DESIGN AND EVALUATION OF THE ROLLBACK CHIP: SPECIAL PURPOSE HARDWARE FOR TIME WARP

Richard M. Fujimoto, Jya-Jang Tsai,
and Ganesh C. Gopalakrishnan

July, 1988

Tech. Report. UUCS-88-011

DESIGN AND EVALUATION OF THE ROLLBACK CHIP: SPECIAL PURPOSE HARDWARE FOR TIME WARP[1]

Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh C. Gopalakrishnan

Abstract — The Time Warp mechanism offers an elegant approach to attacking difficult clock synchronization problems that arise in applications such as parallel discrete event simulation. However, because Time Warp relies on a lookahead and rollback mechanism to achieve widespread exploitation of parallelism, the state of each process must periodically be saved. Existing approaches to implementing state saving and rollback are not appropriate for large Time Warp programs. We propose a component called the rollback chip (RBC) to efficiently implement these functions. Such a component could be used in a programmable, special purpose parallel discrete event simulation engine based on Time Warp. The algorithms implemented by the rollback chip are described, as well as mechanisms that allow efficient implementation. Results of simulation studies are presented that show that the rollback chip can virtually eliminate the state saving and rollback overheads that plague current software implementations of Time Warp.

Index terms — state saving, rollback, Time Warp, parallel discrete event simulation, VLSI component, special purpose computers.

I. INTRODUCTION

Computer simulation of large, complex systems remains a major stumbling block in many research and development efforts today. Computation requirements continue to grow and far exceed the capabilities of general purpose computing hardware. While *special purpose* hardware has been successfully employed in continuous (e.g., fluid flow models) and *synchronous*, time-stepped (e.g., gate level logic circuits [1]) simulation, no such support exists in the more general realm of *asynchronous*, discrete event simulation. The enormous amounts of computing time required to simulate large communication networks, parallel computer architectures, and battlefield scenarios (to name

[1]An earlier version of this paper appeared in the Conference Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988.
The authors are with the Computer Science Department, University of Utah, Salt Lake City, UT 84112.
This work was supported by ONR Contract Number 00014-87-K-0184, NSF Grant Number MIP-8710874, and a University of Utah Development Grant.

a few) thwart advances in system design and development. In many cases, complex simulations cannot be performed because the computation costs are prohibitive.

Although powerful *general purpose* multiple processor computers are now available, considerable doubt exists as to whether these machines can achieve significant speedups for many large simulation problems. As researchers and scientists are now discovering, speedup is elusive for asynchronous simulation because expensive clock synchronization algorithms are required. These algorithms introduce substantial overheads that often completely negate the benefits of parallel execution (for example, see [2,3]). *All* existing parallel simulation algorithms and speedup techniques have serious limitations, and none are appropriate for many large-scale asynchronous simulation applications.

Among the clock synchronization protocols that have been developed, *optimistic* methods such as the Time Warp mechanism [4] offer the most widespread exploitation of parallelism. Significant speedups have been reported for at least one implementation of Time Warp [5].[2] Alternative clock synchronization protocols (called *conservative* or *pessimistic* protocols) have been developed, but existing approaches have serious limitations (notably, objects and their intercommunication pattern must be statically defined) and yield poor performance for many workloads containing high amounts of parallelism [3,6].

However, Time Warp is not without its difficulties. In particular, Time Warp relies on a *rollback mechanism* to undo the effect of clock synchronization errors. To implement rollback, the state of each process must periodically be saved. Unless efficient mechanisms can be developed, state saving and rollback overheads will cripple Time Warp programs containing large amounts of state. Efficient implementation of state saving and rollback is the subject of this paper.

In order to attack the state saving and rollback problem, we define a component called the *rollback chip*[3] *(RBC).* Rather than copying data into "protected" memory areas, the rollback chip manipulates addresses generated by the CPU in order to avoid overwriting data that may later be required after a future rollback operation. The RBC can be viewed as a special type of memory management unit and data cache combined into a single component.

[2]For example, a speedup of 10.66 using 24 processors was reported for a military application; similar speedups for a simulation of colliding pool balls have also been reported. We note, however, that these applications contain only a modest amount of state (e.g., a few thousand bytes in each process).

[3]The name "rollback chip" is actually somewhat of a misnomer because current circuit densities preclude a single chip implementation. Nevertheless, we will continue to use this terminology because we expect a single chip implementation will be feasible in a few years.

A second problem that can degrade performance of Time Warp programs arises when rollbacks occur too frequently. A form of thrashing will result whereby processes are forced to undo most of the computation they perform. Avoidance of *rollback thrashing* is a topic of current research, and is not discussed further here. We note, however, that conservative algorithms will fare no better if rollback thrashing prevails because rollback would be replaced by excessive waiting.

The envisioned system is a message-based multicomputer, e.g., a hypercube machine similar to the Intel iPSC^{TM} or NCube/Ten^{TM}[7], with a rollback chip embedded in each computation node to implement state saving and rollback for that node. Such a machine would be programmed using conventional object-oriented or process oriented (CSP-like) paradigms. Each node could be implemented as a single board microcomputer containing the RBC, a conventional microprocessor, interprocessor communication circuitry, and memory components. Such a board is currently under development. Alternatively, the RBC and conventional memory components could be used to implement a special memory board with state saving and rollback capabilities. Such a board could be plugged into an existing multicomputer system, subject, of course, to physical constraints.

Although parallel discrete event simulation is the principal application area currently envisioned for the rollback chip, use of the chip is *not* restricted to simulation. Time Warp may also be applied to applications such as distributed database concurrency control [8] and parallel execution of prolog programs [9]. The rollback chip may be directly applied to these applications. Further, the ideas used in the RBC design (if not the chip itself) are applicable to virtually any system that requires efficient state saving and rollback capabilities.

The remainder of this paper is organized as follows. The next section discusses alternative approaches to attacking the state saving / rollback problem. We argue that existing approaches to state saving and rollback are not appropriate for Time Warp programs. Section III describes the interface provided by the rollback chip. The algorithm implemented by the RBC is described in section IV. Special mechanisms are proposed in section V which efficiently implement the proposed algorithm. Finally, results of performance evaluation studies are presented in section VI.

II. Alternative Approaches to State Saving and Rollback

The anticipated application domain necessitates use of unconventional methods for attacking the state saving / rollback problem. We will first enumerate properties of Time Warp programs

that are relevant to this problem, and then discuss deficiencies that arise in existing approaches when they are used in the context of Time Warp programs.

In a typical Time Warp based simulation program:

- *The amount of state in each process can be quite large, and the portion that is modified by a single simulation event may be highly variable.* Simulation programs are notorious "memory hogs." For example, it has been reported that military simulations require terrain data objects containing over a megabyte of state [10]. While some events modify only a small portion of the state, e.g., a road washed out by a rain storm, other more catastrophic events may require substantial state changes. Similarly, *continuous* simulations embedded in Time Warp programs may modify substantial amounts of state at each "time-step" of the continuous simulation, while discrete events modify only a few state variables.

- *State save operations must be performed relatively frequently.* This is because:

  - *The most natural point at which to perform a state save operation is after processing each simulation event.* Many discrete event simulation programs execute on the order of 1,000 events per second on a 1 MIP (million instructions per second) processor, implying a state save occurs every millisecond.[4] Further, modern microprocessors such as the INMOS Transputer that are designed to be embedded in parallel systems will achieve computation rates of 10 MIPs or more, and provide rapid interprocessor communication, process scheduling, and context switching. Ignoring state saving overhead, one can expect future parallel simulators to save state much more frequently, perhaps as often as every 100 microseconds, assuming a state save is performed after each event. Although this figure is highly application and implementation dependent, it illustrates in general terms the frequency at which state save operations may be requested.

  - *Less frequent state saving leads to inefficient execution.* Infrequent state saving can severely degrade the efficiency of the Time Warp mechanism because rollback distances are often small, e.g., only one or two events, and rollbacks can be relatively frequent (yet not so frequent as to induce thrashing; see below) [11]. If state saving were performed

[4] Such event processing rates have been observed for at least one parallel simulator [6]; sequential event list simulators routinely run at faster event rates.

infrequently, one would often be forced to roll back the computation much further than is strictly necessary in order to reach the last saved state. This necessitates much additional recomputation to recreate the desired state.

- *Rollback occurs with sufficient frequency that the overhead associated with rollback cannot be ignored.* Time Warp programs that were designed with a relatively coarse grain of computation (several milliseconds per event) have been observed to roll back several times per second within each processor, and still exhibit good speedup characteristics [5]. We expect that much higher rollback rates, e.g., tens or hundreds of rollbacks per second in each processor, will be considered acceptable for other programs using finer grains of computation. Therefore, mechanisms that reduce the cost of state saving at the expense of an expensive rollback operation (e.g., requiring extensive copying to restore the state) are not appropriate.

Existing *software-based* approaches to state saving and rollback were *not* developed in the context of Time Warp programs, and incur unacceptable overheads when used under conditions such as those described above. Current implementations of Time Warp, using general purpose hardware, copy the entire state of a process on each state save operation; this is clearly out of the question when dealing with large amounts of state and frequent state saving. Incremental copying based on compile time flow analysis has difficulty dealing with arrays and dynamic storage, and incurs a substantial compile-time overhead [10,12]. Incremental copying based on an extensive runtime system, e.g., using dirty bits on conventional paging hardware to locate modified pages, requires extensive page table searches; copying also becomes a substantial overhead if much of the state is modified between state saves.[5] Finally, performing runtime checks on each memory write incurs *both* a substantial overhead on each write operation, and expensive rollbacks. Although variations of this latter approach are possible, such techniques essentially degenerate to using software to simulate the actions of the rollback chip, and incur unacceptable overheads.

State saving and rollback mechanisms have been used extensively in the context of fault tolerant computation to allow recovery from transient and/or permanent failures. The *recovery caches* described by Lee, Ghani, and Heron for the PDP-11 [13] and by Feridun, Lee, and Shin for a

[5]We recently learned that techniques using dirty bits have been developed independently by (1) Linton, and (2) Feldman in the context of debugging parallel programs; there, the overheads may be manageable because state saving and rollback occurs less frequently.

fault-tolerant multiprocessor [14,15] have goals that are similar to the rollback chip. However, the approaches that they use reduce the cost of state saving overhead at the expense of the rollback operation — extensive copying may be required on each rollback. While this is reasonable in the realm of fault tolerant computation where errors (and therefore rollback) can be assumed to occur infrequently, it is *not* an appropriate strategy here. Further, these schemes introduce certain additional overheads; in [13] a memory read must precede each write, and in [14,15] extensive copying (for state saving, in addition to that required for rollback) would be required for many Time Warp programs, and excessive amounts of memory are needed.

Time Warp programs require efficient mechanisms that allow state saving and rollback operations to be performed rapidly, ideally in constant time, independent of the size of the process state or the amount of state that is modified between state save operations. It should also be relatively efficient in memory usage. The central contribution of this paper is to propose such a hardware mechanism, and to evaluate its performance.

III. The Roll Back Chip Interface

The rollback chip implements state saving and rollback functions for a single processor in the multicomputer system. It provides each Time Warp process with a data segment known as *version controlled memory (VCM)*. Version controlled memory has identical semantics as ordinary read/write memory, except the process may, at any time, "mark" the state of the memory as one that it may later want to restore via a *ROLLBACK* operation. In a parallel simulation, the processor will normally issue a MARK operation whenever it finishes processing a simulation event. All variables that are subject to state saving and rollback must be stored in version controlled memory.

The RBC only acts on memory references to version controlled memory. Memory references for instruction fetches and local variables that do not have to be restored on rollback (for example, in parallel simulation, local variables typically do not persist from one event to the next) bypass the RBC. In our current design, each version controlled memory data segment may contain up to 4 megabytes of storage, and a single processor may contain up to 64 independent VCMs. This design statically maps VCMs into the processor's address space; for a 32 bit address space, the mapping for 64 data segments uses only 256 megabytes of the address space, leaving 3.75 gigabytes

for conventional memory and memory mapped I/O devices. Throughout the remainder of this paper, we will assume that each process is allocated at most one VCM.

The rollback chip supports six operations: RESET, memory READ, memory WRITE, MARK, ROLLBACK, and ADVANCE. Each operation is assumed to operate on the VCM of the currently executing process. The semantics of these six operations are described below:

RESET. Initialize the rollback chip prior to the execution of a Time Warp program. Certain initialization parameters are also required, however, this is beyond the scope of the present discussion.

MARK. Mark (preserve) the current state of version controlled memory.

WRITE(A,D). Write data D into memory address A.

READ(A):D. Read the most recently written version of data associated with address A (excluding rolled back write operations) and return this data D to the CPU.

ROLLBACK(k). Restore the version controlled memory to the kth previously marked state ($k > 0$).

ADVANCE(k). The k oldest marked states are no longer required, and can be *fossil collected.* During fossil collection, resources that are no longer needed are reclaimed, and irrevocable operations (e.g., I/O) are performed. Determination of which saved stated may be safely fossil collected is made by computing a bound on the longest possible rollback. Computation of this bound (called Global Virtual Time or GVT) is beyond the scope of the current discussion, but is described elsewhere [4].

The RESET, MARK, ROLLBACK, and ADVANCE operations may be invoked by the CPU by writing into the RBC's control registers which are memory mapped into the processor's address space. The READ and WRITE operations represent references to program variables that are generated by the CPU during the normal course of its operation. The CPU also has access to other registers within the rollback chip (described later) to implement context switches between processes.

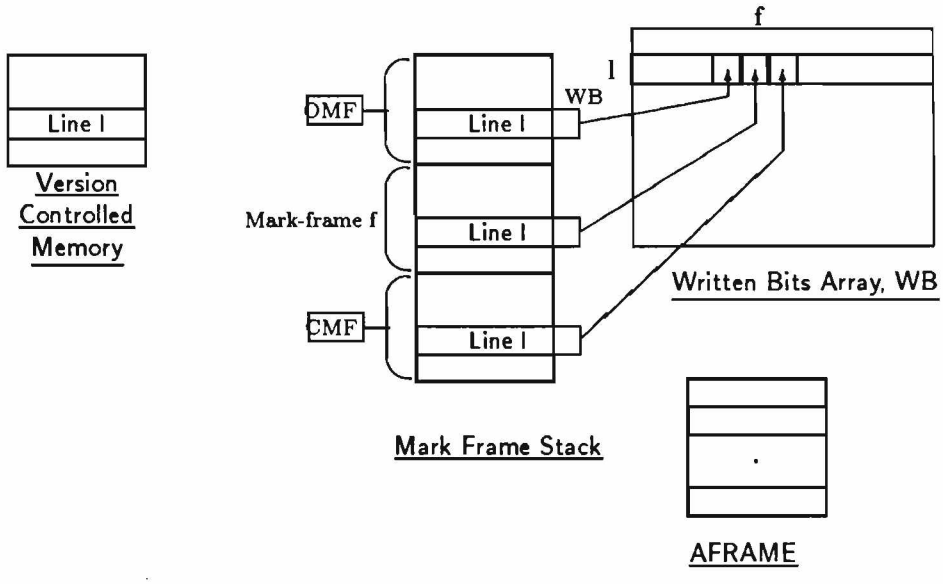

Fig. 1: Data structures used by the RBC algorithm.

IV. The Rollback Chip Algorithm

The discussion that follows will focus on the operation of a single VCM. The rollback chip must maintain different *versions* of each state variable to enable a previous version to be restored. Different versions of the same variable are stored in separate storage areas called *mark frames* (see figure 1). Each mark frame is the same size as version controlled memory, and is divided into some number of fixed length *lines*. An RBC line is similar to a line in a conventional cache memory system; it is transparent to the processor and serves as the quantum of data accessed on each reference to physical memory.

Mark frames are organized as a stack. The stack is implemented as a circular buffer, but to simplify the present discussion, we will assume the stack is unbounded in length. The *current mark frame* or *CMF* refers to the frame at the top of the stack. The CMF register in the rollback chip contains a pointer to this frame. Also, the *oldest mark frame* or *OMF* refers to the frame at the bottom of the stack. Frames older (deeper in the stack) than the OMF are no longer needed so their storage may be reclaimed.

The RBC operations defined earlier can be easily explained in terms of this stack-based im-

plementation. The RESET operation resets the CMF and OMF registers to 0. Each MARK operation pushes a new frame onto the stack by incrementing the CMF register. *No data is copied on MARK operations.* Memory writes access the CMF.[6] The memory write is accomplished (in part) by concatenating the CMF register with the address generated by the CPU to create a new memory address for the write operation. Because the MARK operation does not copy data into the newly acquired frame, mark frames usually contain "holes" where no valid data has been written. Therefore, read operations must search through the stack starting at the CMF to locate the *most recent version* of the data. The RBC caches recently used "most recent version" data to reduce the amount of searching that is actually required. Finally, ROLLBACK(k) pops k frames from the stack by decrementing the CMF register, and ADVANCE(k) removes the k oldest frames by advancing the OMF register.

Two additional aspects of the RBC must be described. First, because each mark frame will usually contain holes, flag bits are required to indicate which lines contain valid data. Second, the ADVANCE operation, as described above, may accidentally discard needed data, so additional precautions are required. These two aspect of the RBC algorithm are described next.

A. *Written Bits*

A *written bit* (WB) is associated with each line of each mark frame, and is set if that line contains valid data. These bits are logically organized as a two dimensional array (see figure 1): $WB[l, f]$ corresponds to line l of mark frame f. The most recent version of line l is found by searching *row* l of the written bit array starting at WB[l,CMF] until a set bit is found.

B. *The Seldom Written Data Problem*

The ADVANCE operation must do more than simply increment the OMF register. If a variable is written infrequently, its most recent version may be buried far into the mark frame stack. If an ADVANCE operation causes the OMF to overtake this frame, precautions must be taken to ensure that this valid data are not discarded. In general, because the OMF provides a bound on

[6] Actually, writes are more complicated because writes only modify a portion of each line; we defer discussion of this until later.

the deepest possible rollback, the most recent version of the data that is at least as old as the OMF must always be preserved to ensure correct operation of the algorithm.

A special mark frame is defined called the *archive* frame (AFRAME) which holds the most recent version of the line that is older than the OMF. The ADVANCE operation copies the most recent version of each line *among the frames it is fossil collecting* to the archive frame before reclaiming storage used by these frames. Also, if the READ operation does not find any set written bits in its search for the most recent version of a line, it assumes the data is stored in the archive frame.

C. *Logical Description of RBC Operation*

Based on the data structures shown in figure 1, the algorithm implemented by the RBC is depicted in figure 2. MRV denotes the frame number holding the most recent version of the line in question, and may refer to the archive frame. The description of specific operations is straightforward. One point worth noting is that the WRITE operation must first copy the MRV line to the CMF if the CMF written bit is not set. This is necessary because, as alluded to earlier, WRITE operations do not modify the entire line.

The observant reader will notice that there are several operations that initially appear to be very expensive. Special mechanism must be defined to efficiently implement the RBC algorithm. These will be described next.

V. Rollback Chip Mechanisms

We will now focus attention on the implementation of the algorithm described in the previous section. Implementation is a challenging problem because several aspects will be unacceptably slow, inefficient, and/or inflexible if implemented in the obvious way. In particular, the major trouble spots (and proposed solutions) are:

Slow access to MRV data. The most recent version of recently used lines are cached in the rollback chip, allowing READ and WRITE "hits" to be performed at conventional cache memory speeds. Additional optimizations are introduced to reduce the search time required for RB cache "misses."

Slow ROLLBACK operation. Many written bits must be reset on each rollback. An efficient

```
INIT()
     CMF:=0; OMF:=0;
     WB[ln,fr]:=0; for all ln and fr;
end INIT;

WRITE(A, D)
     /* A.Line is the line number field of the address */
     /* A.Word is the word/byte/longword address */
     /* Stack[x,fr] refers to line/word x of stack frame fr */
     if (WB[A.Line,CMF] = 0) then
          Stack[A.Line,CMF] := Stack[A.Line,MRV];
          end-if
     Stack[A.Word,CMF] := D;
     WB[A.Line,CMF] := 1;
end WRITE;

READ(A): D
     return (Stack[A.Word,MRV]);
end READ;

MARK()
     CMF := CMF +1;
end MARK;

ROLLBACK(k)
     WB[ln,fr]:=0 for all ln, CMF-k < fr ≤ CMF;
     CMF := CMF - k;
end ROLLBACK;

ADVANCE(k)
     for each line ln do
          /* OMRV is MRV frame older than OMF+k */
          if (OMRV frame exists) then
               AFrame[ln] := Stack[ln,OMRV];
               end-if
          end-for
     OMF := OMF + k;
end ADVANCE;
```

Fig. 2: The rollback chip algorithm. The stack is assumed to be unbounded.

mechanism called the *rollback history* has been devised to avoid updating written bits when a rollback occurs. Instead, a lazy approach is used whereby the bits are cleared "on the fly" when they are read from the written bit memory.

Slow ADVANCE operation. The rollback chip processes this operation in parallel with the CPU. The processor need not wait for the ADVANCE operation to complete unless it runs out of memory. An additional optimization is introduced to reduce the amount of data that must be copied to the archive frame.

Poor memory utilization. If few state variables are modified between MARK operations, most of the memory in the mark frame stack is wasted. A *dynamic* memory allocation scheme based on paging is used to only allocate physical memory when it is needed.

Multiple processes per processor. The rollback chip mechanisms easily accommodate multiple processes per processor. Techniques similar to those used in translation lookaside buffers in memory management units can be used to enhance performance.

Each of these aspects of the RBC will be discussed in turn, after we introduce the notion of working areas.

A. *Working Areas and Dynamic Growth of the Stack*

The mark frame stack is statically partitioned into blocks of mark frames, each of which is referred to as a working area. Each working area contains a fixed number of contiguous frames. Our current design of the RBC supports 16 working areas, each containing 16 mark frames, for a total of 256 frames or saved states per VCM.

The number of the working area in which a particular mark frame is contained is simply the high order bits of the frame number. The *CWA* and *OWA* refer to the working areas containing the CMF and OMF respectively, and are obtained by extracting the high order bits of the CMF and OMF registers. For example, in our current design, the frame number is 8 bits with the upper nibble indicating the working area, and the lower nibble the frame within the working area.

Using working areas, it is possible to devise a scheme to allow the mark frame stack to dynamically expand beyond the size initially allocated to the circular buffer. One could define a set of

working area registers in the rollback chip, each pointing to a single working area of the mark frame stack. Like the mark frame stack, the working area registers would be organized as a circular queue. When the stack overflows, registers corresponding to working areas at the bottom of the stack could be saved in memory, allowing these registers to be used to accommodate the expanding stack. The saved registers would eventually be garbage collected by successive ADVANCE operations. In the event of a very long rollback, it might also be necessary to load this saved information back into the working area registers.

Though feasible, supporting dynamically expanding stacks adds a nontrivial amount of complexity to the rollback chip design. Also, the use of working area registers adds a significant amount of process-specific state, increasing the cost of context switches, or forcing one to support multiple sets of working area registers. Further, even if dynamic stacks are *not* supported, overflow can be easily handled by blocking the offending process until global virtual time advances sufficiently to allow old frames to be garbage collected and reused. It is improbable that such blocking will diminish performance because processes running out of stack space are far ahead of others, making it unlikely that they are on the critical path of the computation. In practice, we expect that 256 frames is far more than will be required in practice (this intuition is shared by Jefferson, and is supported by empirical data [11]; our initial simulations of Time Warp simulators of communication networks indicate that typical programs only require at most 10 or 20 frames). Our current design of the rollback chip does not use working area registers, and assumes a fixed sized mark frame stack.

B. *The Rollback Cache*

The READ operation must return the most recent version of the data that is being referenced. Searching through a row of the written bit matrix on every READ operation is too expensive, so recently used MRV data is cached. One design of the rollback cache (or RB cache) using a copy back protocol is described in [16]. A simpler design using a write-through protocol is described next.

The fields of each entry in the RB cache are:

Valid is a single bit that is '1' if the cache entry contains valid data, and '0' otherwise.

Line indicates the line number to which the remaining fields correspond. Associative searches are performed on this field.

Data contains the data corresponding to the most recent version of the cached line.

MRV is the frame number where the most recent version resides (both the working area and frame within working area fields).

PID is an identifier indicating the process owning the cached data. This is similar to the address space tag sometimes used in conventional memory management units.[7]

The first three fields (and to a certain extent, the PID field) are identical to those found in conventional caches. The MRV and PID fields are used to selectively invalidate certain cache entries when rollback occurs. A simpler, but less efficient, design is to eliminate the MRV field and invalidate the entire cache on each rollback. Although the selective invalidation operation can be easily implemented using a custom integrated circuit, efficient implementation using *only* off-the-shelf parts would require an excessive number of components, so this latter invalidation procedure may be more appropriate in certain designs.

The operation of the cache will be described next in terms of the READ, WRITE, and ROLLBACK operations. The cache is not affected by the MARK, and only slightly affected by the ADVANCE operation. The latter will be discussed later.

To simplify the discussion, we will assume a fully associative cache is used. The mechanisms are easily adapted to direct addressed and set associative caches. Descriptions of the cache operations are shown in figure 3.

READ and WRITE Operations: The initial address translation for READ and WRITE operations is identical to that of a conventional cache — the line number field is extracted from the address, and is used with the PID field to associatively search the cache (this is not shown in figure 3). If the line was found in the cache, a *hit* occurs. Otherwise, a *miss* results.

The operation of the RB cache for READ hits is *identical* to that of conventional caches. The data is read from the cache and the requested word (or byte or longword) is extracted and returned

[7] Actually, if one assumes different version controlled memories are mapped to different areas of the address space, the PID field is simply the high order bits of the Line field. Here, we identify PID as a separate field to simplify the presentation.

```
Read_Hit (cache entry e, line ln):
     Return (Cache[e].Data);
     /* Actually, only requested byte/word/longword is returned */
end Read_Hit;

Read_Miss (line ln):
     Search for MRV frame;
     Cache[lru].Line := ln;     Cache[lru].Valid := 1;
     Cache[lru].Data := Stack[ln,MRV];
     Cache[lru].MRV := MRV;     Cache[lru].PID := PID;
     /* only requested byte/word/longword is returned */
     Return (Cache[e].Data);
end Read_Miss;

Write_Hit (cache entry e, line ln, data D):
     Cache[e].Data := D; /* only modify part of line */
     Cache[e].MRV := CMF;
     Stack[ln,CMF] := Cache[e].Data;
     WB[ln,CMF] := 1;
end Write_Hit;

Write_Miss (line ln, data D):
     Search for MRV frame;
     Cache[lru].Line := ln;     Cache[lru].Valid := 1;
     Cache[lru].Data := Stack[ln,MRV];
     Cache[lru].MRV := CMF;     Cache[lru].PID := PID;
     Cache[lru].Data := D; /* only modify part of line */
     Stack[ln,CMF] := Cache[lru].Data;
     WB[ln,CMF] := 1;
end Write_Miss;

Rollback (to frame dst):
     for each cache entry e do
          if (Cache[e].PID=PID and Cache[e].MRV > dst) then
               Cache[e].Valid := 0;
               end-if
          end-for
end Rollback;
```

Fig. 3: RB Cache operations.

to the CPU. Access times comparable to those obtained by conventional cache memories can be expected.

For READ misses, the replacement algorithm (e.g., LRU) selects an entry to be deleted from the cache, or an invalid entry is selected if there is one available. The frame containing the most recent version of the line must now be determined by searching through the appropriate row of the written bit matrix. The information associated with the line is then loaded into the cache, and the requested data is returned to the CPU.

WRITE operations always modify data in the current mark frame (CMF). WRITE hits must (1) write the data into the cache entry, (2) write the CMF register into the MRV field of the cache entry, and (3) write the line to the CMF in memory (recall a write through protocol is used). In addition, the corresponding written bit must be set. Memory write requests can be buffered, so the CPU may be allowed to continue pending their completion.

A write miss operation is essentially a read miss that is immediately followed by a write hit. The MRV frame is found; the requested line is then read from memory, modified, and written into both the cache and the CMF in memory.

Optimizing MRV Searches: Cache misses require a search for the most recent version of the line that has been referenced. Even though cache misses are infrequently (assuming the program exhibits reasonable locality), overall performance may be significantly degraded if misses are very expensive. Fortunately, several techniques are available to reduce the time of MRV searches:

1. The written bit memory is organized so that sixteen written bits (a single working area) for a single line are read on each memory reference, allowing the hardware to scan 16 mark frames on each iteration. If the size of the mark frame stack is at most 17 frames, the MRV frame will always be found after only one or two references to the written bit memory.

2. The written bits may be stored in high speed (relative to main memory) static RAM.

3. The search procedure may be pipelined. The scan of the first 16 written bits can be overlapped with scans of subsequent blocks of 16 bits, allowing the results of successive scans to be available on subsequent clock cycles.

4. A simple optimization (described below) is available to significantly shorten the search time

for very large mark frame stacks.

Optimization (4) uses the following rules: write the MRV field for each line to a special location in memory for that line whenever the line is deleted from the cache; when the line is next referenced (causing a cache miss), start the search from this previously saved MRV frame rather than the CMF. The saved MRV information will ***usually*** point to the most recent version of the line. It will not if the saved MRV information was invalidated by a rollback operation after the line was deleted from the cache, but before it was referenced again.[8] The ADVANCE operation should also update this information if it fossil collects the MRV data.

We call this technique the *LastWA* optimization, because in practice, one would only store the number of the *working area* containing the MRV of the line. $LastWA[i]$ indicates the last working area into which valid data were written for line i. We will later present performance results that indicate that the LastWA optimization is effective in reducing search times for large mark frame stacks.

Among the four optimizations described above, the first two have the clearest and most direct benefit, and should always be used. The latter two optimizations can be employed if the size of the mark frame stack is expected to be large. Using some combination of these four optimizations, we expect that in practice, the penalty of cache misses can be sufficiently reduced so that misses do not appreciably degrade performance.

ROLLBACK Operations: A ROLLBACK of k frames invalidates any information written into the top k frames of the stack. If any of this data are buffered in the RB cache, they must be invalidated. Also, the RB cache should avoid invalidating cache entries that are being used by processes other than the one being rolled back.

The invalidation operation can be easily implemented using a custom memory chip with embedded comparison logic. The chip holds the valid bit, MRV, and PID fields. An entry is invalidated if its PID matches that of the process being rolled back, and if its MRV field is *greater than* the frame number of the destination of the rollback (the new CMF). Using a custom chip, the invalidation operation can be performed in parallel across all RB cache entries.

[8] One could update this saved MRV information on rollback, however, this would make the rollback operation relatively expensive.

Because the mark frame stack is implemented as a circular buffer, the "greater than" operation must be performed using modulo arithmetic. This can be implemented by providing an extra bit of precision with the MRV field of the cache and using ordinary magnitude comparison logic. The extra precision bit of the "dst" field (the destination frame for the rollback), or any data written into the MRV field of the cache is set if it is less than the OMF register. The extra precision bit of each MRV field in the cache is cleared when the OMF register wraps around.

C. *Rollback Histories*

The RBC algorithm requires that a rollback operation clear all of the written bits corresponding to frames that are rolled back, i.e., popped from the stack. A brute force implementation of this operation will be too expensive for programs containing large amounts of state. An obvious alternative is to clear all written bits for new frames that are pushed onto the stack. However, this simply transfers the problem to the MARK operation, making it too expensive. The rollback history (RBH) mechanism is designed to efficiently clear the appropriate written bits when a ROLLBACK occurs.

The key idea used by the rollback history mechanism is that *no written bits stored in memory are cleared when a rollback occurs; instead, the written bits are cleared on the fly as they are read from the written bit memory (e.g., following a cache miss).* This dramatically improves the efficiency of the rollback operation, at the cost of a small increase in the cache miss penalty.

Using this "lazy" approach, the written bits in memory may not be updated until long after the rollback occurred. Therefore, the question that must be answered is "which written bits must be cleared when they are read from the written bit memory?" The *rollback history (RBH)* mechanism provides this information.

One can rephrase to above query to ask an equivalent question: "what is the deepest rollback that has occurred since the written bits were written into memory?" If the written bits were written to memory at time t (meaning they were correct and up to date at time t), and the deepest rollback that has occurred since time t was to frame f, then the correct written bits are those stored in the memory with all bits in frames newer than f cleared. Therefore, one approach to this problem is to:

1. Define an array of values $RBH[t]$ such that $RBH[t]$ indicates the deepest rollback that has occurred since time t.

2. Whenever a block of written bits are written to memory (normally, 16 written bits will be written at one time), store a timestamp ts with the written bits indicating the current time.

3. Whenever the written bits are read from memory, read the timestamp ts that is stored with them, and clear all bits corresponding to frames that are newer than (greater than) $RBH[ts]$.

Although this approach efficiently implements the bit clearing operation, it has a serious flaw: the size of the RBH array must have an infinite number of entries because the index t is a continuous quantity. This problem is resolved by observing that $RBH[t]$ (the deepest rollback since time t) is identical to $RBH[t+\Delta t]$ if no rollbacks occurred between t and $t+\Delta t$. Therefore, the RBH entries corresponding to times between two consecutive rollbacks can be represented by a *single* RBH entry. This is equivalent to saying that "time," from the perspective of RB histories, is measured by the number of rollbacks that have occurred since the computation began. Each rollback increases RBH time by one unit. The timestamp, described above, is simply "the number of rollbacks that have occurred since the computation began." One need only maintain a counter that is incremented each time the process is rolled back, and use this counter to generate timestamps when written bits are written to memory.

With the above modification, the lazy approach to clearing written bits can be implemented very efficiently. The only question that remains is maintaining the RBH array. The RBH array can be viewed as a stack, with a new element pushed onto the stack whenever a rollback occurs. Stack elements are *never* popped from the top of the stack, however, the oldest entries at the bottom of the stack may be garbage collected. Technically, the RBH mechanism is actually a FIFO queue, but we shall refer to it as a stack to facilitate the presentation. It is actually implemented as a circular buffer, so all of the arithmetic described below is implicitly modulo arithmetic.

$RBH[i]$ indicates the destination frame number of the "deepest" rollback that has occurred *after* the *ith* rollback ($i+1$, $i+2$, etc.), or equivalently, after the stack element was created. The top of stack element always contains the value INFINITY because no subsequent rollbacks have yet occurred. When a rollback occurs, one must update the RBH stack — if the destination dst of the

```
RBH-UPDATE(dst)
     i := CRBI;
     while (dst < RBH[i])
          RBH[i] := dst;
          i := i-1;
          end-while
end RBH-UPDATE;
```

Fig. 4: Update operation for RBH stack for rollback to frame dst.

current rollback is deeper than (less than) $RBH[i]$, then *dst* should be written into $RBH[i]$. At first glance, this would imply the entire RBH stack must be examined on each rollback. Fortunately, this is not the case.

It is easy to see that the condition $RBH[i] \leq RBH[i+1]$ must always be true — the deepest rollback since time i must clearly be at least as deep as the deepest rollback since time $i+1$.[9] Therefore, if rollback history entries are updated from the most recent to the oldest, we can stop the updating process *as soon as a rollback history entry is encountered with a rollback as deep or deeper than the destination of the current rollback.* If the rollback is relatively short (i.e., if there is temporal locality), very few rollback history entries will have to updated. The update procedure for the RBH stack is shown in figure 4.

The update procedure may be efficiently implemented by buffering the top portion of the stack in a special custom memory with embedded comparison logic to update stack entries in parallel whenever a rollback occurs. The remainder of the stack is stored in conventional RAM. Only long rollbacks will require updates to the rollback history elements that are stored in RAM, so the entire rollback history stack can usually be updated very rapidly. After performing extensive simulations of the rollback chip (described later), we have never observed more than ten entries of the RBH stack updated on a single rollback — on average, only two to three entries are updated (one entry, the top of stack, is *always* updated on each rollback). Therefore, by buffering only a modest number of RBH entries in the custom chip (say 16), one would expect that the entire stack can be updated in a single clock cycle.

[9] Another way of seeing this is to observe that the set of rollbacks since $i+1$ are a subset of those since i.

Garbage collecting the rollback history (from the bottom of the stack) is straightforward. A variable called $TAGBOUND[wa]$ is associated with working area wa that holds the pointer to the top of the rollback history stack ($CRBI$ or current roll back index) when wa was *first* created by a MARK operation (i.e., recreation following subsequent rollbacks is ignored). $TAGBOUND[wa]$ is a lower bound of any tag (timestamp) written into working area wa. Consider two consecutive working areas, wa and $wa + 1$, that are currently in use. Because $wa + 1$ must have been created *after* wa was created, and CRBI is always increasing in value (in the modulo sense), then $TAGBOUND[wa] \leq TAGBOUND[wa + 1]$. It immediately follows that $TAGBOUND[OWA]$ is a bound on the smallest tag in use by any working area. Thus, when working area wa is fossil collected, rollback history entries up to, but not including $TAGBOUND[wa+1]$ may be reclaimed.

D. *The ADVANCE Operation*

The ADVANCE operation is responsible for reclaiming storage that is no longer required. It is convenient (and more efficient) to process an entire working area at a time rather than on a frame-by-frame basis. One could, in fact, garbage collect *several* working areas at one time if even greater efficiency is desired, although this will complicate the mechanism somewhat. Reclamation of memory resources is performed in parallel with the execution of user code to enhance performance.

The ADVANCE operation has only a minor effect on the RB cache. If the ADVANCE operation fossil collects data that is stored in some entry of the cache, the MRV field of that cache entry will become out of date. However, even if this MRV information is left out of date, the cache will still operate correctly because the MRV information is only used during the invalidation operation when a rollback occurs; the worst that could happen is a cache entry might be accidentally invalidated by a rollback. Accidental invalidation might degrade performance slightly, but does not compromise correctness (recall a write-through cache is used; additional mechanisms are required if a copy-back protocol were used). If desired, accidental invalidation could be avoided by resetting the MRV field to a special state that cannot be invalidated by rollback whenever the MRV frame is fossil collected.

Because the ADVANCE operation proceeds in parallel with other RBC operations, some care must be taken to avoid race conditions. In particular, the ADVANCE operation copies lines from working areas into the archive frame concurrently with memory operations that may also access

the archive frame. Race conditions can be avoided by simply delaying the increment of the OMF register until all data copying is completed. This avoids races because: if the most recent version of the line is in the archive frame, there are no *set* written in the working area that is being garbage collected, so the ADVANCE operation performs no data copying and no race condition can occur. On the other hand, if the most recent version is *not* in the archive frame, READ and WRITE operations will never access the archive because they only reference the most recent version of data. Again, the correct MRV information will be accessed, so no race condition is possible.

Finally, a simple optimization can be used to reduce the amount of data copied to the archive frame. If there is at least one set written bit in a frame that is newer than the working area being garbage collected but still at least as old as the value of the OMF after the ADVANCE is complete, then the data need not be copied to the archive frame. This requires the ADVANCE operation to read some additional written bits to determine if it need not copy the data, however this is less expensive than copying the line.

E. *Improving Memory Utilization*

The rollback chip would use an unreasonable amount of storage if physical memory were allocated for the entire mark frame stack of each VCM. Further, most of this memory would be wasted if the Time Warp process modified only a small portion of its VCM between successive MARK operations, or if the actual stack size were much less than that allocated to the circular buffer. This problem is addressed in the RBC by *not* allocating physical memory until it is actually needed. The proposed approach is very similar to demand paging in conventional computers (no disks or I/O are used in the scheme proposed here, though they could be easily added), so not surprisingly, the required mechanisms are similar. Further, a similar scheme may also be used to economize on memory used to hold the written bits.

Addresses for the mark frame stack are created by concatenating a frame number with the address generated by the CPU. These are actually *virtual addresses* that are passed to a memory management unit portion of the RBC for translation to a physical address. Each page table entry contains a presence bit that is set if physical memory has been allocated for the page, and reset otherwise. If the presence bit is set, the page table entry also contains a pointer to the page. A list

of free pages is maintained. A new page is allocated from the free list and mapped into the address space on the first memory write into the page (a read corresponds to an access to uninitialized data). Pages are reclaimed by the ADVANCE operation. Techniques used in virtual memory systems (translation lookaside buffers, hierarchies of page tables, etc.) are equally applicable here.

F. *Multiple Processes per Processor*

The RBC state for a single VCM that must be swapped on context switches consists of a few miscellaneous registers (CMF, OMF, CRBI, etc.) and the top of the RBH stack. As discussed earlier, the latter may be reduced to only a few words of storage without significantly degrading performance. Multiple copies of these registers may be stored in the RBC to facilitate rapid context switches, or they may be swapped by the CPU itself. "Synonym" problems associated with traditional virtual memory caches [17] are not a problem in the RBC because Time Warp excludes shared memory between processes.

G. *Implementation of the Rollback Chip*

A block diagram of one possible implementation of a multicomputer node using the rollback chip is shown in figure 5. The CPU provides the computation power for the node and circuitry for interprocessor communications (possibly implemented as a separate coprocessor). We assume the CPU has a conventional cache associated with it to hold instructions and local (non-VCM) variables. This is necessary to reduce memory contention with the RBC; the latter performs storage reclamation activities in parallel with the CPU. Bulk memory contains conventional dynamic RAM. The rollback chip hardware includes:

- *The control unit* (e.g., a microcode sequencer and ROM) to implement storage reclamation and other miscellaneous functions.
- *The RB cache*, including a circuit for implementing the rollback invalidation function.
- *Written bit memory*, implemented with fast static RAM. The RBH stacks should also be stored here or in a separate high speed memory.
- *A memory management unit (MMU)* to implement the dynamic memory allocation scheme.

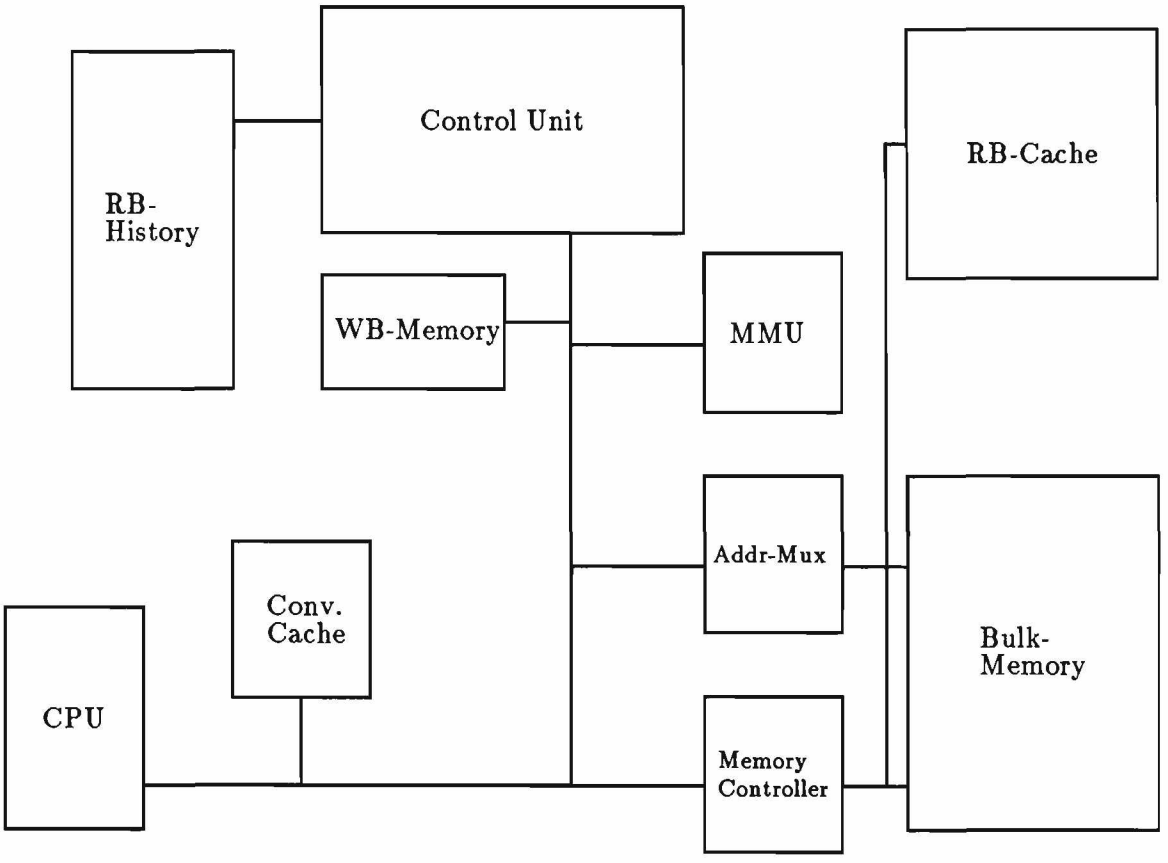

Fig. 5: Configuration for each node of the simulation engine.

- *One or more RBHistory units*, which buffer the top portion of the the RBH stack, and provide circuitry to allow rollback updates to be performed rapidly.

Using current technology, the RBC contains too much circuitry to be implemented as a single chip. However, excluding the static RAM portions of the chip, it could be implemented as a chip set of perhaps two or three VLSI components. Assuming circuit densities continue to grow as they have in the past, we expect that a single chip implementation of the RBC will be feasible within a few years.

Ideally, the rollback chip would be used with a custom processor. However, pragmatic considerations make it highly desirable to use off-the-shelf microprocessors. Many modern microprocessors contain an on-chip data cache. The RBC can be used with such components if appropriate precautions are taken. The most straightforward solution is to ensure that version control memory is never cached, or to simply disable the cache completely. Alternatively, the on-chip cache would have to be invalidated when rollback occurred. Further, if the microprocessor's cache uses a copy-back policy, the cache would have to be flushed before each MARK operation. Similarly, for any processor that is used, one must be sure that internal processor registers are written to memory before each MARK operation if they must be restored on rollback.

Some microprocessors also contain an on-chip memory management unit. In this case, the RBC would have to reside between the MMU and physical memory, and receive physical memory addresses. The RBC assumes, however, that each version controlled memory occupies a contiguous portion of the address space. Therefore, the MMU address mapping would have to be controlled to ensure that this condition is not violated.

VI. Performance Evaluation

The overhead incurred by the rollback chip has been evaluated through extensive simulation studies of the RBC mechanisms. In particular, we focus attention on the performance of the RB cache. The MARK and ROLLBACK operations require only a few clock cycles, assuming the RBH stack can be updated in a single clock cycle, as was discussed earlier.

The value of the rollback chip relative to a software based implementation of state saving and rollback using copying is greatest when (1) the amount of state is large (e.g., a megabyte), and/or

(2) the application makes checkpoints very frequently (e.g., every few hundred microseconds). However, even for modest sized states, copying may represent a significant overhead. For example, if the processor's memory has a cycle time of 200 nanoseconds and data paths are 32 bits wide, copying 10K bytes of state, would require a minimum of 1 millisecond, assuming the memory can be utilized 100%, and no time is required for instruction fetches. This represents a substantial overhead unless the grain of computation is relatively large.

Rather than compare the RBC to a hopelessly inefficient software mechanism, we compare it to a comparable conventional cache memory with *no state saving overheads.* This will enable quantitative measurement of the cost incurred by the RBC to implement state saving.

A. *Simulation Methodology*

A simulator has been developed for the rollback chip. Partial validation of the simulator was obtained by comparing its operation to an independently developed simulator for a simple, brute force implementation of the RBC algorithm. The two simulators were exercised and compared over several million RBC operations across a wide range of parameter settings.

In the experiments reported here, address traces are generated stochastically from a normally distributed random variable. The mean of the address trace distribution is periodically changed to simulate phase changes in the program. Address traces from an existing Time Warp system were not readily available. Even if such traces could be obtained, they would *not* provide a true characterization of the expected RBC workload because the frequency and distance of rollback operations are timing dependent, and would not reflect operation using the RBC. On the other hand, use of a stochastic workload generator facilitates experimentation — parameters such as degree of locality and the distribution of rollbacks can be easily controlled.

As described earlier, the operation of the RB cache is such that READ and WRITE operations that "hit" can be expected to require the same amount of time as a hit in a conventional cache. Although a write hit in the RB cache may generate additional memory traffic (e.g., to update the written bit memory), the CPU need not wait for these memory accesses to complete, and one or more memory cycles will normally elapse (for instruction and local variable references) before the next RBC operation is initiated.

Therefore, the appropriate questions to be asked are (1) can the RB cache achieve hit rates

comparable to conventional caches, and (2) is the miss penalty in the RB cache significantly larger? One would expect a lower hit rate in the RB cache because portions must be invalidated on ROLLBACK operations. The miss penalty is larger because written bits must be searched. To allow fair comparison, a "comparable" conventional cache is defined as one that is identical to the RB cache and receives the same sequence of operations, but ignores all RBC operations other than READs and WRITEs.

Extensive simulations were performed varying:

- the size of the cache,
- the cache organization (direct mapped, set associative, or fully associative),
- the write policy (copy back vs. write through),
- the size of version controlled memory,
- the locality of the address trace,
- the number of reads and writes between MARK operations (the computation granularity),
- the frequency and distance of rollbacks,
- the frequency of WRITE operations relative to READs,
- the size of the mark frame stack,
- use of the LastWA optimization for cache misses, and
- the frequency at which the mean of the address distribution changes.

Complete details of these experiments are reported in [18]. Numerous experiments were conducted to evaluate the effects of each of these parameters on the performance of the RB cache relative to a comparable conventional cache. Here, we summarize the results of these experiments.

Unless stated otherwise, the performance data discussed here assume the program selects among 4096 lines, and the cache contains 256 entries. It is assumed that there are 16 mark frames per working area. Rollback history tags (timestamps) are 8 bits, allowing both a block of written bits

and its tag to be read in a single memory reference. It is assumed there are four READs per WRITE operation, and locality changes occur every 7.5 events, i.e., every 7.5 MARK operations. Both the RB and the conventional cache use a two-way set associative organization, an LRU replacement policy, and a write through strategy (described earlier for the RB cache). Performance results using a fully associative organization and a copy back policy are similar (when compared to the corresponding conventional cache), and are described elsewhere [16].

B. *Hit Rate*

Rollbacks reduce the RB cache hit rate by invalidating entries. The more frequently rollbacks occur, the more often entries are invalidated. Long rollbacks (potentially) invalidate more entries than short ones. However, it is not possible for rollbacks to be *both* frequent (relative to the frequency of MARK operations) *and* long. If F_{MK} and F_{RB} are the frequency of MARK and ROLLBACK operations respectively, and RB_{dist} is the average rollback distance, then the quantity $F_{MK}/(F_{RB} * RB_{dist})$ indicates the net rate at which events are being processed (e.g., two steps forward for every step back). This quantity is referred to as the *relative event rate*, and must be greater than one or else the computation is going backwards! The latter phenomenon is provably impossible in Time Warp [4].

Several experiments were performed using a variety of rollback scenarios ranging from frequent, short rollbacks, to infrequent, long ones. Rollback distances are selected from a negative exponential distribution, truncated to exclude illegal rollbacks that would move the CMF beyond the OMF. The frequency and distance of ADVANCE operations were controlled to make such illegal rollbacks improbable.

The results of experiments for relatively small grained events (an average of 20 READ and WRITE operations between MARK operations) are shown in figure 6. Results for somewhat larger grained events (200 READs and WRITEs between MARKs) are shown in figure 7. In each graph, the degradation in hit rate is plotted as a function of the hit rate in the conventional cache, which in turn is controlled by adjusting the variance in the probability distribution used to generate the address trace. As can be seen, the degradation in hit rate using the RB cache varies from less than 0.05% to as much as 1.7%.

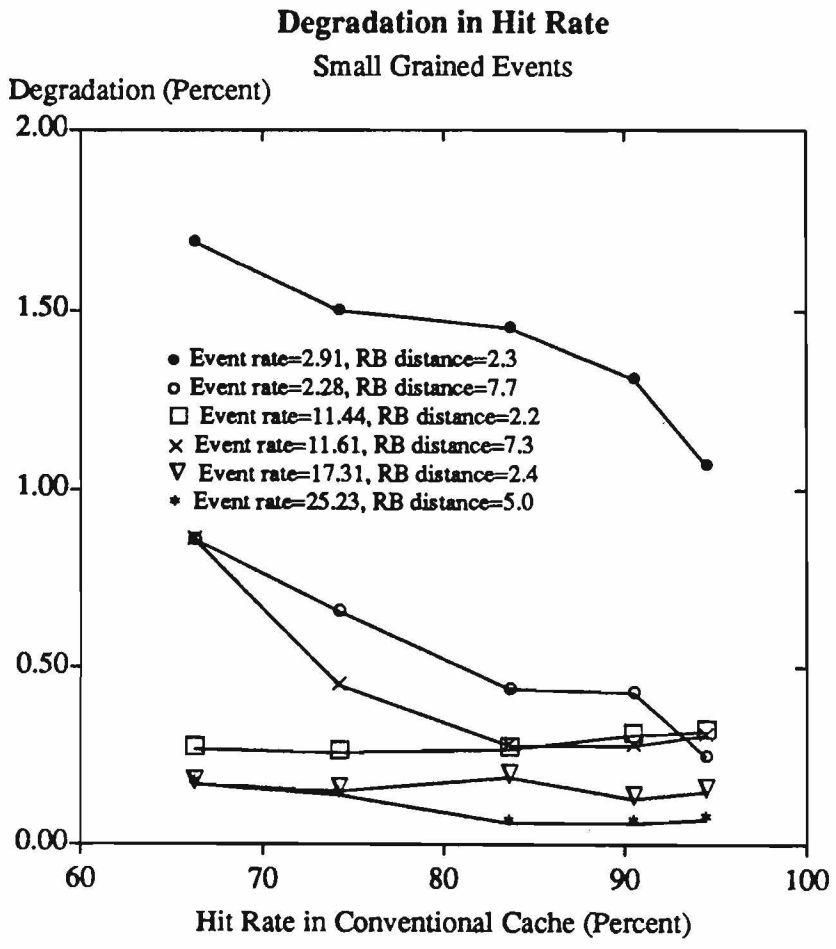

Fig. 6: Hit rate degradation in the RB Cache for small grained events.

The hit rate degradation tends to improve (decrease) as the *absolute* hit rate also improves (increases), especially at very high hit rates. This is because as locality is improved, fewer cache entries tend to be invalidated by rollback; for example, if all memory references were to a *single* memory address, the rollback invalidation operation would only invalidate at most a single entry of the RB cache. This effect is less significant for lower hit rates because the size of the cache then becomes a significant factor; if the cache is too small, the replacement policy will tend to delete entries before the rollback has a chance to invalidate them.

As one might expect, the degradation in hit rate improves (decreases) as the relative event rate also improves (increases); more frequent and/or longer rollbacks cause more cache entries to be invalidated. The situations where the RB cache experiences the most degradation corresponds to

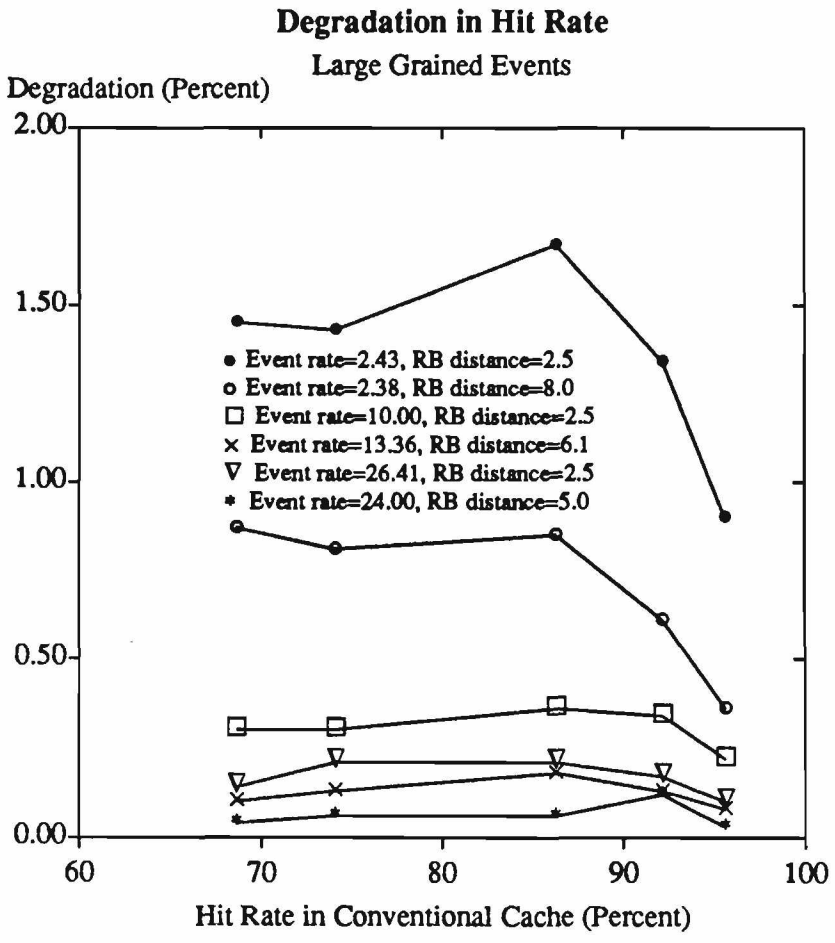

Fig. 7: Hit rate degradation in the RB Cache for larger grained events.

those cases where the event rate is very poor. However, in these situation, the Time Warp program is thrashing, so performance of the RB cache is a mute point. Therefore, only the situations corresponding to "reasonable" event rates (e.g., 8 or 16) are of practical interest. In these cases, the degradation is less than 0.5% in most cases.

C. *Miss Penalty*

Although the hit rate achieved by the RB cache is only slightly lower than that of a comparable, conventional cache, the RB cache's performance could be much worse if the penalty for misses is significantly higher. In particular, an RB cache miss requires the most recent version of the referenced line to be located, requiring a search through the line's written bits.

Overheads incurred by the RB cache on a read or write miss that are *not* incurred in a conventional cache include: (1) the written bits and the associated tag must be read, (2) the RBH stack must be read, and the appropriate written bits cleared, and (3) the page table entry must be read to locate the line data. (1) may be incurred many times on a single miss if the RB cache must search a long distance to locate the MRV frame, and is the principal point of concern. (2) is also required on each iteration of the search, but it incurs a performance penalty on only the first iteration if the hardware is pipelined. The page table reference (3) is only required once at the end of the search. By using a translation lookaside buffer and overlapping access to it with access to the written bit memory, one can eliminate performance degradation for address translations in most situations.

Two search strategies were proposed in the RB cache design. The original approach always begins the MRV search from the CMF frame. An optimization was proposed that begins searching from the "last written" working area (LastWA). This latter approach requires an additional memory reference on each miss to read LastWA.

Additional simulations were performed to determine typical miss penalties and evaluate the usefulness of the proposed optimization. Using the unoptimized search strategy, the search length is the minimum of (1) the size of the stack and (2) the number of MARK operations that have occurred (but have not been rolled back) since data in the referenced line was last written. Therefore, the number of active frames, i.e., $CMF - OMF$, is an important parameter. Secondly, the event granularity will also be important; if each event modifies every line of the state vector, it is guaranteed that searches will only have to look back at most one frame to find the MRV. Finally, one would expect that the rollback distribution will also impact the search length, particularly for the optimized strategy; if no rollbacks occurred, then LastWA will always point to the working area containing the MRV frame.

Results of these simulations are shown in figure 8 for both the optimized and unoptimized strategies. The search distance (number of blocks of written bits that must be read to locate the working area with the MRV frame; recall that 16 written bits are read on each memory reference) is plotted as a function of the number of active frames ($CMF - OMF$). The figures for the optimized version *include* the additional memory reference to access LastWA so that fair comparison can be

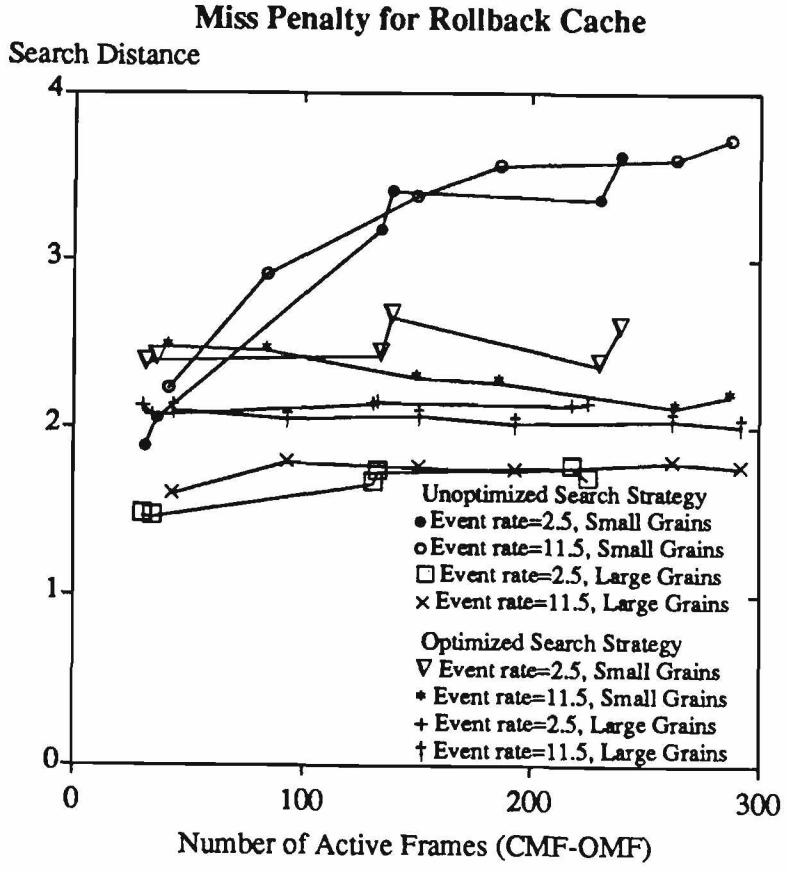

Fig. 8: Search distance for RB Cache misses.

made.

The results are encouraging in that even for long $CMF - OMF$ distances, LastWA in the optimized version usually points directly at the working area containing the MRV information, or close to it. However, for a small number of active frames, the unoptimized scheme is somewhat better because the LastWA value need not be read. Similarly, if large grained events occur that typically modify a large portion of the process state, search distances are also short, so the unoptimized approach is preferred. These results indicate that the unoptimized strategy is adequate, and in fact preferred, in many situations that are expected to arise in practice. If large stacks and small grained events may arise, an adaptive strategy could be used in which the optimization is enabled if it appears that long searches are taking place.

D. *Overall Performance*

The simulation results suggest that when compared to a conventional cache that does not perform any state saving functions, the RB cache can be expected to suffer a small degradation in hit rate, and require 1.5 to 3.0 additional memory accesses on READ and WRITE misses to search through written bits. From these results, an overall estimate of the cost of state saving in the rollback chip can be derived.

The average memory access time for a cache memory system is $P_{hit}T_{cache} + (1.0 - P_{hit})T_{memory}$ where P_{hit} is the probability of a cache hit, and T_{cache} and T_{memory} are the access times to the cache and main memory, respectively. For example, consider a design based on a 30 MHz INMOS Transputer, e.g., the IMS-T800 [19]. Let us assume cache hits can be processed without introducing wait states. For the transputer, this implies an access time T_{cache} of 100 nanoseconds. Assume references to main memory require 200 nanoseconds (T_{memory}), and misses in the RB Cache incur an additional 200 nanosecond penalty (we assume the written bits are stored in fast static RAM). From figure 6, it can be seen that for an event rate of 11.44, average rollback distance of 2.2, and hit rate of 94.5%, the degradation in hit rate is 0.31%. This yields an overall increase in the average memory access time of 11.3%.

Further, it should be pointed out that most memory references do not reference version controlled memory; instruction references, accesses to local variables that do not persist from one event to the next, and code associated with the Time Warp mechanism itself (e.g., for manipulation of input queues; these references constitute a very significant portion of the computation for fine grained events) bypass the rollback chip completely. When taking this into account, overall performance using the RBC will be virtually indistinguishable from that of a CPU with a conventional cache. For instance, if 10% of the memory references access version controlled memory, then the overall cost of state saving in the rollback chip using the parameters listed above is only a 1.1% performance degradation.

Repeating this calculation for the remaining data points in figures 6 and 7 yields the curves shown in figure 9. The cost of state saving and rollback using the RBC is plotted as a function of the hit rate in the conventional cache. As before, T_{cache} is assumed to be 100 nanoseconds, T_{memory} is 200 nanoseconds, and the additional miss penalty in the RBC is 200 nanoseconds. The curves for

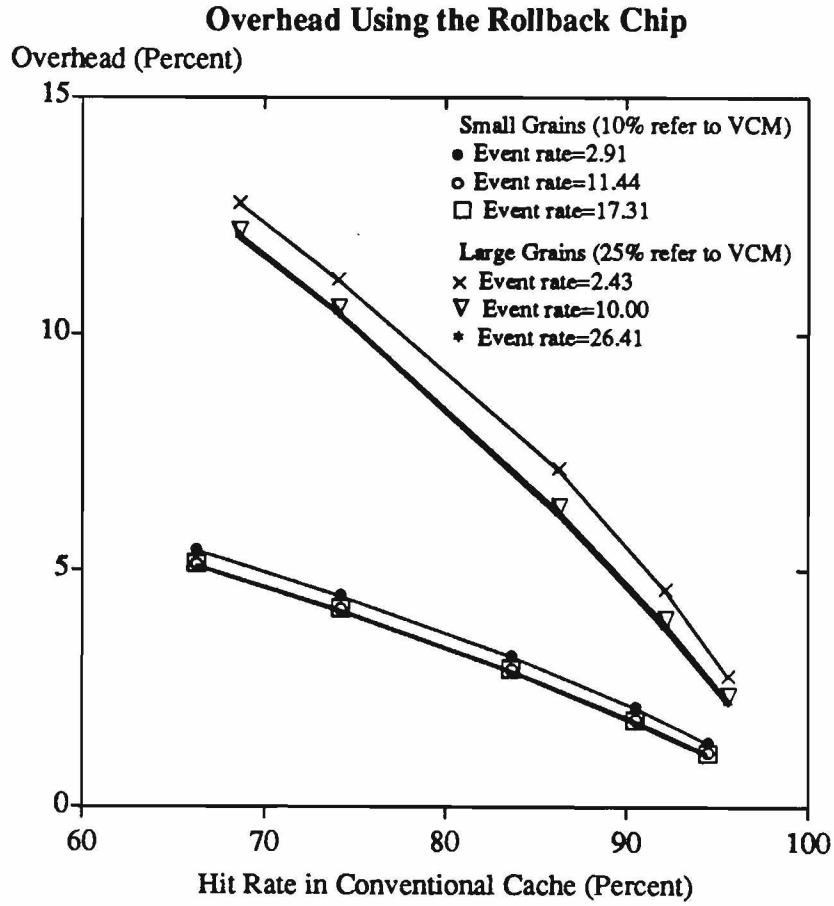

Fig. 9: Overhead for state saving and rollback using the RBC.

short rollback distances (averaging 2.3-2.5 events) are shown; those for longer distances are similar. The curves for small grained events assume 10% of the memory references access the RBC, while those for larger grained events assume 25% (a smaller percentage of references are due to Time Warp overhead as the granularity increases).

RBC performance improves as the hit rate in the cache improves because performance degradation in the RBC only occurs on misses. Further, as noted earlier, hit rate degradation in the RB cache is diminished as the absolute hit rate improves. Today, conventional cache memory systems routinely achieve hit rates well above 90%. Therefore, we expect that the cost of state saving using the rollback chip will typically be only a few percent of processor performance. The RBC will further enhance system performance by performing memory reclamation in parallel with the CPU.

VII. SUMMARY

We have described a special purpose component, the rollback chip, to offload state saving and rollback overheads to special purpose hardware. It is intended as one component of a special purpose, parallel machine architecture that efficiently executes programs based on the Time Warp clock synchronization mechanism. One possible application of such a system is as a special purpose discrete event simulation engine.

The functional operation of the rollback chip has been defined. Simulation results project that a system using the rollback chip will only incur a few percent performance degradation for state saving and rollback operations, even for large state vectors (several megabytes) and frequent state saving (every 100 microseconds) and rollback (every millisecond). We anticipate the rollback chip will allow parallel programs to exploit the advantages of Time Warp while avoiding most of the associated overheads.

References

[1] M. A. Franklin, D. F. Wann, and K. F. Wong. Parallel Machines and Algorithms for Discrete-event Simulation. *Proceedings of the 1984 International Conference on Parallel Processing*, 449–458, August 1984.

[2] R. M. Fujimoto. Performance Measurements of Distributed Simulation Programs. In *Distributed Simulation, 1988*, pages 14–20, Society for Computer Simulation, February 1988.

[3] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[4] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[5] D. Jefferson et al. *Distributed Simulation and the Time Warp Operating System.* Technical Report, Computer Science Dept., UCLA, August 1987.

[6] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[7] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing. Computer Science*, MIT Press, 1987.

[8] D. R. Jefferson and A. Motro. *The Time Warp Mechanism for Database Concurrency Control.* Technical Report U.S.C. Tech Report, Dept. of Computer Science, University of Southern California, Los Angeles, California, June 1983.

[9] J. Cleary, B. Unger, and X. Li. A Distributed AND-Parallel Backtracking Algorithm Using Virtual Time. In *Distributed Simulation, 1988*, pages 177–182, Society for Computer Simulation, February 1988.