execution stream (in terms of the number of instructions executed). When the instruction counter reaches zero, a trap is generated. To re-execute the program deterministically, the instruction counter is loaded with the number of instructions that preceded the interrupt; then it will reach zero and trap at precisely that point, and the effects of the original interrupt handler can be simulated.

Hardware instruction counters have been implemented experimentally [3] as well as in the HP Precision RISC architecture [7]. They can also be simulated in software by counting the number of backward branches and combining this count with the value of the program counter [12].

Instruction counters can also be used to make multithreaded uniprocessor applications repeatable, since the contents of the instruction counter can be recorded every time a scheduling operation is performed.

2.1 Replaying Parallel Programs

In multiprocessors, shared memory interleaving is truly concurrent, and a more complex scheme is required. A number of software-based schemes have been proposed or implemented. The Recap system design[14] proposes modifications to the compiler that instrument the code. Any reference that might be to a shared location is followed by a short instruction sequence that copies the value read into a log and increments the log pointer. Since there is a separate log for each process, this scheme has the advantage of allowing arbitrary subsets of the processes to be recorded and replayed, and parallel replay is easy. On the other hand, since *every* read from a potentially shared location must be recorded, the logs can become very large — the authors estimate their scheme would consume 1 MB per second on a 1 MIP VAX 11/780. This means that 1 MB per second must be transferred from each processor cache into the main memory log area, which will cause significant performance degradation as well as probe effect. Another source of both performance loss and probe effect is the extra instructions that must be executed for each read of potentially shared data.

An alternative approach is to record version numbers of objects rather than their values, and to ensure on replay that the correct version is read. This is the approach taken in the Agora distributed debugger [6]. Version number logging is also used in Instant Replay for the BBN Butterfly [9]. Instant Replay requires that all shared data objects be accessed through procedures which lock the object, update its sharing state, and write an entry into the per-process log. This works well for programs with medium-grain interactions mediated through high-level objects like monitors and message queues which already incur the performance overhead of procedure call and locking. However, fine-grained use of shared memory causes performance degradation and requires changes to the code which "bracket" shared memory access with calls to Instant Replay's logging operations. Thus to have debuggable code, "the programmer can balance the reduction of parallelism incurred when locking for long periods of time with the overhead of frequently executing the locking primitives."

In summary, all software-based methods proposed suffer from at least two of the following defects: significant amounts of probe effect, a significant amount of time and space overhead, or an assumption that sharing is coarse grain and occurs infrequently. Programmer modifications of the code may also be required.

The support for fine-grained sharing is one of the primary advantages of shared-memory multiprocessors, and probe effect can significantly complicate debugging. We therefore pursue a hardware-based solution which eliminates probe effect and allows the processors to access the shared memory at the full speed supported by the bus and memory subsystem.

Op	CPU-Cache Interaction	Bus Transaction	Op	CPU-Cache Interaction	Bus Transaction
r	read hit	none	w	write hit to private line	none
R	read miss	READ	W	write miss	READ-MODIFY
			Wi	write hit to shared line	INVALIDATE
Wr	capacity write-back	WRITE-REPLACE	Wu	coherency write-back	WRITE-UPDATE
P	capacity purge	none	I	invalidation	none

Figure 1: Notation for Cache Operations. The *Op* column indicates the abbreviation used for the operation; the corresponding CPU-Cache interaction and resulting bus transaction are shown. The first group of operations are those which occur in the local cache in direct reponse to a read or write to the designated line by the CPU. The second group are indirect operations are performed by the local CPU when freeing a line (P and Wr), or in response to bus events generated by another CPU (Wu and I).

3 Machine Model

For the bulk of this paper, we will assume that the machine architecture in question is a shared-memory bus-based 16-way multiprocessor. Each processor has a 128KB write-back cache with a 4 or 16 byte line size. The coherency protocol maintains sequential consistency by snooping and invalidation. Our results are applicable to this general class of machines; variations of several binary orders of magnitude in cache size, line size, and the number of processors are possible, and we discuss the relevance of these parameters where appropriate.

Processors are assumed to be RISC design, with most instructions referencing registers rather than memory. The bus is assumed to be pipelined, with read requests placed on the bus on one cycle and results read three cycles later. For instance, a 10 MHz bus with a eight byte wide data-path and a 16 byte cache line is assumed to be able to sustain 3.3 million transactions second. It is also assumed that the identity of the processor making each bus request is available.

Our machine model is largely based on the Sequent Symmetry, with the exception of our assumption of RISC processors [10]. We chose the Symmetry because its pipelined bus and relatively large number of processors for a snoopy cache machine would push the limits of our techniques. We also consider two other bus/cache architectures: the DEC SRC Firefly [17] and the IEEE Futurebus [18]. When differences between our standard model and these architectures require different solutions from those proposed, we will comment on the modifications required to support them. By considering three different bus architectures, we demonstrate the robustness of our scheme.

The Firefly is generally less demanding because it has fewer processors, making bus performance less critical. This is reflected in the Firefly's use of write-through rather than write-back, and its lack of bus pipelining, yielding a bandwidth that is roughly one third of the Symmetry's.

We assume a write-back/write-invalidate protocol: dirty lines are not written through to memory until required by cache capacity or coherency, and when a cache writes a line, all other processors invalidate it in their cache. A *shared* bus signal is asserted when a processor reads a line that is present in another processor's cache.

The following list describes the bus transaction types and their characteristics. In parentheses it is noted whether the transaction requires an address (one cycle), data (two or four cycles), or both. In square brackets the notation used for the cache operations corresponding to the bus transactions is shown. This information is summarized in Figure 1.

- READ [R]: The processor reads a location which is not cached. An address is presented on the bus; three cycles later the cache line words appear consecutively on the bus. (address and data)

  If the line is in another cache and is unmodified, the *shared* line is asserted and the line is marked non-exclusive in all caches, and all cached copies are marked shared. If the line is in another cache and is modified, the owning cache performs a WRITE-UPDATE [Wu].

  If a line must be replaced to read this line, it is written back if dirty [Wr] or purged if clean [P].

- READ-MODIFY [W]: This is equivalent to a read followed by an invalidate, but a bus cycle is saved since the address is only presented once. This corresponds to a write miss, since the cache line must be read before an individual word can be modified. (address and data)

If a line must be replaced to read this line, it is written back if dirty [Wr] or purged if clean [P].

- INVALIDATE [Wi]: If a processor wishes to write a line that is non-exclusive, it puts the address on the bus, and all other caches invalidate the line. (address only)

  The invalidation in the other caches is labeled I.

- WRITE-REPLACE [Wr]: If a dirty cache line is flushed, it must be written back to main memory. An address is presented, followed by the successive words of the cache line. (address and data)

- WRITE-UPDATE [Wu]: If another CPU reads a line which is dirty in another cache, the read from memory is preempted and the modified data is placed on the bus by its owner (indicated by Wu in the owner's cache stream). The data is read by the requesting cache controller and written back to main memory, and the line is marked non-exclusive if the original transaction was a READ, or the line is flushed in the writer and marked exclusive in the reader if the original transaction was a READ-MODIFY. (consumes no additional cycles)

We often make use of diagrams showing the sequence of operations in the local caches of the CPU's. These diagrams may or may not show the purely local events (r, w, P, and I). The operations will generally be subscripted with the cache line address. WRITE-UPDATE and WRITE-REPLACE transactions are collectively referred to as *write-back* transactions [B].

4 Event Tracing Methods

In this section we describe two alternative methods for making parallel executions replayable, without going into the details of how these methods would be implemented in hardware. Section 5 describes their implementation in detail.

Since we are assuming a RISC CPU design, each data reference corresponds to exactly one instruction, and each instruction to at most one data reference. Instruction fetches are ignored because all code is assumed to be read-only, so instruction fetches can not cause any non-determinism. This means that every memory reference can be uniquely identified by the instruction counter of the issuing instruction and the processor number of the issuing processor. In our diagrams we will show multi-processor executions as streams of cache operations, with each operation identified by a local instruction counter. An *event* in the execution of a processor occurs at a particular instruction, identified by its instruction counter value.

In a parallel program, events within a process are totally ordered, and processes are partially ordered with respect to each other. In a shared memory multiprocessor, it is reads and writes to shared memory which create this partial order. The point at which a reader process reads a data value follows the point at which the writing process wrote the value; similarly, writes follow the reads which they destroy. The partial order is called the *happens before* relation, denoted by the "$\rightarrow$" symbol.

In Figure 2, $B_{33} \rightarrow A_4$, $C_{98} \rightarrow A_8$, and (due to transitivity and total ordering within a processor) $C_{95} \rightarrow A_{10}$. However, A_5 and B_{35} are concurrent, since neither happens before the other, but they must both precede A_8. For more details on partial orders and their relation to debugging see [8, 5, 15].

Our initial approach was to provide version logging (as in Instant Replay or Agora) in the hardware, but this would have required extensive modifications to the cache, considerable increase in bus traffic to transmit version numbers, and a directory that could accommodate a version number for each (potentially) shared line. So although version logging has many attractive features, we abandoned it because of these defects.

The alternative to version logging is *event logging*, in which the relative order of non-deterministic events is logged and forced to repeat itself on replay. Since the events occur in the same order, the values generated by the execution will be the same. All of our schemes are variations on this theme. Note that we will only deal with non-deterministic events caused by shared-memory parallelism; we assume that individual processes are deterministic. This can be achieved in an orthogonal fashion through the use of instruction counters or other techniques as described in Section 2.

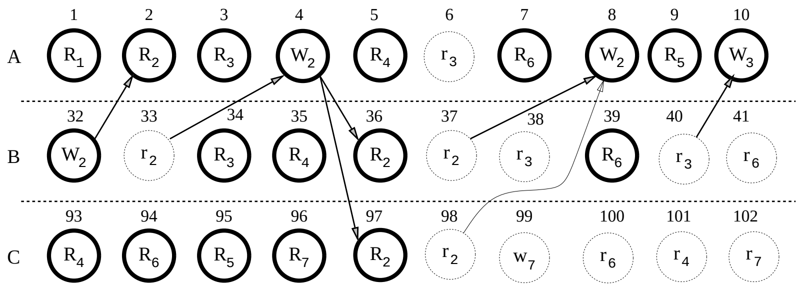


Figure 2: A Happens Before relation induced by memory operations of three processors. The dark circles represent operations that result in bus transactions. The number above the circle is the instruction counter value of the issuing instruction.

4.1 Cache Traffic as an Event Stream

The cache reduces the number of bus transactions by several orders of magnitude. Our approach is to log a subset of the cache-to-memory transactions. On replay we guarantee that these events occur in an order indistinguishable from the original execution. Since we have only logged cache-to-memory traffic, we must take the references that were satisfied by the cache into account to demonstrate that our replay algorithm works correctly.

In Figure 2 we showed all memory accesses by the processors. Due to caching only some of these will be broadcast on the bus, but any memory access which might create an inter-processor dependency in the happens before relation *will* be broadcast, because a write of a shared cache line will cause an INVALIDATE transaction to be broadcast, and a read of a previously written line will miss in the cache because the line has been previously invalidated or flushed, and will therefore generate a READ transaction. Additional bus transactions may be generated which are not relevant to the partial order between the processors; we will describe the effects of these transactions shortly.

A *partial order* exactly captures the required synchronizations between processors as they access memory. On the other hand, recording the partial order requires recording the order of accesses to *each line*, and can be very expensive to collect and process. We present two different schemes for simplifying the logging procedure. The first converts the partial order into a total order. This requires the processors to execute serially on replay, but has the advantage of requiring a minimum of hardware. The second scheme records a subset of the partial order and allows significant parallelism on replay.

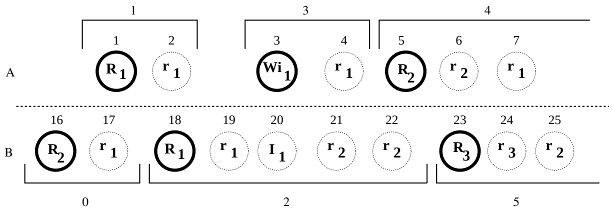
In either case, due to the non-deterministic interaction between caches, it is necessary to record the point at which the cache operations occur. This is done by recording the instruction counter, which precisely locates each memory reference. In the total order, the memory addresses can be ignored. In the partial order, addresses must be considered as well, because the partitioning of the address space into lines is being used to allow potential parallelism of memory access.

4.2 Recording a Total Order

If all bus transactions are logged by recording the CPU number and the instruction counter in a sequential buffer, a total order is placed on the bus transactions in the original execution. In order to guarantee correctness the memory accesses that occur *between* bus transactions must be replayed in the same (or an equivalent) order. First, changing the order of the operations between bus transactions could cause different coherency traffic; second, replacements and misses may occur in different places on replay, and these operations must be satisfied without affecting the correctness of replay.

Because we have a total order on the bus transactions, we will replay the CPUs sequentially. This will result in a slow-down by as much as 16, but for debugging and recovery, this is acceptable if replay starts from a relatively recent checkpoint. The correct order for sequential replay is to allow the first processor in the log to run up to the instruction *preceding* the instruction that caused that processor's next bus transaction.

This is shown in Figure 3: on replay, segment 2 must run to completion before the write to location 1 that begins segment 3 is executed; otherwise, the read of location 1 could get the wrong value. However, because location



(a) Concurrent execution of bus events.

B:16, A:1, B:18, A:3, A:5, B:23, ...

(b) Order in which events arrive at the logger.

B:17, A:2, B:22, A:4, A:∞, B:∞

(c) Information written to the log.

B:0-17, A:0-2, B:18-22, A:3-4, A:5-∞, B:23-∞

(d) Resulting sequential replay schedule.

Figure 3: A parallel execution and its replay schedule. The replay schedule of (d) is also shown by the segments labeled 0 through 5 in (a).

1 is invalidated in processor B, we know that there can not have been any references to location 1 before the next logged event of processor B. Therefore it is safe to run the rest of segment 2 before beginning segment 3.

Note that we always run the CPUs in the order of their log events up until one instruction before the next logged event for that CPU. The exception comes at the beginning, where processors run from instruction 0 up to the end of the first interval, since they are guaranteed to be deterministic prior to their first shared memory access.

The first log entry, *B:17*, is not actually completed until the second bus transaction is initiated by B. It is only at this point that we know that B must be run up to instruction 17, one instruction before the one that caused the next bus transaction. This "look-ahead" makes it necessary to wait for the next transaction from a CPU before completing that CPU's current log entry. An alternative would be to simply record the event stream of Figure 3(b), but this would require post-processing of the log, which we would like to avoid.

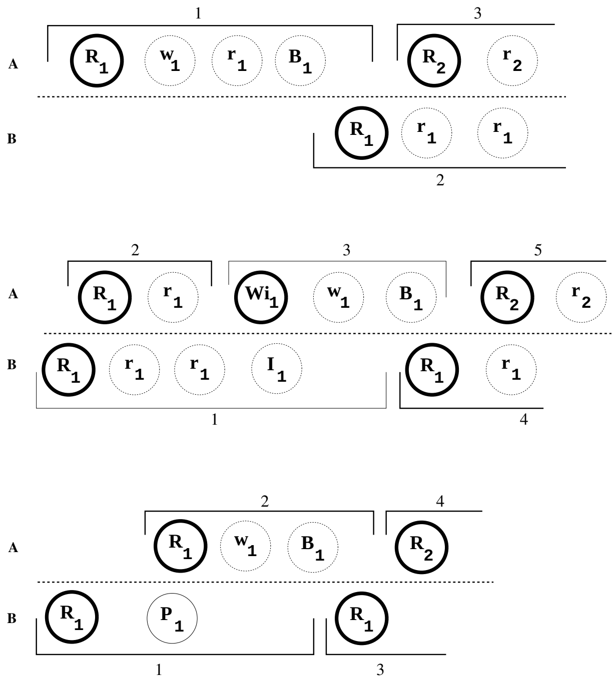


Figure 4: Three Memory Reference Streams Involving a Write-Back

4.2.1 Removing Transaction Runs

Note that when the same CPU issues two successive bus transactions, on replay it will be allowed to execute up to (but not including) the instruction generating the first transaction, and immediately afterward up to the instruction generating the second transaction. This leads to the first optimization: when two successive transactions are from the same CPU, the first one can be discarded.

4.2.2 Removing Writeback Transactions

Since WRITE-UPDATE's are fully overlapped with READ or READ-MODIFY transactions, they are a potential problem for the logger because they do not consume any bus cycles. If both the WRITE-UPDATE and the READ transaction must be logged, this would require the logging device to be capable of twice the peak speed that would be required in the absence of WRITE-UPDATE.

All write-backs (WRITE-UPDATE and WRITE-REPLACE transactions) can be removed from the log by the following reasoning: since the line was dirty, it was not in any other cache. Therefore, any reference to the line by

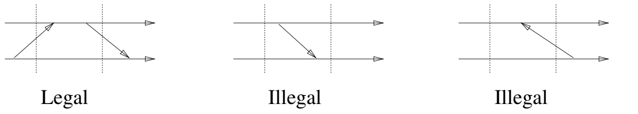


Figure 5: Legal and Illegal Dependencies Between Slices

a processor that follows the write in the original execution will also follow it on replay, since the reference must be logged and will occur in the log after the write-back transaction (see example 1 in Figure 4). Any reference to the line that precedes the write-back in the original execution must either precede an INVALIDATE that precedes the write-back (see example 2), or it must itself have been logged and then flushed before the writer referenced it (example 3).

4.3 Recording the Partial Order

In the total order scheme we do not examine the addresses that are being referenced — in essence we are treating the memory as though it were only a single location, and any read that follows a write during execution *must* follow it on replay. However, this is not really true: if we look at memory at a finer granularity, namely the cache line size, much of this "false sharing" effect is eliminated: Wi_1 by processor A and R_2 by processor B can occur in any order, since they reference different memory locations.

Dynamically scheduling the processors from the partial order graph induced by the cache traffic would yield maximum parallelism, but the overhead would also be extremely high. We have adopted a scheme that sacrifices some potential parallelism on replay in favor of a more compact log and much lower replay overhead. We break the partial order into "slices" which are run in parallel, consisting of a certain number of instructions for each processor. As long as these slices obey the constraints of the partial order, the result of running the slices consecutively will be the same as the original execution. Essentially, instead of serializing the entire execution as in the total order scheme, we insert a barrier synchronization between each slice.

Figure 5 shows the characteristics that must be maintained by the slices: a processor can depend upon an event in a previous slice, but it may not depend upon an event in a future slice or an event in the same slice (except when the event is local to the processor in question).

5 Detailed Design

Both the partial order and the total order log are created by recording cache to memory transactions. In order to record the necessary events a logging device is attached to the memory bus. The logging device performs the logging function by interacting with the operating system and by listening to the memory bus.

Basically, both loggers listen to the bus and record the bus transaction type, the processor that generated the transaction and its current instruction count. In addition, the partial order logger records the cache line that was affected by the transaction. The total order logger has four main components: The *page status table*, the *instruction count table*, the *log buffer*, and the *logging disk*. In addition to the four components for the total order logger, the partial order logger also has an *event buffer* and a *scheduler queue*.

5.1 Instruction Count Information

Since the logger records the current instruction counter (IC) of the CPU generating the event, some provision must be made for transferring the IC from the CPU to the logger via the bus. Instead of transmitting a complete instruction count across the bus which would require at least 32 bits per transaction, each CPU keeps track of the number of instructions executed since the previous cache transaction, and only transmits the difference (the *IC-delta*). The width of the IC-delta is 12 bits. In the extremely rare event that a CPU does not cause a bus transaction to occur in 4096 instructions, it will generate an INVALIDATE to a line in a page owned by the logger. This *delta overflow* pseudo-transaction causes the IC-delta to be transmitted over the bus.

5.2 Non-Shared Transactions

The logger ignores transactions to memory that are guaranteed to be non-shared (e.g., instruction fetches). This is accomplished by comparing every address on the bus to the *page status table*. The page status table has one bit for every page in the physical address space. If the bit is 1, the page is sharable, otherwise it is non-sharable and references falling on the page are not logged. The page status table is managed by the operating system as it does virtual memory management.

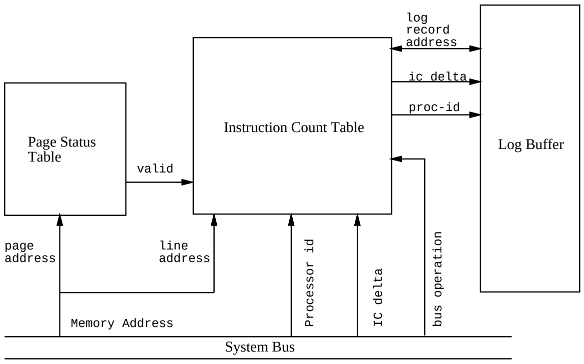


Figure 6: Total Order Logger

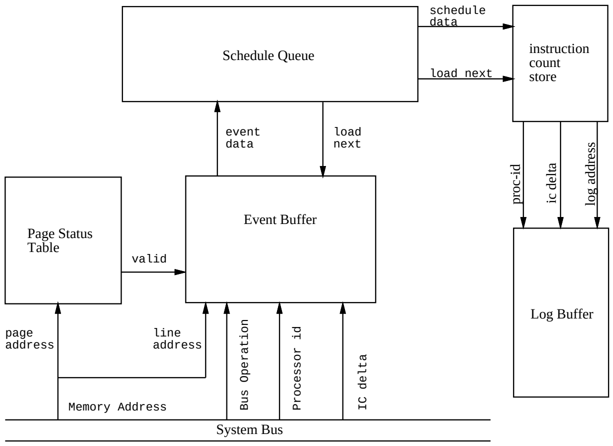


Figure 7: The Partial Order Logger

5.3 Storing the Log

A log consists of a series of log records. Each record contains a CPU number (4 bits) and an IC-delta (12 bits) which on replay contains the number of instructions that the CPU can execute until it must suspend and wait for the logger to start it again.

On replay a CPU executes the number of instructions specified in the *next* event. Thus, the logger uses the current IC-delta to complete the last record for the CPU that caused the current event. This is accomplished by the *instruction count table*, which has one entry per processor. Each entry contains a log record number which is a pointer to the last event for the given processor.

When a loggable event occurs the instruction count table completes the previous record and creates a new log record for the current event and its address overwrites the old log record number in the instruction count table for the processor that caused the event.

In parallel with filling up the RAM log buffer the logger moves the completed records to a disk on the logging device. If the log buffer approaches capacity (which means that there is an incomplete record in the log buffer which is blocking the movement of the records from the buffer to the disk) the logger will perform a *delta request* pseudo-transaction to the processor to get an IC-delta for the incomplete log record (this is done by sending an INVALIDATE for a line on a page owned by the logger). The processor will perform a delta overflow pseudo-transaction, which will reset its instruction counter and put its current IC-delta on the bus.

5.4 The Total Order Logger

The simplest way of ensuring that playback will be deterministic is to gather a total order log. This log contains the schedule for replaying the processors one at a time in the order determined by the cache transactions. The log is created by the total order logger (see Figure 6).

For each event the logger, in parallel, determines if the event type and address make the event "loggable." The logged transactions are READ, READ-MODIFY, and INVALIDATE. If the event should be logged it passes to the instruction count table which processes the event as described above.

5.5 Total Order Playback

Once a total order schedule has been created, the logger can be used to playback the execution that created the schedule. For this the log buffer is loaded with the logged events. Each log event contains a processor number and an IC-delta. The logger starts the processor indicated by the processor number of the current log record and loads the processor's instruction counter with the IC-delta of the current log record.

The processor will execute IC-delta number of instructions and then perform a delta overflow transaction while putting itself into a hold state. The logger will then retrieve the next log record and continue playback.

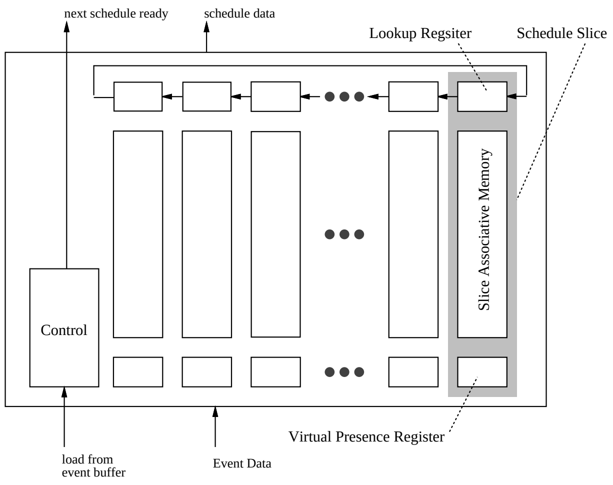


Figure 8: Block Diagram of the Scheduler

5.6 The Partial Order Logger

While the total order logger requires minimal hardware, it restricts playback to running only one processor at a time. If, on the other hand, more analysis is done and the partial order implied by the events is recorded, a significant speedup may be possible. The partial order logger performs such analysis with a surprisingly small amount of hardware.

The partial order logger (see Figure 7) determines whether events are loggable as does the total order logger. Once a transaction is considered loggable, the processor number, IC-delta, address, and transaction type are put into the *event buffer*.

The events are pulled out of the event buffer by the *scheduler*. The scheduler processes the events and puts schedule *slices* into the log buffer. Log buffer overflows are handled in the same manner as for the total order logger.

Each slice is a set of processor number and IC-delta pairs. The entire set of processors in the slice may be played back at the same time. The only difference between the total order logger and the partial order logger for playback is that the partial order logger will start each processor in the slice at once and will not execute another slice until they have all reached the IC-delta specified.

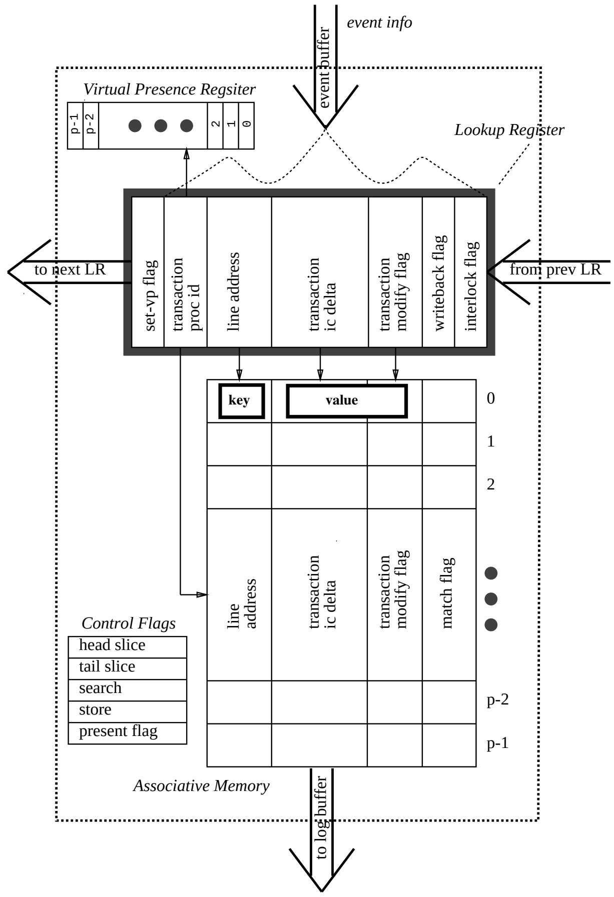


Figure 9: A logical view of a slice of the Scheduler

5.7 The Partial Order Scheduler

The heart of the partial order logger is the scheduler. It is the scheduler's task to translate the events coming from the bus into a series of slices. The scheduler (see Figure 8) is composed of a circular queue of *slices* controlled in a SIMD fashion. As events come from the event buffer they are pushed onto the tail of the queue. The scheduler will then percolate the events toward the head of the queue. When the head of the queue is "full" it will consist of all the events that can be scheduled in one slice.

Each slice (see Figure 9) in the scheduler is composed of four main components: the lookup register, virtual presence register, control flags, and associative memory. The *lookup registers* are connected in a circular queue. At any time one lookup register is designated as the head and another as the tail of the queue. The *virtual presence register* (VPR) is a p-bit wide bit addressable register, where p is the number of processors in the sys-

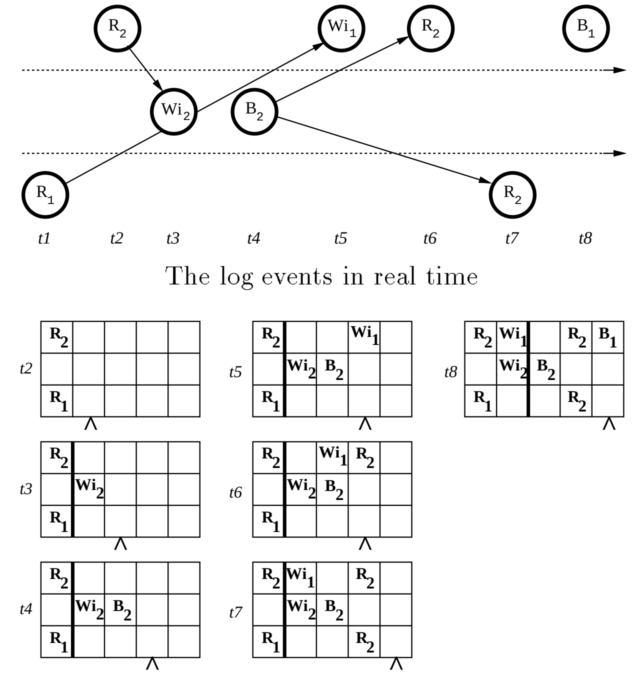


Figure 10: The Scheduler state at the end of every time step. Slices that have been written to the instruction count table are to the left of the the thick line. The tail-slice is indicated by a caret.

tem. The *control flags* are used to control the actions of an individual slice. The *associative memory* holds the actual event information once it has been stored in a given slice.

Each event enters the scheduler by being inserted into the lookup register which is the current tail of the queue. The event is then checked to see if it belongs in the tail's queue. If it doesn't, it is shifted to the left (towards the head of the queue) and the next event can be loaded into the tail slice.

Basically, a read event can be moved toward the head of the queue *iff* no other event in the slice is a write to the same address. A write event can be moved toward the head of the queue *iff* no other event in the slice is either a read or a write to the same address. Writebacks are entered into the scheduler to keep it consistent, but are not entered into the log.

The slices are controlled by a microprogram which in conjunction with the control flags determines what action each slice will take during one of the six microcycles. Using standard technology about 16-32 slices can be put on a chip making a 50ns microcycle time realistic. Even at peak bus usage, transactions will only arrive at 300 to 400ns intervals [17, 10, 16].

The example in Figure 10 shows the state of the scheduler at the end of every real time step for the partial order indicated. The events in the partial order are all bus transactions. Notice that when Wi_2 is stored in the second slice, it sends a virtual presence bit to the left. When it reaches the head-slice, the leftmost slice, the head-slice will write out the schedule and move one slice to the right. At time step 5, Wi_1 enters the scheduler. At time step 6, both Wi_1 and R_2 are searching for the correct slice to occupy. Notice that Wi_1 moves to the left of B_2 even though it followed B_2 in real-time. This is because B_2 and Wi_1 are not dependent on each other.

5.8 Bus transactions for logging

In order to capture the required information the logger must be able to determine which processor is initiating each bus transaction and what kind of transaction is taking place. For split-transaction busses (such as the Futurebus [18]) the processor number is obtainable. For pipelined buses (such as the Symmetry [10]) there is extra bandwidth on the unused address pins. For systems that hold the bus during the entire memory transaction (such as the Firefly [17]), there is enough extra bandwidth during the idle cycles to put the processor number and the IC-delta on the bus.

5.9 Bus transactions for playback

The *delta overflow* pseudo-transaction is used to do playback efficiently. When a processor's instruction counter reaches zero it interrupts the processer and it writes to a special memory location on the logger. This signals the logger that the processor is ready for its next schedule. The processor then reads the new IC-delta from a location on the logger.

6 Simulation

In order to determine the expected size of the log and various other parameters (such as the number of slices in the partial-order logger), we used a simulation of 16 MIPS Co. R2000 CPUs running at 12 million instructions per second on a shared memory bus to generate

Program Name	Line Size	CPU References	Bus Transactions	Write Backs	Compulsory Misses	Log Entries
Verify	4	100,000,000	1,150,000	243,000	326,000	586,000
Verify	16	100,000,000	1,600,000	530,000	200,000	900,000
Genie	4	44,000,000	310,000	91,000	40,000	171,000
Genie	16	44,000,000	300,000	93,000	26,000	182,000

Figure 11: Simulation Results. Each application was run with 4 and 16 byte cache line sizes. Listed are rates per second of total CPU-to-cache references, bus transactions, write-backs, compulsory misses on non-shared data (local variables and read-only segments), and the resulting number of 2-byte log entries. Note that the last three columns approximately equal the number of bus transactions.

traces [13]. This simulator generates every memory reference from the 16 machines for each instruction fetch, data reference, barrier, and lock.

The multiprocessor trace was then fed into a cache simulator which managed coherency for an infinite sized cache. Reads and writes are handled in the obvious way. Locks and unlocks each generated only one write each which reduces the amount of coherency traffic for those applications that used locks. However, the number of processes queueing on each lock was small and thus the distortion was small. Barriers were handled by producing a write for each barrier and a read for all processes currently waiting on the barrier.

We ran simulations for two different workloads: VERIFY, a logic verifier that used locks and fine-grained parallelism; and GENIE, a PLA layout program that used barriers and coarse-grained parallelism. By running these simulations with 4 and 16 byte cache lines we were able to generalize our analysis by comparing the collected statistics with the literature (in particular [4]).

6.1 Size of Logs

Obviously the most crucial factor for the success of our system is the size of the generated logs. The pipelined Symmetry bus operates at 10 MHz, or about 3.3 M bus transactions per second. Since our log entries are two bytes, this means that at 100% bus utilization we would be logging 6.6 MB/second. This is well within the capacity of current memory and disk technology.

However, real bus utilization is substantially lower. To evaluate this we ran simulations that modeled the effect of the logger, and kept statistics on the number of logged bus transactions, the number of shared bus transactions, and the number of read-only and private bus transactions. Measured total miss ratios were between 0.5 and 2%, with shared references making up 8 to 14% of all CPU-cache references. 5% of all shared references generate cache misses that must be logged, so that about 0.5% of all references are logged.

The results of our simulations are summarized in Figure 11. As expected, the fine-grained application (VERIFY) required substantially more log entries than the coarse-grained application (GENIE). Increasing the cache line size leads to false sharing causing an increase both in writebacks and in total log entries; this effect is primarily seen in VERIFY.

Were it not for our idealized memory system, the rate of bus transactions would be further reduced by contention; thus these rates are an upper bound for these applications. Even with these pessimistic assumptions, the worst case measured was 900K log events per second, or 1.8 MB per second with 2-byte log entries.

For a Symmetry, a 64 MB RAM buffer would hold between 0.5 and 3 minutes of log activity (for VERIFY and GENIE, respectively); with a 500 MB dedicated disk this would be 3.5 to 24 minutes. This is sufficiently long to allow periodic checkpointing of the application to take place, as is already done by many long-running compute-intensive tasks to guard against machine failures.

The above figures reflect a substantial reduction in log size due to elimination of writeback transactions: about half of all shared-memory bus transactions cause a write-back to occur, indicating that on average shared objects are read and modified by one CPU, and then another, rather than being read by a large number of CPUs before being written.

Note that a further reduction in total order log size can be obtained by the elimination of successive operations

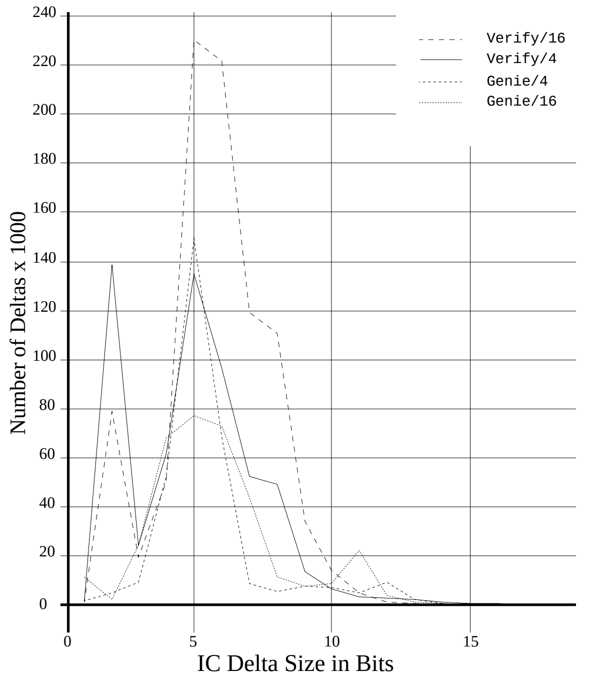


Figure 12: Distribution of Instruction Counter Delta Sizes

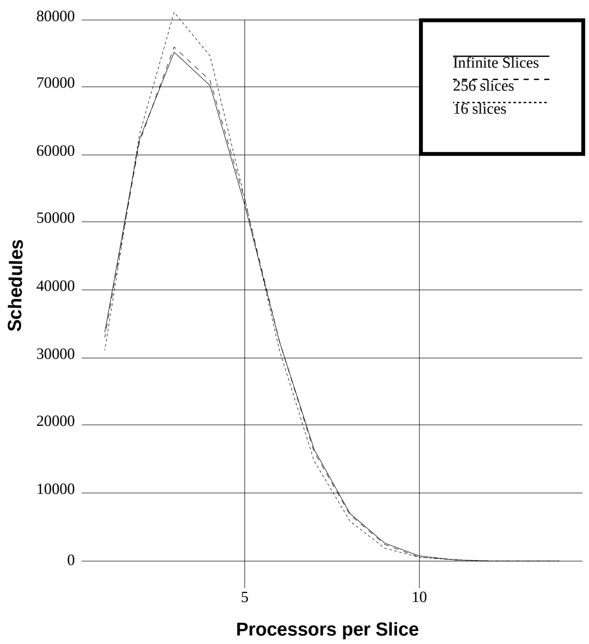


Figure 13: Effect of the number of slices on parallelism available for replay.

by the same CPU: measurements show that approximately 12% of all log events are eliminated.

6.2 Size of Instruction Count Deltas

The delta in the successive instruction counts sent to the logger by each processor were measured to determine how many bits would be required in the log and on the bus. Figure 12 shows the results: there were no deltas of more than 16 bits, meaning that processors executed no more than 65536 instructions between bus transactions.

6.3 Effect of the Number of Slices

For the partial order logger, the largest hardware expense is the associative memory and attached logic that implements the slices. To evaluate the number of slices needed, we simulated the operation of the partial order logger for 16, 256, and an infinite number of slices. With a finite number of slices, when the queue overflows the earliest slice will be written to the log even though more events could be inserted into it, resulting in a loss of parallelism on replay. By comparing the amount of parallelism achieved with infinite slices to that achieved with finite slices, we can decide how many slices are worth their cost in hardware.

Figure 13 shows that the difference between 16 slices and infinite slices is extremely small. Regardless of the number of slices, schedules with three processors dominate. The extra parallelism achieved by infinite slices with large numbers of processors is clearly irrelevant compared to the average schedules achieved.

Thus 16 or even fewer slices will serve very well, allowing the entire partial order logging logic to be embedded on a single low to moderate density chip.

7 Conclusions

We have presented a hardware-assisted scheme for deterministic playback of multiprocessor executions. Unlike software-based schemes proposed or implemented to date, our scheme is virtually free of probe effect: logging is done in hardware using extra bandwidth available on the bus. Because logging is centralized on a board with its own dedicated memory and optional disk, there is no effect on available resources for the CPUs.

Our detailed simulations have shown that the amount of log data is small enough that even for programs that exhibit very fine-grained sharing behavior, several minutes of program history can be kept. Because the logging process does not intrude upon or degrade normal execution, logging can be done continuously during normal operation. If a bug occurs, the programmer can replay the last several minutes of execution repeatedly until the bug is found. If the logging device is being used for fault-tolerance purposes, the log can be used to recreate the exact system state prior to the crash.

Previous debuggers for parallel programs have not been able to perform well with programs that make maximal use of available parallelism, because of overheads in both time and space to perform logging. By reducing log entries to 2 bytes, we are able to keep up with a pipelined multiprocessor bus like the Symmetry's even when it is running at full speed.

Our system allows "debug mode" to be the default and allows all the available parallelism on the machine to be used without compromising access to high-level debugging facilities.

References

[1] Bacon, D. F., and Goldstein, S. C. Hardware-assisted replay of multiprocessor programs. Tech. Rep. 91/624, Computer Science Division, University of California, Berkeley, CA 94720, 1991.

[2] Balzer, R. M. EXDAMS: EXtendable Debugging and Monitoring System. In *Proceedings of the AFIPS Spring Joint Computer Conference* (1969), pp. 567–580.

[3] Cargill, T. A., and Locanthi, B. N. Cheap hardware support for software debugging and profiling. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987), pp. 82–83.

[4] Eggers, S. J., and Katz, R. H. The effect of sharing on the cache and bus performance of parallel programs. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 257–271.

[5] Fidge, C. Partial orders for parallel debugging. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging* (1989), pp. 183–194.

[6] Forin, A. Debugging of heterogeneous parallel programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging* (1989), pp. 130–139.

[7] Hewlett Packard. *Precision Architecture and Instruction Set Reference Manual*, third ed., September 1989.

[8] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[9] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers C-36*, 4 (April 1987), 471–482.

[10] Lovett, T., and Thakkar, S. The Symmetry multiprocessor system. In *International Conference on Parallel Processing* (August 1988), pp. 303–310.

[11] McDowell, C. E., and Helmbold, D. P. Debugging concurrent programs. *ACM Computing Surveys 21*, 4 (December 1989), 593–623.

[12] Mellor-Crummey, J., and LeBlanc, T. A software instruction counter. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 78–86.

[13] Okrafka, B. Personal communication. University of California, Berkeley, 1990.

[14] Pan, D. Z., and Linton, M. A. Supporting reverse execution of parallel programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging* (1989), pp. 124–129.

[15] Pratt, V. Modelling concurrency with partial orders. *International Journal of Parallel Programming 15*, 1 (1986).

[16] Taub, D. M. Improved control acquisition scheme for the IEEE 896 Futurebus. *IEEE Micro* (June 1987), 52–62.

[17] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers 37*, 8 (August 1988), 909–920.

[18] Veal, B., Sweazey, P., and Hawley, D. Technology-independent bus speeds multiprocessing systems. *Computer Technology Review* (Spring 1987), 31–35.

[19] Zelkowitz, M. V. Reversible execution. *Commun. ACM 16*, 9 (September 1973), 566.