Proceedings of the
18th Fault-Tolerant Computing Symposium
Tokyo, Japan, pp. 234-239, June 1988.

THE IMPLEMENTATION AND APPLICATION OF MICRO ROLLBACK IN FAULT-TOLERANT VLSI SYSTEMS†

Yuval Tamir, Marc Tremblay, and David A. Rennels

Computer Science Department
University of California, Los Angeles, California 90024
U.S.A.

ABSTRACT

Concurrent error detection and confinement requires checking the outputs of every module in the system during every clock cycle. This is usually accomplished by checkers and isolation circuits in the communication paths from each module to the rest of the system. This additional circuitry reduces system performance. We present a technique, called *micro rollback*, which allows most of the performance penalty for concurrent error detection to be eliminated. Detection is performed in *parallel* with the transmission of information between modules, thus removing the delay for detection from the critical path. Erroneous information may thus reach its destination module several clock cycles before an error indication. Operations performed on this erroneous information are ''undone'' using a hardware mechanism for fast rollback of a few cycles. We discuss the implementation of a VLSI processor capable of micro rollback as well as several critical issues related to its use in a complete system.

I. Introduction

One of the keys to achieving a high degree of fault tolerance is the ability to detect errors immediately after they occur and prevent erroneous information from spreading throughout the system. In order to achieve concurrent error detection and confine the damage caused by the error to the failed module, it is often necessary to check the outputs of the module during every clock cycle. These requirements are usually satisfied by connecting checkers and isolation circuits in the communication path from each module to the rest of the system. As a result, either the clock cycle time must be increased to allow the checks to complete or additional pipeline stages are added to the system, thus slowing down the system whenever the pipeline needs to be flushed or is not full due to data dependencies. Hence, systems with high-coverage concurrent error detection often experience significant performance penalties due to checking delays. This problem is especially important in VLSI implementations, where checkers often contain many series stages, and may introduce as much or more delay than the circuit being checked. These delays can compound as, for example, when a processor reads a memory word, and (1) Hamming Code checks are made, (2) the word is encoded for bus transmission, and (3) the word is checked when it arrives at the processor before use.

† This research is supported by Hughes Aircraft Company and the State of California MICRO program. The second author is partially supported by the Natural Sciences and Engineering Research Council of Canada.

One way to solve the problem described above is to perform checking *in parallel* with the transmission of information between modules. The receiving module does not wait for checks to complete. It proceeds with execution as the check is being carried out and the checking result is sent one (or a few) cycles later.

Performing error checking in parallel with inter-module communication largely solves the problem of checking delays, but it introduces a new problem in recovery. The state of the system (starting with the receiving module) may be polluted with damaged information before an error signal arrives. Therefore it is necessary to back up processing to the state that existed just before the error first occurred. This returns the system to an error-free state where the offending operation can be retried (or correction may be attempted by other means such as restoring information from a redundant module or initiating higher level rollback). We call the process of backing up a system several cycles in response to a delayed error signal *micro rollback*. In order to support micro rollback, each module in the system must buffer all the information necessary to undo the state changes that have occurred in the last few cycles.

Micro rollback differs from single-instruction retry [2] in that it occurs at a lower level - on the basis of clock cycles rather than instructions. Since the system undoes cycles rather than instructions, it can be done at the logic level without keeping track of microprogram-level instruction semantics and instruction pipeline conditions. As a result, the micro rollback capability can be independently implemented in each module of a synchronous system following very simple specifications. Moreover, our work is specifically aimed at efficient VLSI implementation techniques for supporting micro rollback.

This paper discusses the micro architecture and VLSI implementation of a VLSI RISC processor that is capable of micro rollback. We show how the updated state of the entire processor can be checkpointed after every cycle without replicating all the storage. The VLSI implementation of the basic building blocks needed for micro rollback is discussed. It is shown that the micro rollback functionality can be added with only a small performance penalty and with a low area penalty relative to the size of the entire chip. We show how the concept of micro rollback can be used throughout the system, discuss the requirements from modules other than the processor, and show how the various modules operate in a multiprocessor system.

II. Micro Rollback

A micro rollback of a subsystem (module) consists of bringing the subsystem back a few cycles to a *state* reached in the past. It is thus necessary to save the state of the subsystem (*checkpoint*) at each cycle boundary [3]. If the ''subsystem'' is a processor, the *state* is the contents of all storage elements which carry useful information across cycle boundaries. It is composed of the program counter, the program status word, the instruction register, and the register file, It also includes the contents of some pipeline latches and some registers in the state machine which can be changed during the execution of a multicycle instruction.

Since instructions also modify external memory (*loads* and *stores*), the state of the cache must also be preserved. A rollback restores the contents of the cache to its state a few cycles earlier. We discuss the interaction between the processor and the cache in section IV.

III. Support for Micro Rollback in a VLSI RISC

As part of our investigation of the cost of hardware support for micro rollback, we are designing and implementing a VLSI processor capable of micro rollback. We chose to start with the Berkeley RISC II processor [4] and determine the area overhead and performance penalty for adding to it the ability to perform micro rollback. The process of saving the state of a RISC processor and the method used to rollback in one cycle are described below. The state to be saved and restored is located in the register file and the individual state registers.

A. Micro Rollback of the Register File

At every cycle, there is a possibility that a *write* into the register file occurs. It is impractical to preserve the state of the file for N cycles by replicating it N times (for example, using shift registers) due to the large area occupied by even one copy of the on-chip register file (40% of chip area in RISC II) [4]. We propose an alternative method which minimizes the extra hardware while still allowing a rollback of up to N cycles to be executed in one cycle.

High-level Description. Whenever the processor writes data to one of its registers, the full address of the destination register as well as the data to be written, is stored in an N-word FIFO queue (see Figure 1). The part which holds the address of the register to be written is associative while the part containing the data, is composed of memory cells which can also be shifted. During the register read phase of every instruction the register addresses of the two operands are compared with the addresses of the registers stored in the FIFO/CAM. If there is a match, the data of the matching register is put on the corresponding internal data bus. If there is more than one match for a particular operand, a priority circuit is used to provide the most recent version available in the FIFO/CAM. This corresponds to the rightmost valid register in the FIFO in Figure 1.

The FIFO acts as a buffer which delays each *write* by N cycles before it is written into the register file. During every cycle an entry is made in the FIFO. If a write occurs, the data is entered and the FIFO position is marked *valid*. If no write occurs, the FIFO word is reserved but marked as *invalid*. During every cycle, the oldest (leftmost) entry in the FIFO/CAM is written to its corresponding address in the register file if its valid bit is set, and discarded otherwise. In order to roll back p cycles ($1 \le p \le N$), the last p entries in the FIFO (the right-most p entries in Figure 1) are invalidated.

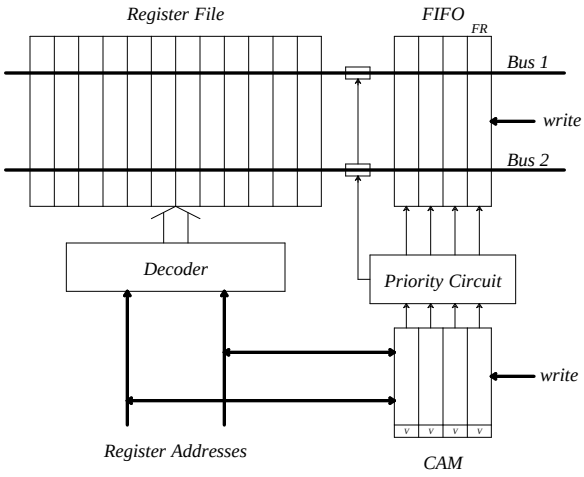

Figure1: Rollback of Register File

An important feature of this design is that the storage of a new entry into the FIFO and the movement of an old word from the FIFO to the register file can take place simultaneously. Since there are no conflicts in accessing the buses, the primary additional delay in using this structure is the associative lookup operation, which can be made quite small.

The Standard Register File. As in RISC II, the datapath includes a large register file consisting of 128 32-bit registers, organized in eight overlapping register banks. The basic ram cell used in this register file is a two-port cell which allows two simultaneous *reads* and one *write* during a processor cycle. Both buses are precharged prior to a *read*. During a *write* the buses are loaded with complementary values to force the chosen cell to be overwritten. The decoders, based on a design made by Bill Barnard from University of Washington, use dynamic CMOS logic for fast decoding of the 7-bits register addresses.

The FIFO/CAM. The top section of the FIFO/CAM contains the data to be written into the register file, while the bottom part contains the register addresses of the corresponding data. The data part is a FIFO which is also a RAM so that each register, in addition of shifting to the left, is also accessible from the bus. The bottom part is a FIFO which is also a CAM. Each FIFO/CAM cell consists of a one-bit static shift-register cell as well as circuitry for associative lookup. Since an instruction may require two operands, two lookups in the CAM can be performed simultaneously. As a result of the associative lookup, one or more ''match lines'' are asserted and the most recent one is determined by the priority circuit and used to address the top section of the FIFO/CAM.

Data Buses and Connectivity Logic. The data bus through the standard register file is connected, through switching logic, to another bus which goes through the FIFO. Precharging the bus and selecting the proper register is done in parallel for the register file and the FIFO/CAM. The outcome of the lookup in the CAM determines whether the buses are connected or disconnected. If there is no match, the register file provides the data, while the FIFO/CAM takes over if there is a match.

Pipeline Organization. In RISC II, instructions are executed in a three-stage pipeline: instruction fetch, execute (includes reading from the register file), and write (store the result in the register file) [4]. The separate stage for writing to the register file is needed because writing a value to a large register file is a time consuming task and the separate stage for it facilitates most efficient use of processor resources. In order to prevent pipeline interlocks, RISC II uses *internal forwarding*: the result of the operation is stored in a temporary forward register at the end of the second pipeline stage and, if necessary, it is immediately (in the next cycle) forwarded to the next instruction. In the third stage of the pipeline the value from the forward register is written to the register file.

With our register file organization, writing the result can be done very quickly since it is never written directly into the large register file. Instead, the result is written to the first (right-most in Figure 1) entry in the FIFO/CAM. The entire FIFO/CAM serves a function similar to that of the forward register in RISC II. If we consider an instruction to be complete only when its result is written into the real register file, and given a FIFO/CAM with n stages, a processor using our register file organization has an $n+2$ stage pipeline: instruction fetch, execute, $n-1$ stages of advancing through the FIFO/CAM, and writing the result into the real register file.

B. Implementation of the Register File

The *read* and *write* delays when accessing a large register file are often critical factors in determining the overall processor timing. In a RISC processor, where the control is very simple, the register file is likely to be the largest single module on the chip [4]. In this subsection we present the details of the timing of the RISC processor with micro rollback support, focusing on the timing of the register file. The performance penalty and area overhead for micro rollback support in the register file are discussed.

Timing. Processor timing is based on a four-phase clock. The internal buses are precharges during ϕ_1 (phase 1), the registers are read during ϕ_2, the ALU operates and writes the results into the FIFO/CAM during ϕ_3 and ϕ_4. During ϕ_3 and ϕ_4 the FIFO/CAM is also shifted and the oldest entry is written into the register file [6]. With three micron CMOS implementation, a register file of 128 registers, and a FIFO/CAM with four entries, we can achieve a cycle time of 100 nsec, where ϕ_1 and ϕ_3 are each 15 nsec long while ϕ_2 and ϕ_4 are each 35 nsec long.

To determine the performance penalty for micro rollback support in the register file we have produced a complete VLSI layout of several register file and FIFO/CAM sizes and simulated the operation of these circuits using SPICE. The results of our simulations are shown in Figure 2. The overhead introduced for micro rollback support can be determined by comparing the read delay with a FIFO/CAM of the size of interest to the read delay with a FIFO/CAM of size 0 (no micro rollback support). For example, a file of 128 registers with a FIFO of 8 registers has a *read* delay of 34 ns (vs. 31 ns for a standard register file).

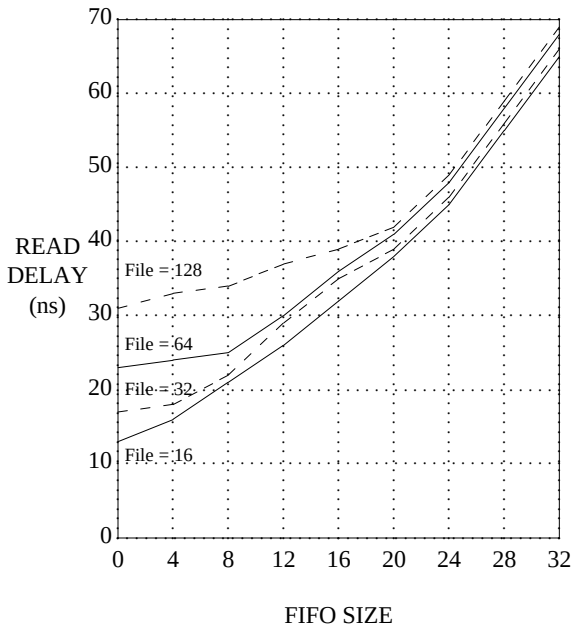

Figure 2: Read Delay vs File and FIFO Sizes

The *read* delay can originate from two different sources:
(1) when there is no match: the address is sent to the register file, the chosen register discharges the file bus lines, a superbuffer connected to the register file bus discharges the FIFO bus lines.
(2) when there is a match: the address is looked up in the CAM, a match is detected, the priority circuit determines which register to select, and the chosen FIFO register discharges the FIFO bus lines.
For a large register file the critical path will most likely be path (1) (for small FIFO). For small register files with relatively large FIFO, the main data bus is discharged before the buses are connected. In this case the critical path is path (2).

Area Overhead. Using the same combination of file and FIFO sizes, the area overhead required for micro rollback support in the register file is shown in Figure 3. Note that this overhead includes all the circuitry involved with the FIFO/CAM: the storage cells in the FIFO/RAM, the CAM cells, the priority circuit, the control, etc.

C. Detection and Correction of Errors in the Register File

A large register file is the module most likely to fail in a VLSI processor (especially due to transient faults). If an error occurs in one of the registers in the file, it is necessary to have

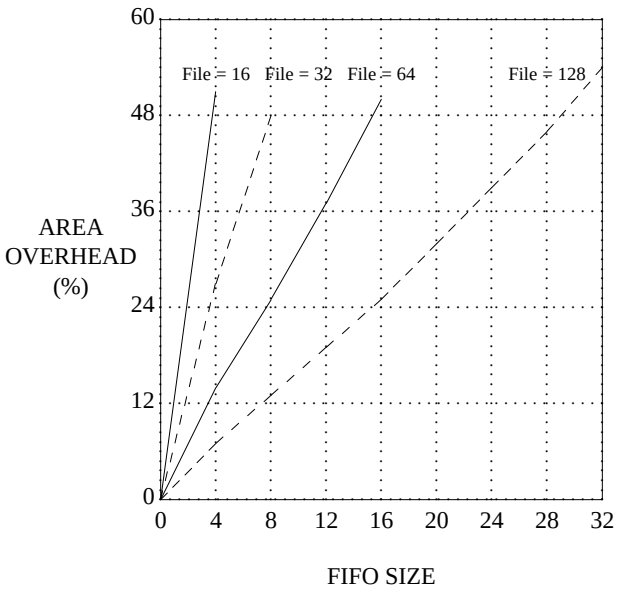

Figure 3: Area Overhead vs File and FIFO sizes

redundancy in order to recover. One approach is based on the use of a duplex processor from which valid copies of damaged information may be recovered. An alternative approach is to use error correcting codes in the register file. In both cases micro rollback prevents the checking or correction delays from slowing down normal operation.

Duplex Processors. This approach assumes that there are two processors executing the same instructions in lock-step. A separable error detecting code (e.g. parity) is used in the register file of each processor so that it will be possible to determine which processor has the valid data. If a processor detects an error in a value read from the register file, it initiates a micro rollback in both processors to the cycle prior to the register file access. The correct data is then copied from the second processor to the one that detected the error and normal operation is resumed.

Use of Error Correcting Codes. If some separable error correcting code (e.g. Hamming) is used in the register file, it is possible to recover from errors in the register file without the use of a duplex processor. The code bits are generated when a word is transferred to the FIFO/CAM. During *reads* from the register file, the correction circuitry detects and corrects them appropriately. Since it is undesirable to lengthen the critical timing path, words read out of the register file are captured by latches and checked in parallel with their processing by the ALU. If an error is detected, processing is stopped, the word is then corrected and stored in the regular register file. A micro rollback is then initiated in order to restore the state of the processor to the cycle prior to the one when the register file read was initiated.

D. Micro Rollback of Individual State Registers

In various locations all over the chip there are individual registers that contain part of the processor state. These include the Program Counter, Instruction Register, Program Status Word, Pipeline Latches, etc. Since these registers are not in one location support for micro rollback of up to *N* cycles means that for each register there must be *N* ‘‘backup registers’’ to store its state in the previous *N* cycles. While organizing the backup registers as a stack is possible, such an organization would result in slow rollback. In order to achieve rollback in one cycle, regardless of how many cycles are being rolled back, each set of *N* backup registers, is organized as a small RAM. (see figure 4). The input to the address decoder of this RAM is the same throughout the chip. This value is a pointer the current active backup register. During each cycle, the contents of the current state register are written into the active backup register in the RAM and the pointer is incremented. To restore the state during a rollback, the appropriate backup register is loaded into the current register.

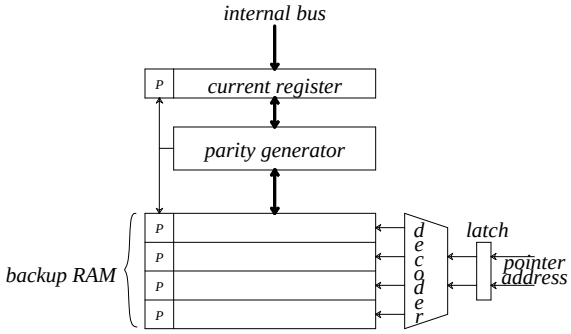

Figure 4: Saving of a State Register

A potential problem with the scheme is if micro rollback is initiated and the version of the individual register read from the backup RAM is erroneous (due to a transient fault that occurred after the value was stored in the backup RAM). Detection of such errors can be supported by adding error detection bits to the value of the register when it is stored in the backup RAM (e.g. parity as in Figure 4). During a rollback, the contents of the appropriate backup register goes through the parity checker and is stored into the current state register. If an error is detected, a micro rollback on one additional cycle can be initiated, thus recovering from this potentially serious error.

The extra hardware for micro rollback support in individual registers has only a small effect on processor speed since the extra logic is not connected serially to the path followed by the buses; only the one ‘‘current register’’ interacts with the rest of the system. On the other hand the area for each state register must be increased. For example, in order to implement the capability of rolling back four cycles the area must be increased by about a factor of six.

IV. System Issues in the use of Micro rollback

In addition to the processor, micro rollback can be used effectively with other modules of the system. Parallel error checking and delayed error signals can be implemented using techniques quite similar to those used in the processor. The decision to use these techniques instead of serial checks is based on the potential performance improvement and the hardware overhead involved. In general, whenever data can be

received or transmitted before it is checked, a FIFO/CAM, such as the one described in Section III, may be used to delay permanent modification of critical state until the results of the check are received.

A. Micro Rollback and Cache Memory

Most modern machines use caches to increase performance by decreasing the effective memory access time. The introduction of serial checking in the path between the processor and the cache introduces a severe degradation of the benefits gained by the cache. In this case, parallel checking means a significant performance gain.

Since cache resembles a register file, a FIFO/CAM can be used as in a register file to support micro rollback. During a *load*, data comes either directly from the cache on a *hit*, or from main memory on a *miss*. On a hit, the state of the cache is unaffected and no actions need to be taken to undo the *load*. On a miss, a line in the cache has to be replaced by data fetched from main memory. During a micro rollback, it is impractical to restore the contents of a line in the cache with its previous contents. Instead, we leave the line intact knowing that the worst that can happen is that a line has been fetched unnecessarily.

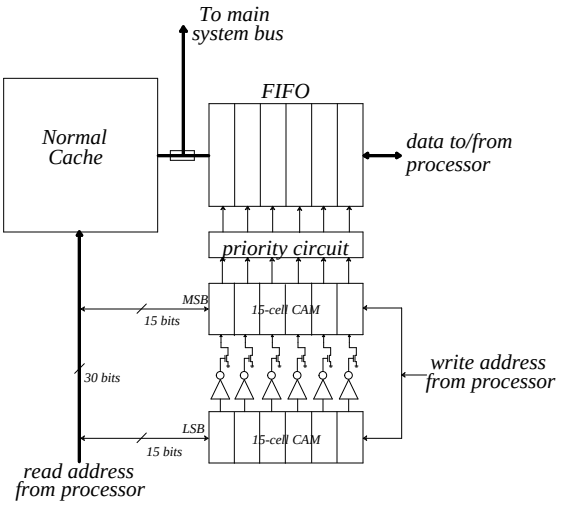

Figure 5: A Cache with Support for Micro Rollback

A *store* instruction requires additional hardware and is handled in the same way as a *write* for the register file (see Figure 5). The *store* is delayed for N cycles in a FIFO/CAM. For every load instruction there is a lookup in the CAM to ensure that the processor will obtain the most recent value stored to the specified address. Entries in the FIFO/CAM can be marked *invalid* in order to cancel recent stores, thus performing micro rollback. It should be noted that the interaction between the cache and main memory occurs after the data has gone through the FIFO/CAM. Thus, either write-back or write-through caches may be used and in both cases there will never be a need to undo a write to main memory (checking of the data is complete before the write to main memory occurs).

Figure 5 shows that the comparison in the CAM is now made in two sections of 15 bits, forming the real address, instead of the 7 bits register addresses inside the processor. This comparison takes 22 ns, instead of 8 ns for the register file. The lookup in the FIFO/CAM is performed in parallel with the lookup in the cache. Since the access time of the cache in current VLSI systems is typically longer than 22 ns, the lookup time in the FIFO/CAM will not degrade system performance.

B. Micro Rollback and Main Memory

In many systems it may not be desirable to include main memory in micro rollbacks. One of the problems with performing micro rollback in all modules of a large multi-module system is that it requires synchronous operation. For micro rollback of up to N cycles, each module must buffer up to N ''versions'' of its state and be capable of precise roll back of a specified number of cycles. This is difficult or impossible to do if the entire system is not completely synchronous. Many high-performance buses have asynchronous protocols so it may be difficult to coordinate micro rollbacks of the processor and main memory.

If the frequency of interactions (communication) between two particular modules in a system is small, the receiving module can simply wait for the error checks to be completed before using the data. Under these conditions the effect of waiting on performance is small and there is no justification for the extra hardware and added system complexity for micro rollback support. This argument often holds for main memory and I/O interface units in systems in which the processor uses a cache. In such systems the processor ''rarely'' accesses main memory. In addition, since communication accross the system bus typically occurs in multiple word blocks, performing the checks serially, in a pipeline fashion, only delays the first word of the block and causes only minimal performance degradation.

V. Micro Rollbacks in a Multiprocessors Environment

So far we have demonstrated how micro rollbacks work in a single processor system. We now describe how micro rollbacks can be used with a shared-memory multiprocessor system. The architectural model considered is the following: a collection of processors, each with its own private cache, are connected to each other as well as to main memory through a single shared system bus.

When a *write* is executed by a processor, the local cache and the rest of the system are normally not aware of it until the data exits the FIFO and is written into the local cache (see Figure 5). Any actions required by the cache coherency protocol then takes place as with normal caches.

In a multiprocessor system where each processor has a local cache there is a problem of maintaining identical views of the logically shared memory from all the processors [1]. Specifically, in order for the caches to be transparent to the software, the system is often required to be *memory coherent*, i.e., the memory system is required to ensure that ''the value returned on a *load* is always the value given by the latest *store* instruction with the same address'' [1]. A multiprocessor

system in which our FIFO/CAMs are used with the caches is *not* memory coherent. Specifically, a *load* executed by one processor cannot return the latest value written to the address by a *store* from another processor until the value stored ''propagates'' to the head of the FIFO/CAM and is written to the cache.

In a multiprocessor system that is not memory coherent it is desirable to maintain the weaker condition of *sequential consistency* [5]. Sequential consistency requires that ''the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.'' Unfortunately, a multiprocessor system that uses our FIFO/CAMs is *not* sequentially consistent. The problem is illustrated by Lamport's [5] example of a mutual exclusion protocol:

```
Process 1: a := 1 ;
           if  b = 0  then critical section ;
                a := 0
           else  . . .  fi
Process 2: b := 1 ;
           if  a = 0  then critical section ;
                b := 0
           else  . . .  fi
```

With our FIFO/CAMs, Process 1 can set a:=1 at the same time that Process 2 sets b:=1. They can then both reach their *if* statements before the *stores* setting *a* and *b* have time to propagate to their respective caches. Sequential consistency is violated since the result of the execution is as though the sequence of operations is:

```
Process 1: if  b = 0  then critical section ;
Process 2: if  a = 0  then critical section ;
Process 1: a := 1 ;
Process 2: b := 1 ;
```

Without modifications to the scheme shown in Figure 5, a multiprocessor in which the processors use the FIFO/CAMs with their local caches is a very limited system. Indeed synchronization through atomic instructions such as *test-and-set* can also become a problem. For a *test-and-set* instruction, the *set* is made right after the *test* during the same bus transaction. Using a conventional processor with the cache described in Figure 5, the *test* consists of reading the variable from the cache, while the *set* involves a *write* to the FIFO. A problem occurs if processor P_1 performs a *test-and-set* and the *write* is still in its FIFO when processor P_2 performs another *test-and-set* on the same variable. Processor P_2 will not be able to observe the *set* by P_1 and both processors will enter the critical section concurrently. As shown above, because the system is also not sequentially consistent, other mutual exclusion protocols that work on conventional multiprocessors may not work on a multiprocessor that uses our FIFO/CAMs.

In order to allow synchronization in a multiprocessor with our FIFO/CAMs, the cache controller must be modified. The simplest modification is to allow cache blocks to be locked during test-and-set operations. Specifically, a block containing a semaphore should be locked from the time it is accessed, until the time that it is modified. For a FIFO/CAM of depth N, this means that the block is inaccessible during N cycles plus the time it takes to execute the *test-and-set* instruction. Alternative solutions are currently under investigation.

VI. Summary and Conclusions

The delays due to error checking being performed in series with intermodule communication are one of the primary causes of performance degradation associated with implementing concurrent error detection in VLSI systems. The circuitry required to perform such checks as parity, residue codes, and m-out-of-n codes often reduce the speed of the circuit by 50-100%. Checks that are area efficient are usually serial and slow, and faster checks are often trees which take up large area, thus slowing down surrounding circuits. Checking delays are compounded when data is transferred between several modules, and checked at several places.

This fundamental problem in achieving fault tolerance in high performance VLSI systems can be overcome by performing checks on the data in parallel with intermodule communication. Data to be checked can be latched, and error checking can take place in one or more subsequent cycles (as a pipeline). As a consequence error signals can arrive one or more cycles after error-damaged data is received for processing. In this paper, we have described a mechanism, called *micro rollback*, which allows checking to proceed in parallel with communication by supporting fast rollback of a few cycles when a delayed error signal arrives. We presented a systematic way to design VLSI computer modules that can roll back and restore the state which existed when the error occurred. When applied to a VLSI RISC processor, this technique is characterized by extremely low performance overhead and a modest area overhead compared to the area of the entire processor. The recoverable register file we designed is a new critical building block which is particularly well-suited to VLSI implementations and is likely to have wide application in future systems that will combine fault-tolerance and high performance.

References

1. Lucien M. Censier and Paul Feautrier, ''A New Solution to Coherence Problems in Multicache Systems,'' *IEEE Transactions on Computers* **C-27**(12), pp. 1112-1118 (December 1978).
2. M. L. Ciacelli, ''Fault Handling on the IBM 4341 Processor,'' *11th Fault-Tolerant Computing Symposium*, Portland, Maine, pp. 9-12 (June 1981).
3. W. W. Hwu and Y. N. Patt, ''Checkpoint Repair for Out-of-order Execution Machines,'' *14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 18-26 (June 1987).
4. M. G. H. Katevenis, ''Reduced Instruction Set Computer Architectures for VLSI,'' CS Division Report No. UCB/CSD 83/141, University of California, Berkeley, CA (October 1983).
5. L. Lamport, " ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs",'' *IEEE Transactions on Computers* **C-28**(9), pp. 690-691 (September 1979).
6. Yuval Tamir, Marc Tremblay, and David A. Rennels, ''The Implementation and Application of Micro Rollbacks in Fault-Tolerant VLSI Systems,'' Computer Science Department Technical Report CSD-880004, University of California, Los Angeles, CA (January 1988).