THE IMPLEMENTATION AND APPLICATION OF MICRO ROLLBACKS IN FAULT TOLERANT VLSI SYSTEMS

Yuval Tamir
Marc Tremblay
David A. Rennels

January 1988
CSD-880004

The Implementation and Application of Micro Rollbacks in Fault-Tolerant VLSI Systems

Yuval Tamir, Marc Tremblay, and David A. Rennels

Computer Science Department

4731 Boelter Hall

University of California

Los Angeles, California 90024-1596

U.S.A.

Phone: (213)825-4033   E-mail: tamir@cs.ucla.edu

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 rollbacks*, for eliminating most of the performance penalty for concurrent error detection. 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 rollbacks, its use in a complete system, and the use of micro rollbacks in a fault tolerant multiprocessor.

Keywords: rollback, VLSI systems, building blocks, fault tolerant architectures.

January 1988

1. 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 aggravated in VLSI implementations where checkers often stretch over long lengths, contain many series stages, and introduce as much or more delay than the circuit being checked. These delays can compound as when a memory word is read 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.

One way to solve the problem described above is to perform checking in *parallel* with the transmission of information between modules. The machine 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 largely solves the problem of checking delays, but it introduces a new problem in recovery. The state of the computer may have been polluted with damaged information before the error signal arrives. Therefore it is necessary to back up processing to the state that existed when error first occurred. This returns the computer 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 a program rollback). We will refer to the process of backing up a computer several cycles in response to a delayed error signal as *micro rollback*. This involves buffering changes of state for several cycles in each module of a computer and being able to return to a previous state. Micro rollback differs from single-instruction retry[3] in that rollback 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. Moreover, this work is specifically aimed at VLSI implementation and

specifically the problem of delayed error checks.

This paper discusses the micro architecture and VLSI implementation of a VLSI RISC processor that is capable of micro rollbacks. 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 to implement micro rollbacks 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 describe how micro rollbacks can be used with parallel (delayed) error checking to provide error recovery using either ECC in the processor or restoration of state from an external processor. 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.

2. Micro Rollbacks

A micro rollback consists of bringing the processor back a few cycles to a *state* reached in the past. It is thus necessary to save the state of the processor (*checkpoint*) at each cycle boundary.[4] The *state* of a processor 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, 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.

Because of the fact that instructions also modify external memory (*loads* and *stores*), the state of the cache must also be preserved. A rollback merely restores the contents of the cache present a few cycles ago. We discuss the interaction between the processor and the cache in section 4.

3. Implementation of Micro Rollbacks in a VLSI RISC Processor

The process of saving the state of a RISC processor and the method used to rollback in one cycle is described below. It basically consists of saving two different entities, the register file and the individual state registers.

3.1. Micro Rollback of the Register File

At every cycle, there is a possibility that a *write* into the register file occurs. To preserve the state of the file for N cycles by replicating N times (using for example shift registers) is unacceptable due to the large area occupied by an on-chip file (40% in RISC II). [5] We use a different method which minimizes hardware and still allows a rollback of up to N cycles in one cycle.

3.1.1. High-level Description

Whenever the processor writes data in 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. Every time there is a register-read operation, we compare the address of the two operands with the address of the registers stored in the associative part of the FIFO. 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 register matching the address of the operands, a priority circuit is used to provide the most recent version available in the FIFO. 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 each 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 occurred during the cycle the corresponding FIFO word is reserved but marked as *invalid*. Similarly during each cycle when the FIFO is full the oldest word is removed. Only if its valid bit is set will its contents be written to its corresponding address in the register file. In order to roll back p cycles ($1 \leq p \leq N$) we invalidate the last p entries in the FIFO (the right-most p entries in Figure 1) which may have contained from 0 to p valid writes.

A very 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. It turns out that there are never any conflicts for buses so that the primary additional delays in using this structure are only the associative lookup operations which can be made quite small. Design features are discussed below.

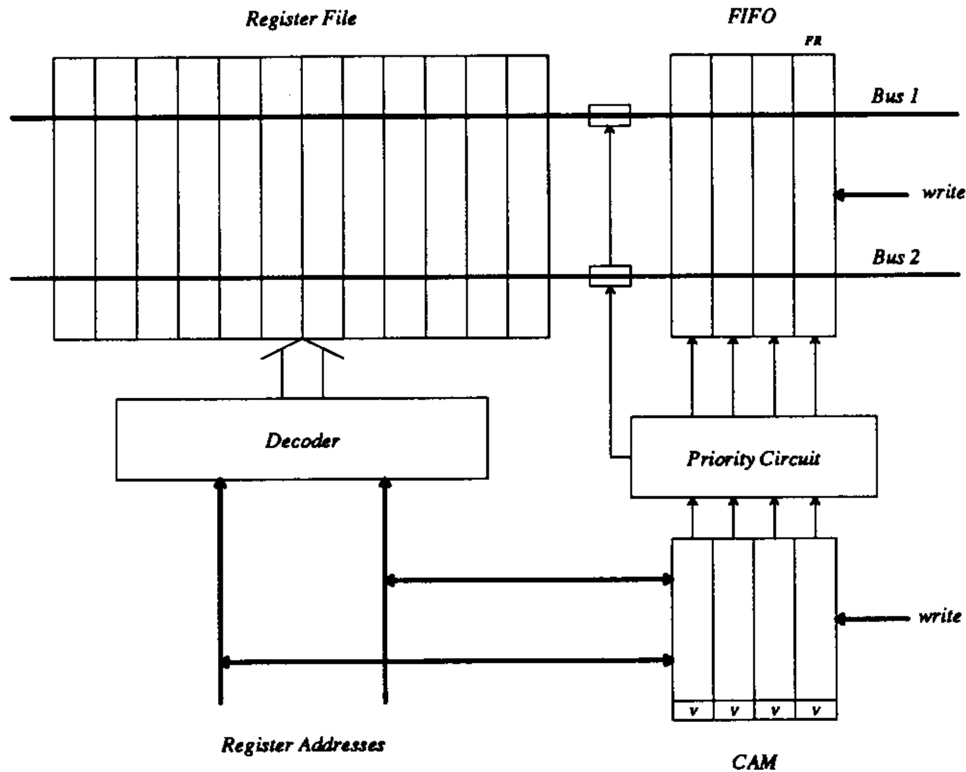

Figure 1: Block Diagram

3.1.2. Description of the RAM

Connected to the data path, is a large register file consisting of 128 32-bit registers. The basic ram cell is a two-port cell which allows two simultaneous *reads* and one *write* during a processor cycle. During a *read* both buses are precharged to speed up the delays. During a *write* the buses are loaded with complementary values to force the chosen cell to be overwritten.

The layout of the RAM cell is shown in Figure 2. The cell measures $1560\lambda^2$.

To improve the decoding of the 7 bit composing a register address, we use large pass transistors. With the help of dynamic CMOS logic, we were able to keep the size of the decoder relatively small and fairly fast. We based our decoder on a design made by Bill Barnard from University of Washington. Since two registers may be read simultaneously, two decoders are provided. One precharges a line which controls a PMOS transistor, while the other one predischarges an NMOS transistor. Figure 3 shows a transistor level description of the decoder.

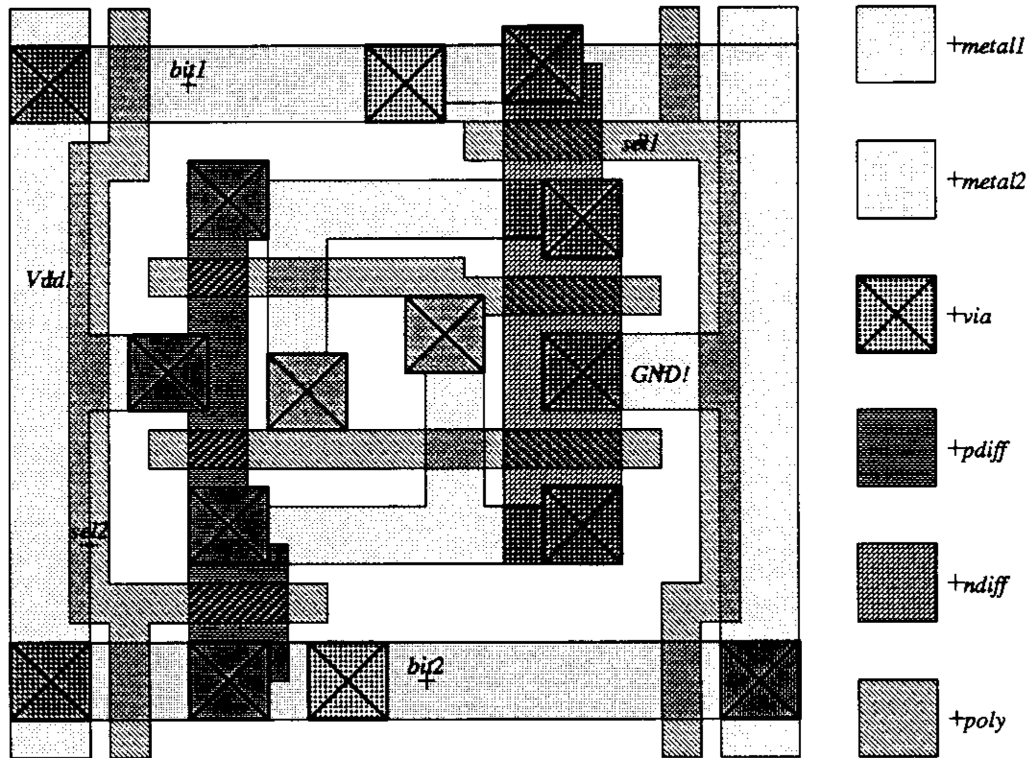

Figure 2: Layout of a RAM cell

3.1.3. Description of 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 address accompanying the data. The data part is a simple FIFO where each register, in addition of shifting to the left, must also be accessible from the bus. The bottom part is composed of basic CAM cells consisting of a one-bit static shift-register with some extra logic for the cell to behave like an associative memory. A loop consisting of two inverters and two pass transistors forms the storage part of the cell (see Figure 4). For the comparison with a current address bit, pass transistor logic is used to implement XOR gates. Two lines, Match1 and Match2, initially precharged, represent the status of the comparison. A one means that there is a match while a zero represents a mismatch. A mismatch will discharge the match line through two pass transistors. The speed of this discharge is critical and therefore appropriate transistor sizes must be calculated. Figure 5 shows the layout of the CAM cell.

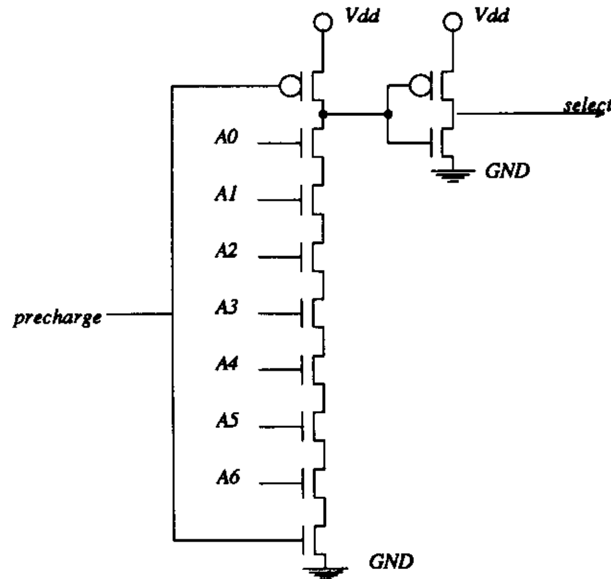

Figure 3: Transistor Diagram of the Decoder

3.1.4. Data buses and connectivity logic

The data bus going through the 128 registers is connected, with the help of some logic (see Figure 6), 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 for the FIFO. The outcome of the comparison performed in the CAM with the register addresses decides if the the buses are connected or disconnected. If there is no match, the register file provides the data, while the FIFO takes over if there is a match.

3.1.5. Forward Register

As explained earlier, instructions are executed in a three-stage pipeline; fetch, execute and write-back (see Figure 7). In order to avoid interlocks, we provide a latch which contains the result of the ALU operation at the end of cycle 2. This latch is called Forward Register (FR). It is necessary to use such a scheme to allow instructions such as:

add R1,R2,R3

add R3,R4,R5

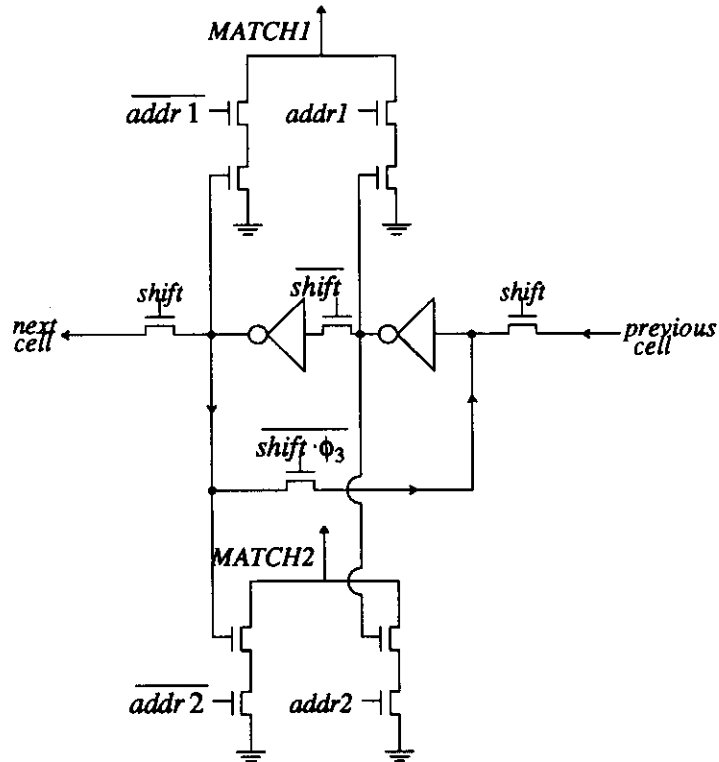

Figure 4: Logic of CAM cell

to be executed properly. In this case, the result of the ALU, which is written into the register file during cycle 3 would not be available for the next instruction. But using FR, which is written into at the end of cycle 2, the data is readily available for the next instruction. The physical location of the FR can be seen on the datapath described in Figure 8.

3.2. Timing and Metrics of the Register File

The register file is an important contributor to the critical path in the processor. Indeed the lengths of the different processor clock phases are mainly based on worst case delays from the file. We present the details of the four phase processor cycle.

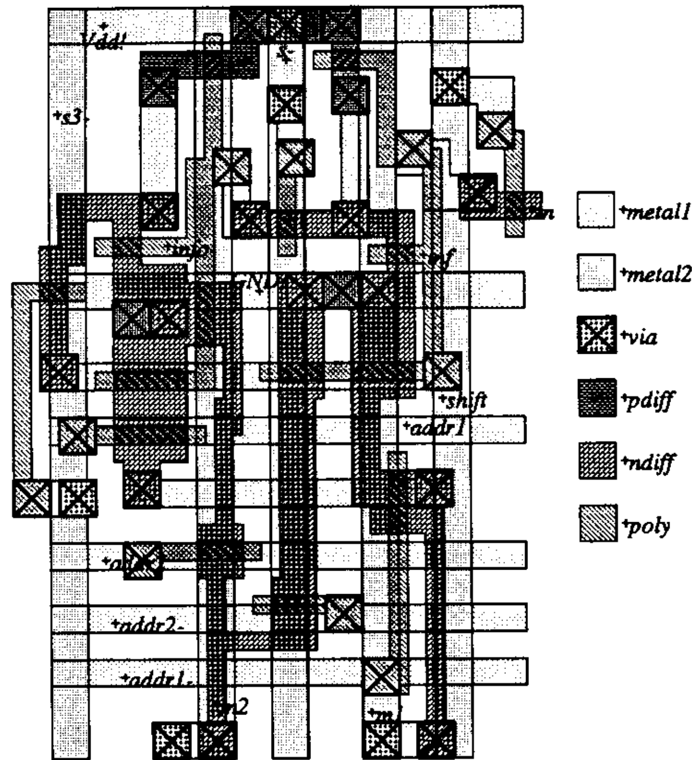

Figure 5: Layout of the CAM cell

3.2.1. Timing diagram

The following times are based on a 128 register file with a FIFO of depth 4. Our basic processor cycle is composed of four non-overlapping phases which altogether add up to a period of 100ns. Phase 1 (ϕ_1) and phase 3 (ϕ_3) are 15ns long while phase 2 (ϕ_2) and phase 4 (ϕ_4) are 35ns long (see Figure 9).

After we pre-discharge the decoder during the first part of ϕ_1, it takes about 9ns to fully decode the 7-bit register addresses. During that time the main bus in the register file is precharged and is ready to be discharged by the chosen cells. The read discharge takes approximately 33ns. This completes the first two phases. ϕ_3 and ϕ_4 are dedicated to the write operation. The diagram also shows the timing associated with the shifting in the FIFO/CAM.

The price to pay for adding all this logic to the the register file is mainly in terms of area. From our layout we measure an overhead, in terms of extra logic dedicated to the register file, of 10%.

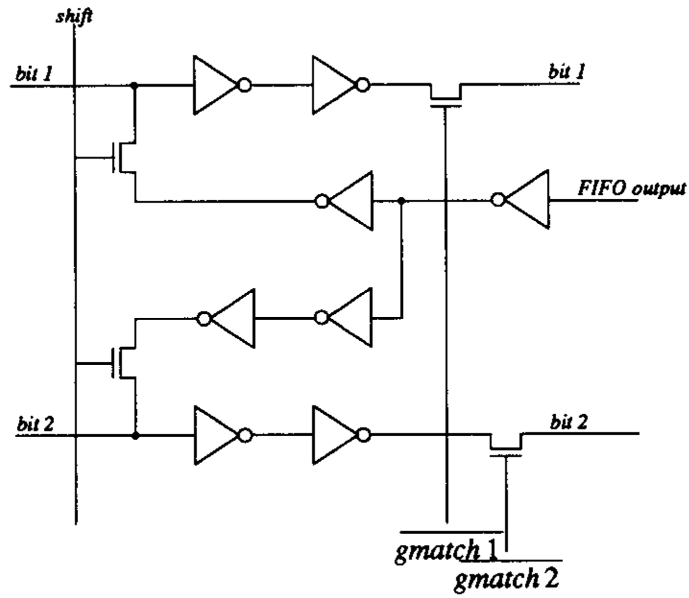

Figure 6: Connectivity Logic between RAM and FIFO

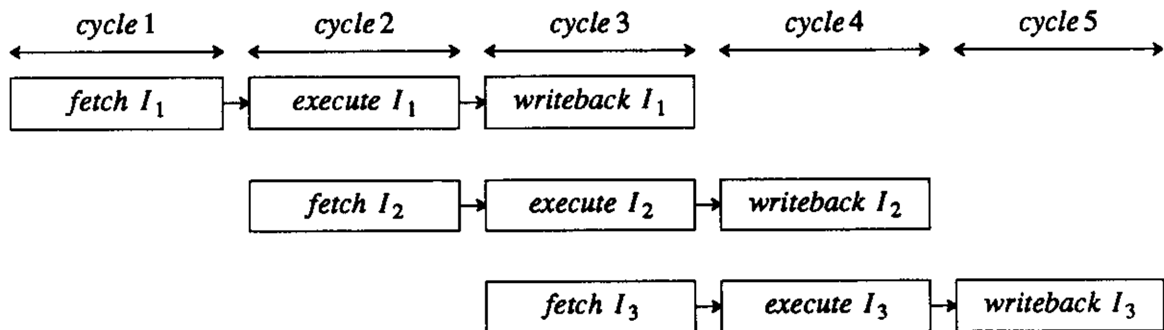

Figure 7: Instructions Pipeline

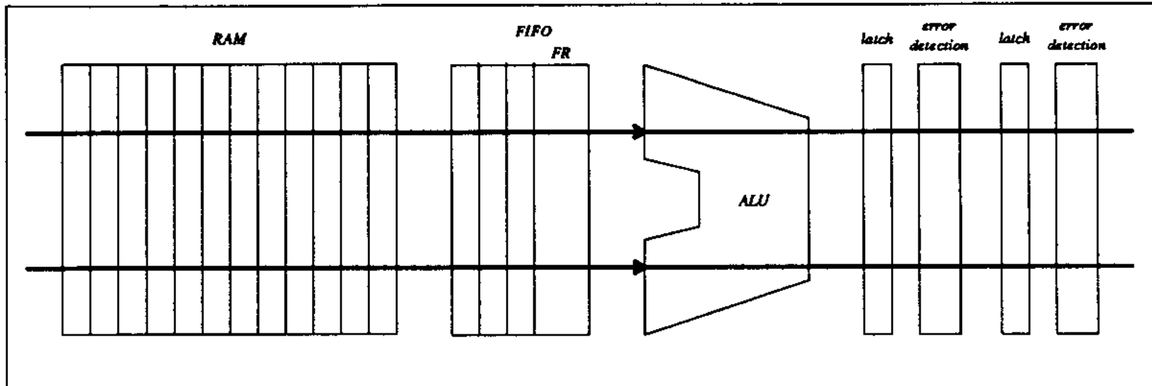

Figure 8: Datapath

3.2.2. READ Delay vs Scaling of the Design

We have simulate 4 different register file sizes (16, 32, 64, 128) with 8 different FIFO sizes (4, 8, 12, 16, 20, 24, 28, 32). Using Figure 10, a design team can decide what FIFO size is appropriate for a given register file. From the graphic one can measure the overhead introduced by the micro rollback by comparing it to a FIFO size of 0 (no micro rollback). For example, a file of 128 registers with a FIFO of 8 registers has a *read* delay of 34ns (vs 31 for a plain file).

The *read* delay calculated in the graphic 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 discharge the FIFO bus lines. Notice that the signal which connects both data buses comes during the mean time.
(2) when there is a match: the address goes through the CAM, a match is detected, the priority circuit detects which register to select, 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).

Figure 9: Timing Diagram

3.2.3. Area Overhead vs Scaling of the design

Using the same combination of file and FIFO sizes, we present in Figure 11, the area overhead required to add the micro rollback logic.

3.3. Protection of The Register File

If an error occurs in one of the registers in the file, it is necessary to have a redundancy in order to recover. One approach is to use a duplicated processor and recover damaged information from the duplex copy. The other approach is to use error correcting codes in the register file. The micro-rollback mechanism can be used to hide the checking delays associated with this since, if an error occurs, the processor can back up several cycles.

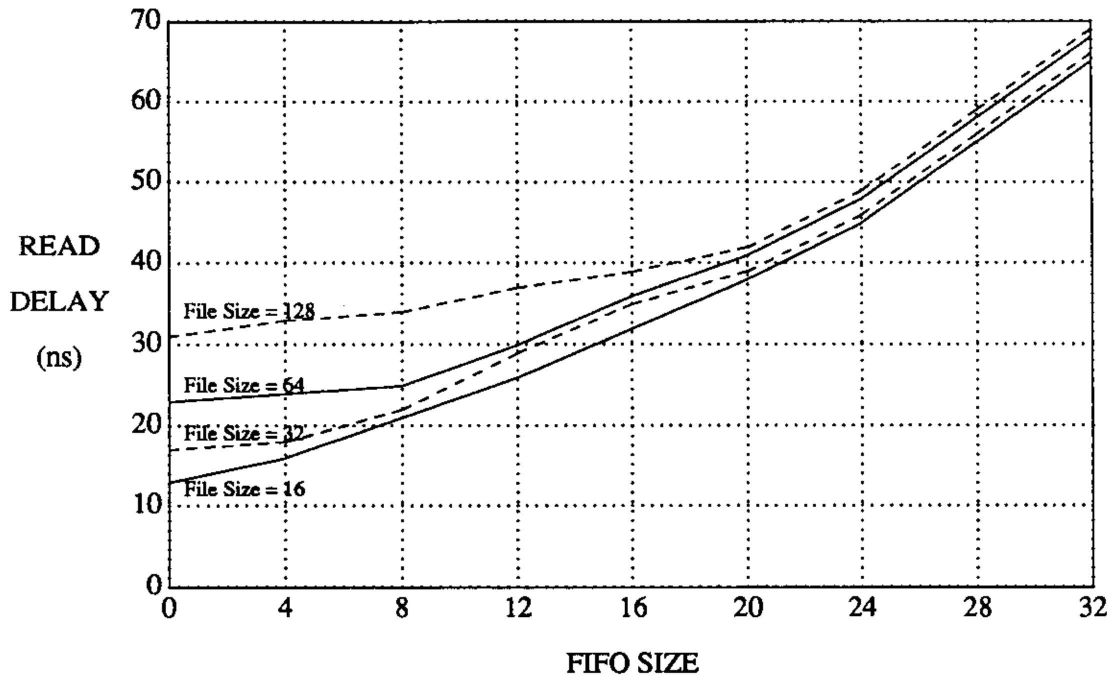

Figure 10: Read Delay vs File and FIFO Sizes

3.3.1. Detection from Parity and Correction by Processor Transfer

This approach assumes that there are two processors (P_1 and P_2) executing the same instructions in parallel. All the words in their respective register files are stored with an accompanying parity bit, inverse residue or similar separable code. Whenever *read* instructions are executed, checking of each word is performed. When an error is detected in the register file by P_1, it sends the address of the faulty register to P_2 and and both rollback to the point before the file access. Notice that P_2 will be requested to rollback more cycles because it kept executing during the time that P_1 sends the address. At this point P_2 transfers its correct data to P_1 and the operation continues.

3.3.2. Data Correction Based on Hamming Code

Using this approach, every word in the register file is encoded with an error correcting code (e.g. Hamming SEC-DED). The code bits are created when a word is transferred to the FIFO/CAM. During *reads* from the register file, the correction circuitry should detect errors and correct them appropriately. Of course we do not want to do this by lengthening the critical timing path. Therefore when words are

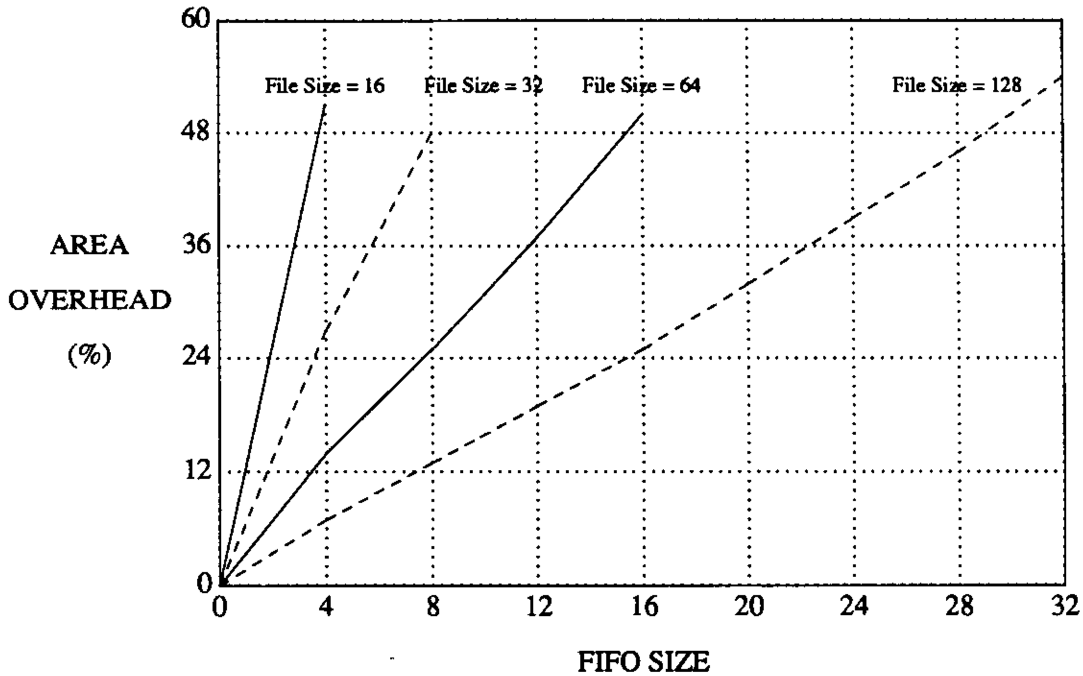

Figure 11: Area Overhead vs File and FIFO sizes

read out of the register file they are captured by latches and checked in parallel as processing is carried out in the ALU. If an error is detected processing is stopped, the word is corrected, and it is restored in the file. Then a micro rollback is issued in order to bring the processor to the cycle when a read was requested.

Based on published results[8] and on our experience, we believe that the method based on the duplication of the processors is more efficient in terms of implementations costs, performance, and fault coverage.

3.4. Micro Rollback of Individual State Registers

The state registers consist of the various individual registers which are typically at widely separated locations on the chip. These include the Program Counter, Instruction Register, Program Status Word, Pipeline Latches, etc. Here, each working register must be protected up with an additional N backup registers to assure that the state can be rolled back for up to N cycles. Each set of N backup registers,

(where N represents the maximum number of cycles that we may want to roll back to), is organized as a small RAM. (see figure 12). A decoder points to the active backup register in the small RAM, and a control signal indicates if a READ of a WRITE operation is needed.

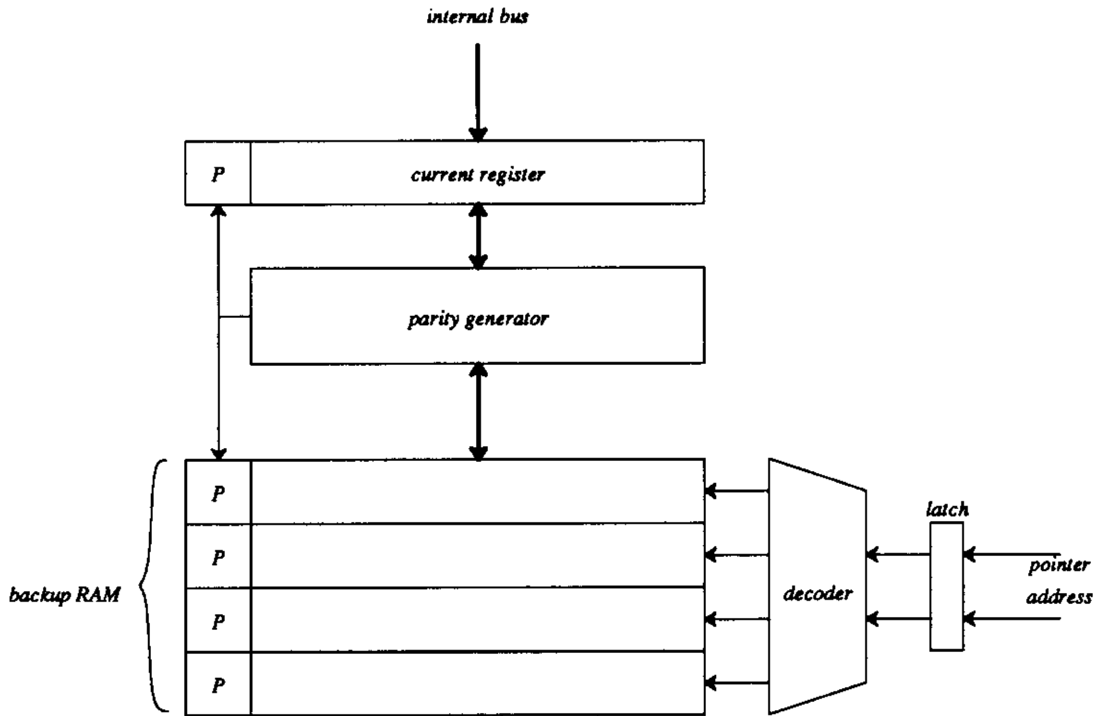

Figure 12: Saving of a State Register

During each cycle, if the current state register is modified, its contents are written into the small RAM. To restore the state during a rollback, the appropriate back-up register is loaded into the current register. In this way a roll back of 3 cycles is performed just as fast as one of 1 cycle, which would not the case if we were using a stack. The layout is quite compact and very regular as one can see from the transistor level diagram on Figure 13.

An error detecting code can be used to protect the states. If the word comes from external memory, its code bit(s) are brought into the chip. A check is performed on the word when the transfer from the current register to the backup RAM is done. If an error is detected, we rollback one cycle which actually *corrects* the state register by restoring its previous contents. During a rollback, the opposite transfer occurs. The contents of the appropriate backup register goes through the parity checker and is stored into

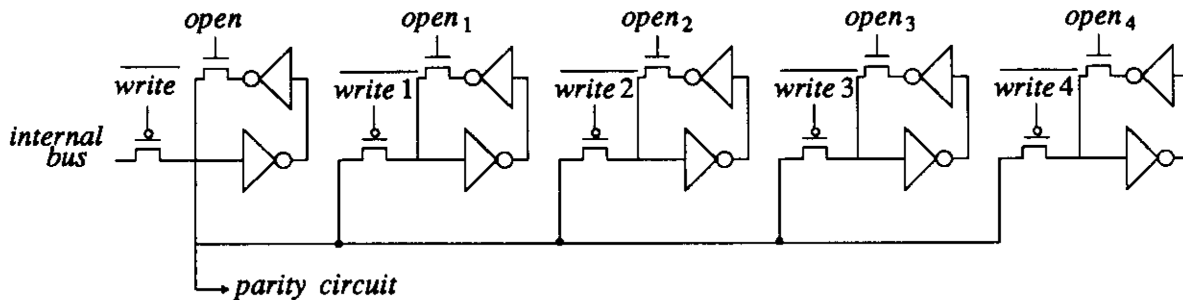

Figure 13: Transistor Logic of Backup RAM

the current state register. If an error is detected, once again we rollback one more cycle if the depth of the FIFO allows it.

The effect of the extra hardware on the processor speed is small because 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.

3.5. Examples

The general method described above applies to any single state register. There are singularities attached to some specific registers that are described below:

Program Counter (PC): Besides saving the state of PC, the backup RAM serves a dual purpose by saving LAST PC, which is very useful for restarting instructions during interrupts.

Instruction Register (IR): This register is really a set of a few smaller registers containing information spread out at different places in the layout (opcode, registers address, etc). Each individual part has its own back-up RAM. One tradeoff would be to group state registers together to decrease the total extra logic such as the decoder and the parity generator.

Another alternative concerning the instruction register is not to preserve it and refetch it from the

program counter. This eliminates some hardware but it also adds an extra cycle before any useful work can start. Since our method throughout the cpu and over the whole system aims at a one-cycle recovery, we chose the first option which favors performance over increased area.

Program Status Word (PSW): The program status word presents a peculiar situation. One or several individual bits can be changed at every cycle. Since the parity circuit cannot make the difference between a bit being changed by the control unit and a bit being flipped because of a fault, we use a different strategy. Duplication of the PSW together with a comparator circuit provides error detection and can generate a micro rollback.

Multistage Latches: Some of the latches involved in the pipeline are only required in the second or third stage. For example one bit of an instruction specifies if the PSW needs to be modified during the third stage. Instead of replicating the cascade latches we group them together under the same backup RAM and use two decoders to refer to the. READ and WRITE location in the RAM.

4. System Issues in the use of Micro rollback

In addition to the processor, micro rollbacks 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 those techniques compared to a serial check, is based upon performance improvement/overhead and the degree of simplicity/complexity involved. In general, whenever data can be received or transmitted before it is checked, a FIFO/CAM, such as the one described in Section 3, may be used to delay permanent modification of critical state until the results of the check are received.

4.1. Micro Rollback and Cache Memory

Most modern machines use an instruction and data cache to increase performance by decreasing the effective memory access time. For example the Fairchild Clipper[2] accesses the cache in 4 clock cycles compared to 9-10 for a zero wait state main memory. Error detection (using Hamming code) applied to 32-bit words coming from the cache takes approximately 50ns. In this way, a serial check would add two more cycles to a 33MHz Clipper, an overhead of 50% in the cache access time. Considering that the cache is active for every instruction and data access, it is a huge price to pay. A parallel checking is a

significant performance gain in this case.

The fact that a cache resembles a register file, suggests the use of the same method, i.e. a FIFO/CAM, to preserve data entry. Indeed, except for some minor modifications, the same method works and we verify it by looking at *load* and *store* instructions individually.

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 which can happen is that a line has been fetched for no reason.

A *store* instruction requires additional hardware and is handled in the same way as a *write* for the register file. Figure 14 shows a logic diagram of the cache and the extra circuitry attached to it to provide micro rollback capabilities.

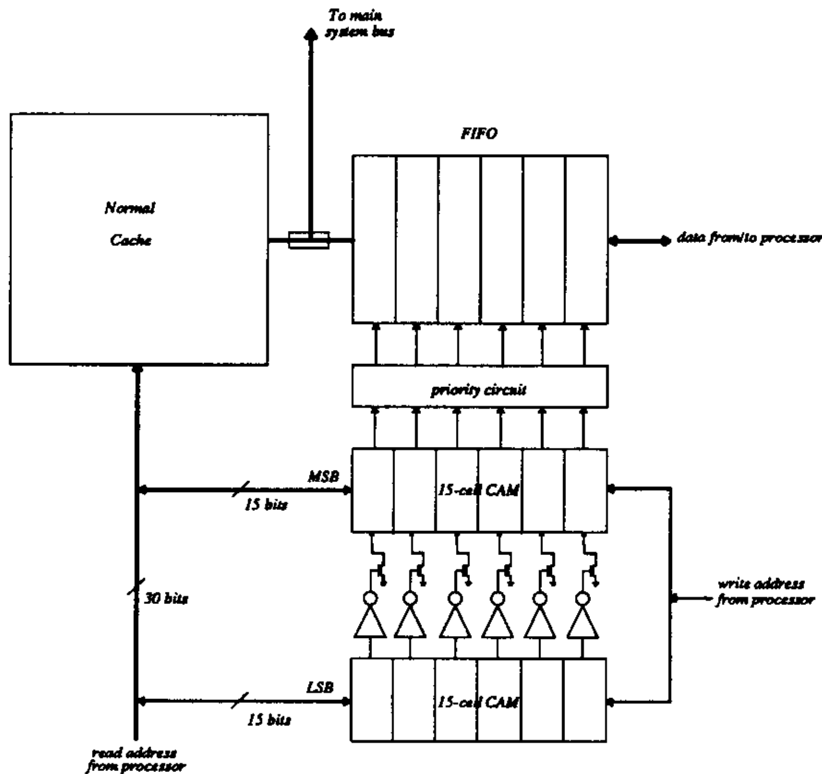

Figure 14: Logic diagram of the cache

The *store* is simply delayed for N cycles in a CAM during which time it can either be re-read by the processor or canceled in the instance of a rollback. Note that interaction between the cache and main memory occurs after the data has gone through the FIFO. 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 14 shows that the comparison in the CAM is now made on 2 sections of 15 bits, forming the real address, instead of the 7 bits composing register addresses inside the processor. The new comparison takes 22ns, instead of 8ns for the register file, but there is also more time available because of longer access time in the cache.

4.2. Micro Rollback and Main Memory

One of the problems with using micro rollbacks in all modules of the computer is that it requires synchronous operation. Each module must buffer up to N clocks of data and precisely roll back some specified number of cycles. This is difficult or impossible to do if the entire system is not completely synchronous (e.g. some high-performance buses have asynchronous protocols.)

If a cache memory is used with the processor, it is possible to use micro rollback in only the processor and cache and avoid using it with main memory. The cache provides the necessary buffering of data from the processor for N cycles before writing back to memory or I/O. Hence, if serial checking is performed for transfers between the cache and memory and between the cache and I/O devices, data received at the memory or I/O from the cache will never have to be rolled back. Thus the memory and I/O do not have to provide micro rollback if serial checking does not cause unacceptable speed delays in these modules.

Primary memory, and I/O units can wait for checking to assure data to be correct before sending it. This is because the bandwidth requirements and the effects of latency of memory and I/O are reduced because the processor uses a cache. Since communication between the cache, I/O, and main memory typically occurs in multiple word blocks performing the checks serially, in a pipeline, is likely to be an acceptable solution since the only delay is for the first word of the block. Thus the relative performance cost of waiting for checking to complete is significantly lessened.

There are significant advantages to using serial rather than parallel checks in the main memory module: 1) there is never a need to rollback main memory, 2) no special hardware is required for main memory, and 3) the cache coherency protocol does not need any modifications.

The only disadvantage in a serial check in primary memory is that it increases the memory access time. If an effective cache (with a high hit-rate) is used, accesses to main memory occur relatively rarely and this is not a major problem.

Referring to the same RISC processor, the Fairchild Clipper coupled with two 4096-byte caches for instruction and data,[2] we calculated the overhead associated with a serial check for main memory. Based on the data given in table 1, we obtained an effective memory access overhead of 9.3%, which corresponds to a 3.5% overhead of the average instruction time for a 33MHz machine.[7]

Clipper Features	
processor cycle	30ns
bus cycle	60ns
cache read	120ns
instruction miss ratio	3.15%
data miss ratio	6%
read miss delay	585ns
tlb miss ratio	0.34%
tlb miss delay	960ns
copy back percentage	3%
copy back delay	480ns

Table 1: Timing for the Clipper

Because of the advantages mentioned above and because of the low overhead, a serial check for main memory accesses proves to be the right choice.

5. 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 14). 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 memory system should ensure that "the value returned on a *load* is always the value given by the latest *store* instruction with the same address."[1] A system that meets this condition is said to be *memory coherent.*

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.*[6]

> "[A system is sequentially consistent if] 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[6] 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 14, 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 14, 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 unaccessible during N cycles plus the time it takes to execute the *test-and-set* instruction. Alternative solutions are currently under investigation.

6. 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 chips. 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 slowing down surrounding circuits). Checking delays are compounded when data is transferred between several modules, and checked at several places.

This is a fundamental problem in achieving fault tolerance in high speed VLSI systems. Concurrent fault detection will not be feasible if checking is done in series with execution (i.e. each step must be checked before proceeding to the next, and the checking delay is added to each clock cycle). Thus it is necessary to perform checking in parallel with execution. Signals to be checked are latched, and error checking takes place in one or more subsequent cycles (as a pipeline). As a consequence error signals will arrive one or more cycles after error-damaged data is received for processing.

In this paper, we have described a *micro rollback* mechanism which allows recovery when a delayed error signal arrives. It is a systematic way to design VLSI computer modules so that they can roll back and restore the state which existed when the error occurred. 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 design is a new approach which is particularly well-suited to VLSI implementations and which we believe will have wide application in future systems which must combine fault-tolerance and high performance.

Acknowledgements

This research is supported by Hughes Aircraft Company, the Natural Sciences and Engineering Research Council of Canada, and the State of California MICRO program.

References

1. L. Censier and P. Feautrier, ''A New Solution to the Coherence Problems in Multicache Systems,'' *IEEE transactions on Computers* TC-12(12) (December 1978).

2. James Cho, Alan J. Smith, and Howard Sachs, ''The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor,'' Technical Report No. 86/289, Computer Science Division, EECS Dept., University of California, Berkeley, Ca 94720.

3. M. L. Ciacelli, ''Fault Handling on the IBM 4341 Processor,'' *11th Fault-Tolerant Computing Symposium*, Portland, Main, pp. 9-12 (June 1981).

4. W. W. Hwu and Y. N. Patt, ''Checkpoint Repair for Out-of-order Execution Machines,'' *The 14th Annual International Symposium on Computer Architecture*, Pittsburgh, pp. 18-26 (June 1987).

5. M. G.H. Katevenis, ''Reduced Instruction Set Computer Architectures for VLSI,'' Report No. UCB/CSD 83/141, Computer Science Division (EECS), University of California at Berkeley, CA 94720 (October 1983).

6. L. Lamport, " ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs",'' *IEEE Transactions on Computers* **C-28**(9) (September 1979).

7. Laura Neff, ''CLIPPER Microprocessor Architecture Overview,'' *Proceedings Compcon*, San Francisco, pp. 191-195 (March 1986).

8. M. M. Yen, W. K. Fuchs, and J. A. Abraham, ''Designing for Concurrent Error Detection in VLSI: Application to a Microprogram Control Unit,'' *IEEE Journal of Solid-State Circuits* **SC-22**(4), pp. 595-605 (August 1987).