Advanced Research in VLSI:
Proceedings of the Decennial Caltech Conference on VLSI
Pasadena, California, March 1989,
The MIT Press, pp. 297-316.

Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback

Marc Tremblay and Yuval Tamir

Computer Science Department
University of California
Los Angeles, California 90024-1596
U.S.A.

In order to achieve fault tolerance, highly reliable systems often require hardware-supported concurrent error detection for all system components. Checkers are connected in the communication paths from each module to the rest of the system, reducing system performance by requiring either longer clock cycles or additional pipeline stages. The performance penalty of concurrent error detection can be minimized by performing the checks *in parallel* with the transmission of information between modules, thus removing the delay for detection from the critical path. Erroneous information may thus reach a module several clock cycles before an error indication. Operations based on this information are ‘‘undone’’ using *micro rollback* — a hardware mechanism for rapid rollback of a few cycles. In this paper we show how micro rollback can be efficiently implemented in complex VLSI systems consisting of multiple modules which interact asynchronously. The implementation of key building blocks in CMOS VLSI is described and evaluated.

1. Introduction

The ability to detect errors as soon as they occur and prevent the spread of erroneous information throughout the system is a key requirement in many fault-tolerant systems. In some environments (e.g. with high levels of radiation) a high rate of transient faults is expected and system components must be able to recover from (correct) the majority of the resulting errors without resorting to expensive software-driven rollback and reconfiguration. In order to meet these requirements, checkers, error-correction circuitry, and/or isolation circuits are usually connected in the communication paths between each module and the rest of the system. Information transfers between modules in the system are delayed by the need to wait for checks or possible correction to complete. This results in lower system performance due to increased clock cycle time or additional pipeline stages.

The delays due to concurrent error detection and/or correction can be minimized if these operations are performed *in parallel* with the transmission of information between modules. Each module processes its inputs immediately when they become available ‘‘assuming’’ that they are correct. If the data is erroneous, it is followed, after a delay of a few cycles, by an error indication or the corrected data [12]. If the data processed by the receiver is later flagged as erroneous, any changes to the state of the system due to this information must be undone. Hence, 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* [12].

Micro rollback requires each module in the system to store the information necessary to undo state changes that have occurred within the last few cycles. Efficient implementation of micro rollback in simple synchronous VLSI modules has been described in detail elsewhere [12, 3]. These results are summarized in Section 2. As described in [12], micro rollback differs from single-instruction retry [1] 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 by following simple specifications.

Current VLSI computer systems consist of many complex chips in addition to a CPU chip — for example, floating point coprocessors, memory management units, communication coprocessors, etc. In this paper we show how micro rollback can be implemented in such complex heterogeneous systems consisting of a variety of modules which may interact asynchronously. In some of the modules, state changes may occur at a lower rate than once per cycle. For these modules, the fact that there are no state changes in major components at every cycle can be used to reduce the amount of information kept for micro rollback. Specifically, if a module must be capable of a rollback of up to N cycles but it is known that there are no more than M state changes ($M < N$) during N cycles, there is no need to keep N copies of the state (or a log of N state changes). Instead, the amount of information maintained can be proportional to M. This requires a mechanism to map a ‘‘request’’ to roll back a specified number of cycles to an operation on the corresponding state changes. In Section 3 we discuss new techniques for handling such

modules, thus further reducing the hardware and performance overheads for micro rollback compared to previously published schemes [12, 3].

Previously published techniques for micro rollback are based on system-wide rollback of a specified number of cycles. In a synchronous system, when one module rolls back, other connected modules roll back the same number of cycles in order to maintain a *consistent state* [12, 9]. In a system with several modules which operate with different clocks and interact using an asynchronous protocol, (for example, the processor and coprocessors in a system based on the Motorola 68020 [8]), maintaining a consistent state is not as simple. Specifically, once one module rolls back, the number of cycles that other modules should roll back depends on recent interactions between the modules. This is a special case of the general problem of recovery in distributed systems [9] except for the requirement that the entire operation should be performed by hardware in only one or two cycles. In Section 4 we present techniques for meetings these requirements. These techniques are based on using inter-module interactions as a basis for synchronization when coordinating roll back of multiple modules to a consistent global state. This is accomplished with special-purpose circuits that translate between inter-module interactions and internal clock cycles of each module. We present the design of these circuits and their evaluation based on VLSI layouts and extensive simulation.

2. Micro Rollback

A micro rollback of a module (e.g. a CPU or a coprocessor) consists of bringing the module back a few cycles to a *state* that it had reached in the past. In order to be able to perform such an operation it is necessary to save the state of the subsystem (*checkpoint*) at each cycle boundary [3]. The *state* of a module (subsystem) is the contents of all storage elements which carry useful information across cycle boundaries. For example the state of a processor 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 registers in the state machine which can be changed during the execution of a multicycle instruction.

Checkpointing is the equivalent of taking a ‘‘snapshot’’ of the state of the subsystem every cycle. Micro rollback restores the state of a subsystem by overwriting the current state with a ‘‘snapshot’’ taken in the past (see Figure 1).

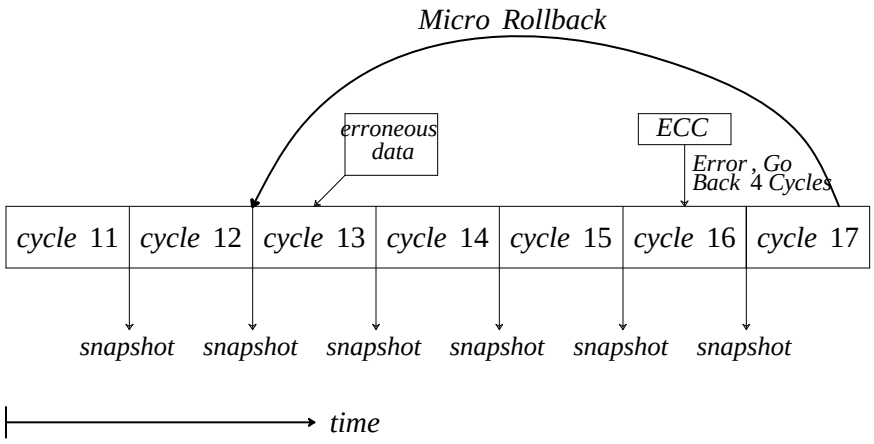

Figure 1: Micro Rollback of a Module by Restoring a Previous *Snapshot*

2.1. Application of Micro Rollback in VLSI Systems

As discussed in Section 1, micro rollback is used to allow each module in the system to accept inputs and begin processing them without waiting for detection and correction circuits to operate on the data. Error detection is thus performed in parallel with the transmission and ‘‘consumption’’ of data by modules throughout the system. This removes the checkers from the critical path of the system and permits the use of checkers that are area efficient but relatively slow, without sacrificing system performance [12]. With micro rollback it is also possible to exploit error detection techniques in which the hardware used to compute a result may then be used to verify the validity of the result and, potentially, produce an error indication a few cycles later [10, 11].

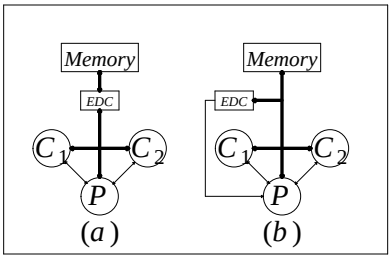

Figure 2: Concurrent vs Parallel EDC

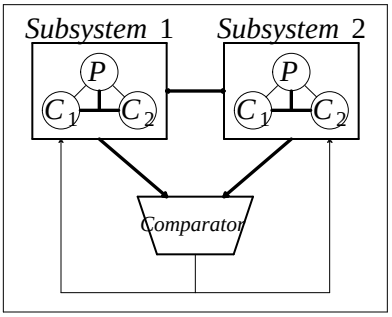

Figure 3: Subsystems in duplex mode

A common situation where parallel error checks may be useful is at the interface between memory and processors or coprocessors (Figure 2). Minimizing the access time to memory (or cache) is often critical to achieving high performance. With micro rollback the checkers (or correction circuitry) can be removed from this critical path.

Another important use of parallel checks is in duplex systems where

error detection is accomplished by running two identical subsystems in parallel and comparing their outputs (Figure 3). With this technique, which is supported by some current commercial chips [2], the two subsystems may be on different chips and there is thus a significant delay in getting both outputs to the comparator (off chip communication) and obtaining the results of the comparison. With micro rollback, the receiver of data from the duplex subsystem can begin to process it before the output of the comparator can be used to determine if the data is valid. A system based on triplication and voting (TMR) can benefit in a similar way from micro rollback.

2.2. Implementing Micro Rollback in a Simple Synchronous System

As discussed above, parallel error checks require the receiver to be able to *roll back* its state to undo state changes that are due to data which has turned out to be erroneous. Special circuits must be added to the module for preserving the state information and for executing micro rollback. We present two general techniques for rolling back the state of a typical synchronous system: the first one is used for the state of a collection of registers that are physically grouped together (e.g. register files), the second one applies to individual registers which are distributed across the chip (e.g. various status registers) [12].

All interactions between modules capable of micro rollback can be completed without waiting for error checks. Interactions with modules that are not capable of micro rollback (e.g. various peripherals, interrupts) can be handled using special-purpose interfaces between each module capable of micro rollback, M_1, and one that is not, M_2. Specifically, the interface must buffer (delay) data from M_1 to M_2 for N cycles — the maximum ‘‘distance’’ that M_1 may roll back. This ensures that the data is released to M_2 only when it is *committed* [9]. Similarly, data from M_2 to M_1 is buffered for N cycles after its receipt so that it can be retransmitted to M_1 by the interface unit if M_1 decides to roll back.

2.2.1. Micro Rollback of a Large Register File

The state of a large register file can be preserved for N cycles by simply replicating the storage N times and copying the current values to the ‘‘oldest’’ replica every cycle. This method can be very costly in terms of area. For example, the area of one copy of the register file in the Berkeley RISC II processor takes up 33% of the total chip area [5]. We have previously proposed [12] an alternative technique that takes advantage of the fact that only one register out of the set can be modified every cycle. An example of this technique is shown in Figure 4.

Every time a *write* is performed, the data and its full register address

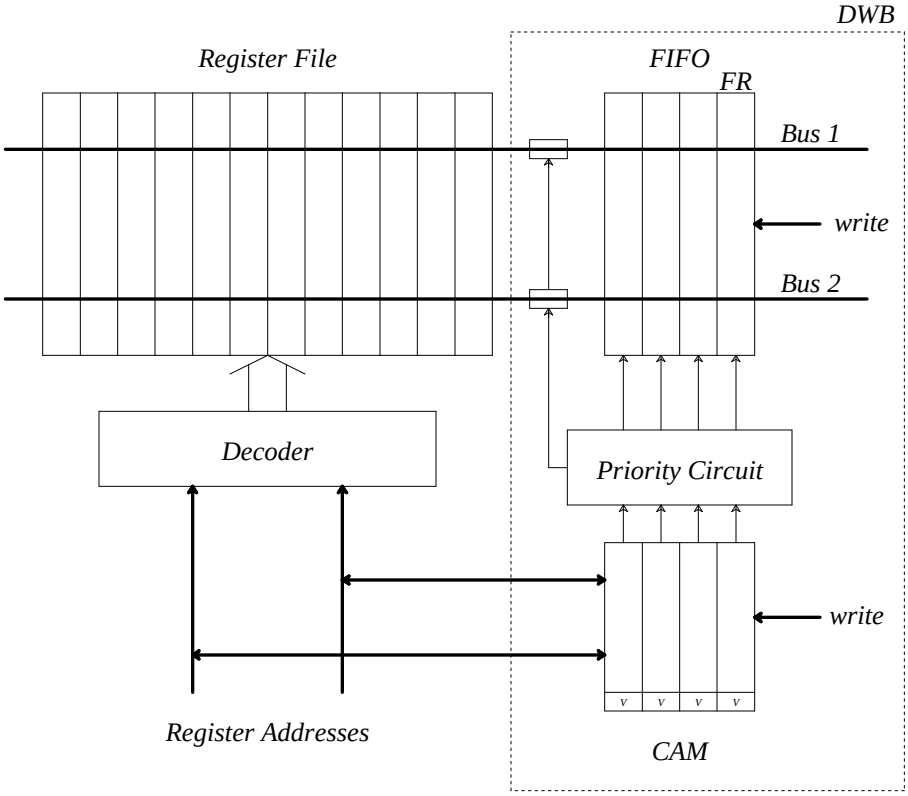

Figure 4: Micro Rollback of the Register File

are stored in a FIFO buffer, which we call a *Delayed Write Buffer* (DWB). Depending on the maximum latency (N) of errors signals, the *write* into the ‘‘real’’ register file is delayed for N cycles. During a *read*, the DWB is scanned in parallel with the register file in order to provide the most recent update of a register. At each cycle, all the data in the DWB is shifted one position to the left. A valid value in the left-most cell is written into the conventional register file. When a micro rollback of C cycles is required, we simply invalidate the last C entries in the DWB. By invalidating the last C *writes*, the state of the register file is brought back to the state it had C cycles ago [12].

We have implemented this scheme for a 32-bit VLSI RISC processor based on the Berkeley RISC II. The MOSIS Scalable CMOS (SCMOS) design rules were used. Detailed evaluation of the performance and area penalties of the scheme is presented in [12] for several register file and DWB sizes. For example, SPICE simulations of the circuit extracted from the layout, assuming a $3\,\mu$ ($\lambda = 1.5$) process, show that the access time to a register file of 64 register with the DWB of 4 cells is less than

5% slower than the access time to a standard register file of 64 registers. For the same register file and DWB, the area overhead of the scheme is 14% compared to a standard register file [12].

2.2.2. Micro Rollback of Individual Registers

The technique for implementing micro rollback of a register file cannot be used for individual registers, such as the program counter and various pipeline latches, which are physically far apart on the chip. Instead, in order to support micro rollback of up to *N* cycles, for each individual register there are *N* ‘‘shadow registers’’ which store its state for the last *N* cycles (see Figure 5). A pointer keeps track of the location in the shadow register set used for each backup. During a rollback, the pointer is decremented by the number of cycles to roll back and the contents of the state register is restored with the correct ‘‘old’’ value. The use of a RAM for the shadow registers allows a micro rollback to be achieved in one cycle (compared with up to N cycles for a stack) [12].

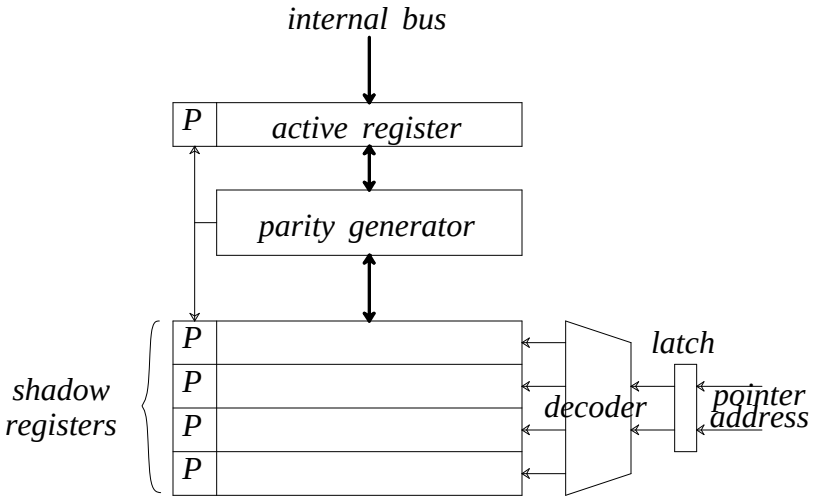

Figure 5: Micro Rollback of a Single State Register

3. A Delayed-Write Buffer for Infrequently Modified Registers

The method used to roll back the register file, described in section 2, is based on the fact that *writes* into the register file can occur every cycle. For example many RISC processors complete an instruction every cycle [4, 5]. Because it is possible to have N *writes* during N cycles, a Delayed-Write Buffer of depth **N** is necessary.

In some modules, *writes* to the register file may occur at a lower rate

than once per cycle, so a smaller DWB may be sufficient. For example in the Motorola 68881 most instructions take at least 30 cycles to execute [7]. It is unnecessary to dedicate a DWB of N registers if it is known that during N cycles at most one *write* can occur so there is at most one value to invalidate in the entire DWB at any point in time. If a rollback of N cycles can ''undo'' at most M *writes*, only M registers are needed in the DWB. Considering that floating-point registers are 80-bits wide, the gain in area can be considerable.

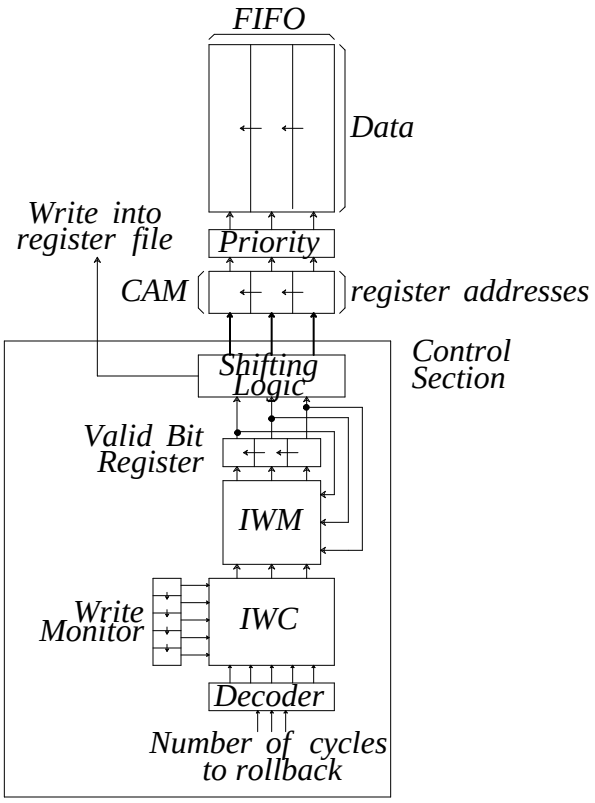

Figure 6: DWB for Infrequently Modified Registers

A DWB for a system in which at most three *writes* can occur during five consecutive cycles ($N = 5$, $M = 3$) is shown in Figure 6. The part containing the data and the register addresses is similar to the full DWB, except that it now contains M registers instead of N. The control section is significantly more complex. It includes three major new components: a Write Monitor (WM), an Invalidate Write Counter (IWC), an Invalidate Write Mapper (IWM), and some logic for the shifting of the FIFO buffer.

The Write Monitor keeps track of all the *writes* executed by the module; a one is shifted in whenever a *write* is executed, while a zero is shifted in otherwise. The number of bits in the Write Monitor is the maximum number of cycles that the register file may have to roll back (in

this case, $N = 5$). The number of ones in the Write Monitor should never exceed M (in this case, $M = 3$). If it does, an error will be signaled.

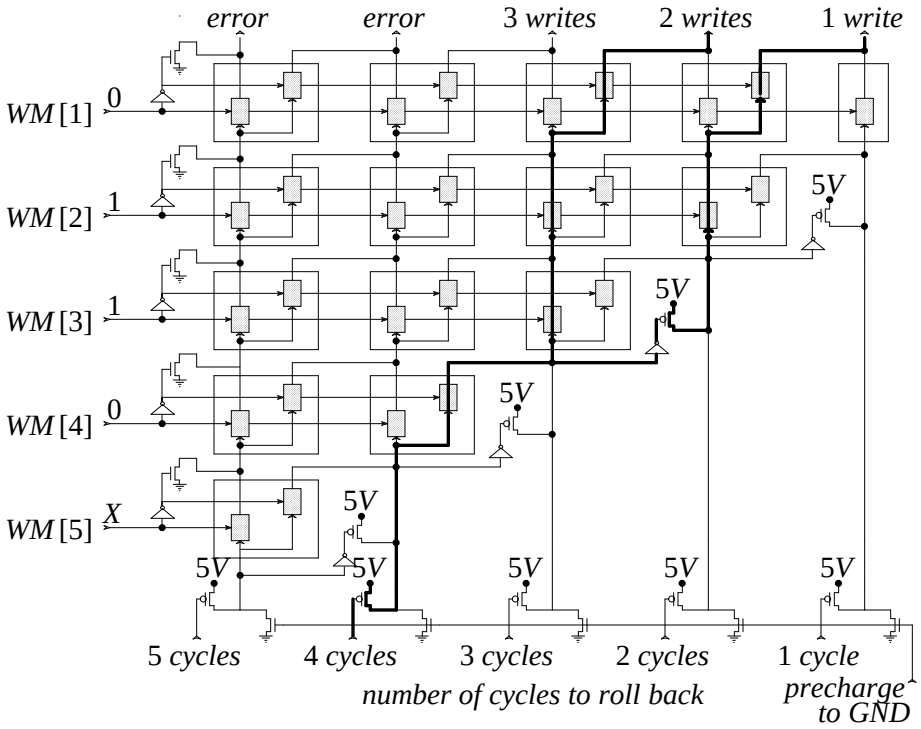

Figure 7: Invalidate Write Counter (IWC)

The Invalidate Write Counter (see Figure 7) determines how many *writes* have been executed in the past **C** cycles ($C \leq N$) based on the contents of the WM. The inputs to the circuit use negative logic. Internally, the lines carrying the number of cycles to roll back are precharged to GND in the first phase phase. In the second phase, only one input is zero so all but one of the lines are effectively disconnected from the inputs. The output of the circuit indicates that W *writes* are to be invalidated. In order to simplify the rest of the control circuitry, the circuit also indicates that $W-1$, $W-2, \cdots, 1$ *writes* should be invalidated. An error indication is signaled if the inputs result in a request to invalidate more than M writes. The basic cell for the IWC is a simple demultiplexer implemented with full transmission gates (see Figure 8). Its input is connected to output 1 when *select* $= 1$ and output 0 when *select* $= 0$. In Figure 7 we have shown a path created when the contents of the Write Monitor is [X0110] and a rollback of 4 cycles is requested. As shown in the figure, 2 *writes* (and 1 *write*) must be invalidated. The fifth entry in the Write Monitor does not modify the output because it occurred more than 4 cycles ago.

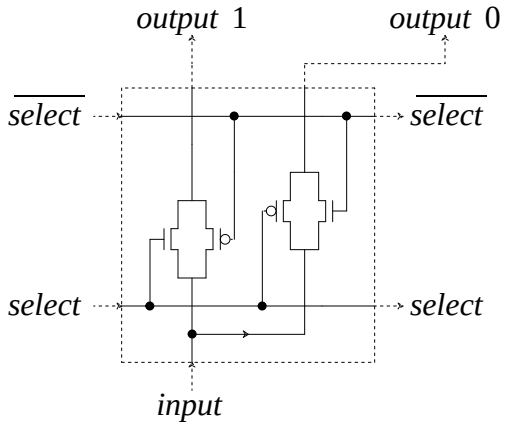

Figure 8: Basic cell for Invalidate Write Counter

The Invalidate Write Mapper is used to identify which locations in the Valid Bit Register must be invalidated. The input is the number of *writes* to undo, while the outputs are asserted when the corresponding bits in the Valid Bit Register are to be cleared (Figure 9).

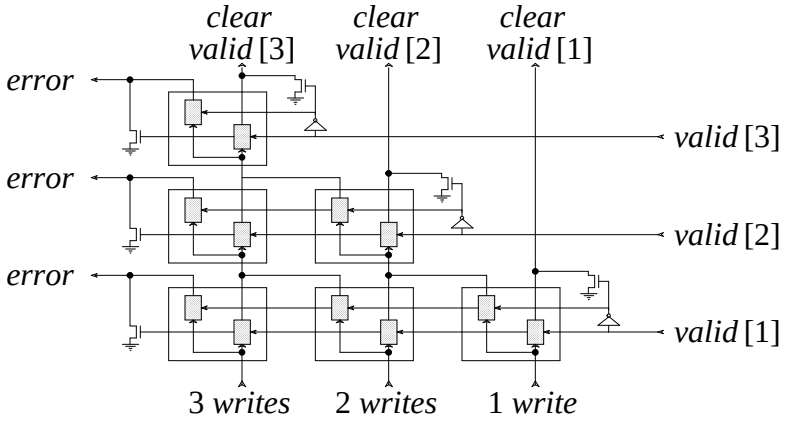

Figure 9: Invalidate Write Mapper (IWM)

The circuitry that controls the shifting in the DWB is shown in Figure 10. The leftmost register is shifted out to the register file only if the ‘‘oldest’’ bit in the Write Monitor is one. The other registers shift to the left only if there is room, otherwise they remain idle.

We have produced a layout of a complete Delayed Write Buffer similar as the one shown in Figure 6 ($N = 5$ and $M = 3$) using the MOSIS SCMOS ($\lambda = 1 micron$) design rules. The area of the circuit, including both the control and the all the storage, is 778640 λ^2, which is approximately 20% of the area of a dual-port register file with 64 32-bit registers. The control circuitry accounts for 13% of the total area for the DWB. We have determined the area of a DWB for several values of N

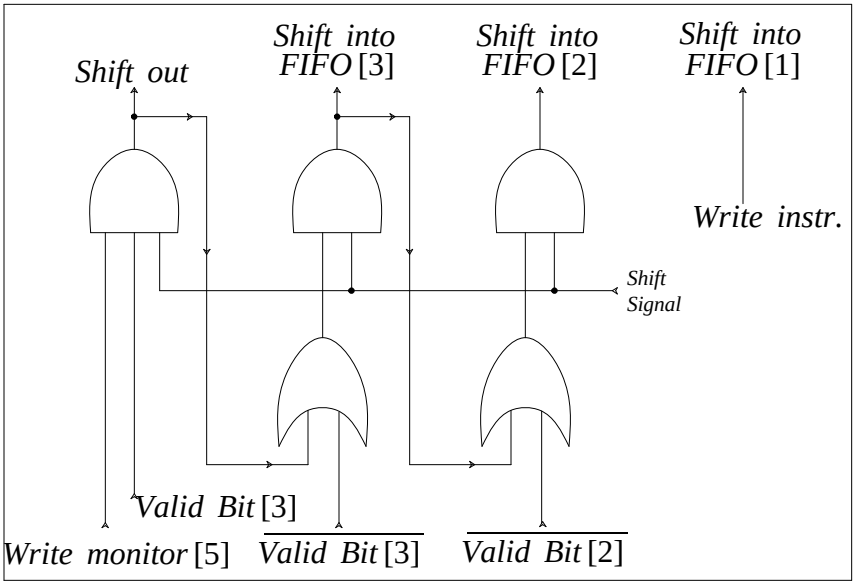

Figure 10: Control circuitry for shifting the DWB

(the maximum number of cycles that can be rolled back) and M (the maximum number of writes that can be undone). These results are summarized in Table 1 which indicates the ratio of the area of an ‘‘optimized’’ DWB (optimized for a small number of *writes*) over the area of a full version with $M = N$. For example if a module operates in such a way that its register file is never modified more than once every 4 cycles and the module may have to roll back up to 8 cycles, the table indicates that for $N = 8$ and $M = 2$ the optimized DWB takes only 34% of the area taken by a full DWB with 8 registers.

		Delayed Write Buffer Area (λ^2)			
		Generalized DWB			Full
		M=2	M=3	M=4	M=N
N (Maximum number of cycles to rollback)	4	527380 (58%)	758030 (84%)	988680 (110%)	900200 (100%)
	5	547040 (49%)	778640 (69%)	1010240 (90%)	1125250 (100%)
	8	617420 (34%)	851870 (47%)	1086320 (60%)	1800400 (100%)
	16	888700 (25%)	1130750 (31%)	1372800 (38%)	3600800 (100%)

Table 1: The Areas of ‘‘Optimized’’ and ‘‘Full’’ DWBs

We have simulated the DWB circuitry using SPICE. The critical path is through the decoder, the IWC, the IWM, and the Valid Bit register, resulting in a delay of approximately 13ns. This fast operation allows single cycle rollback.

4. Micro Rollback in a Systems with Multiple Modules

In a system that consists of several modules, a rollback signal initiated by a module may affect other modules connected to it. Following a rollback of one of the modules, its state may be inconsistent with the state of other modules. If at time T module M_1 is rolled back t time units, its new state is consistent with the state of another module M_2 if, and only if, one of the following conditions is met:

(a) there were no interactions between M_1 and M_2 in the interval [T-t,T], or

(b) M_2 is rolled back to its state prior to any interactions with M_1 during the interval [T-t,T].

In case (a) there is no need to roll back module M_2. Since M_2 has not interacted with M_1 since time $T - t$, if the states of the two modules were consistent before M_1 was rolled back, they remain consistent following the rollback without requiring further action by M_2. In case (b) both M_1 and M_2 must be rolled back. To determine which case applies as well as the ''distance'' that M_2 may have to roll back, interactions between modules must be monitored. An *interaction* or a *transaction* is any transfer of information between two modules, such as data transfer, control signals, etc. In this section we describe how we monitor transactions and how we maintain consistency throughout the system when a rollback occurs.

In a synchronous system all inter-module interactions are synchronous with a common clock. If one module, M_1, rolls back C cycles, the simplest way to maintain consistency in the system is to roll back all other modules C cycles. This implies that some modules unnecessarily roll back even if they have not interacted with module M_1 in the past C cycles. In some cases, performance can be improved if modules are rolled back selectively depending on their recent interaction with M_1. This method will be described in the context of asynchronous systems.

Many systems consist of modules that operate with different clocks and interact asynchronously. For example the Motorola 68020 processor can operate at 25MHz together with a Motorola 68881 floating-point unit operating with a 16.7MHz clock. Since there is no common clock, rollback in an asynchronous system cannot be coordinated based on the number of cycles to roll back. If two modules, M_1 and M_2, each roll back

C cycles of their internal clock, their states following rollbacks may be *inconsistent*. If module M_1 rolls back C_1 cycles internally and during the last C_1 cycles it has participated in T transactions with another module M_2, then module M_2 must roll back to the state it had prior to the last T transactions with M_1. For M_2, T transactions may correspond to a different number of internal cycles, C_2.

In order to coordinate rollback that will result in consistent states, a module that initiates rollback of a specified number of internal cycles must be able to translate that number to the number of transactions that have occurred during that time with other modules and send this number of transactions to the other modules. In order to participate in a rollback initiated by other modules, each module must be able to receive the number of transactions to rollback and translate them to internal cycles. We have designed a circuit for performing the mapping between internal cycles and transactions. We call this circuit a transactions-to-cycles/cycles-to-transactions *transducer* (see Figure 11). A module connected to several different modules through dedicated communication ports and links requires a transducer for *each* connection (see Figure 12).

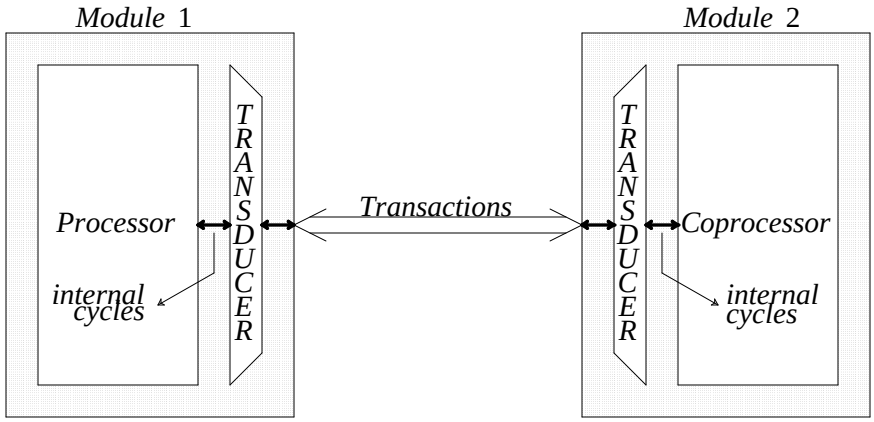

Figure 11: Synchronization Through Transducers

4.1. Cycles-to-Transactions and Transactions-to-Cycles Transducer

For a module M_1, a transducer (see Figure 13) performs three basic functions:

(1) It keeps track of transactions with another module M_2.

(2) When a rollback signal is generated internally, indicating that M_1 must roll back T cycles, the transducer determines how many transactions M_1 has performed with M_2, and sends that number to M_2.

(3) Upon receiving a rollback signal from M_2, the transducer converts the incoming number of transactions to an internal number of cycles. The

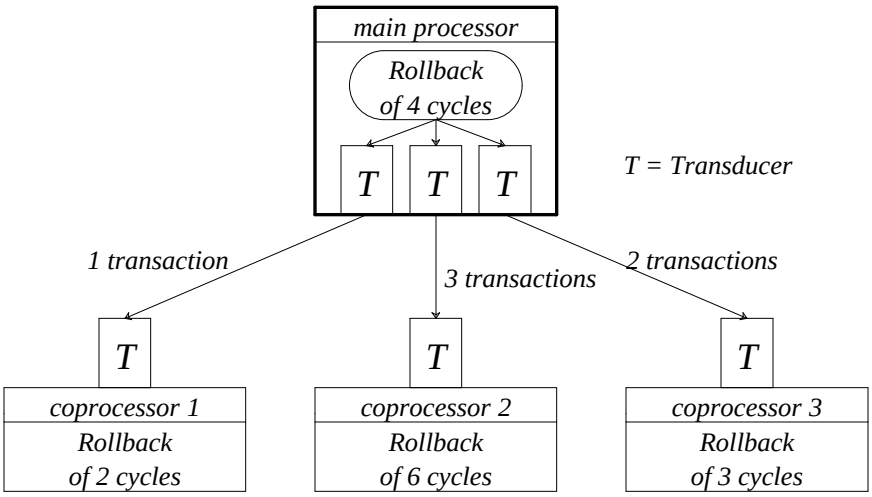

Figure 12: Transducers in an Asynchronous Multi-Module System

number of cycles is then spread through the chip and a micro rollback is initiated.

A shift register, called a *Transaction Monitor*, is used to keep track of transactions with another module. During each cycle, a 1 is shifted in if a transaction occurs, a 0 is shifted in otherwise (no transaction). When a rollback of C internal cycles is performed for a module, the first C entries in the Transaction Monitor are cleared.

Two special circuits, a ‘‘Cycles-to-Transactions Unit’’ (CTU) and a ‘‘Transactions-to-Cycles Unit’’ (TCU), are used in a transducer to perform the necessary translations. A CTU which converts a number of cycles ranging from 1 to 5 to a number of transactions ranging from 1 to 4, is shown in Figure 14. This circuit is similar to the Invalidate Write Counter described in Section 3 (see Figure 7). Instead of monitoring *writes*, it monitors transactions. The inputs are: a) the number of cycles to roll back — N, and b) the contents of the shift register monitoring transactions. The output is the number of transactions that have occurred during the last N cycles, i.e., the number of transactions that should be ‘‘undone.’’ The thick line in Figure 14 represents the connection established when a rollback of 5 cycles occurs and 2 transactions are ‘‘stored’’ in the Transaction Monitor. An error is signaled if the CTU determines that more than 4 transactions have to be undone. The maximum number of transactions possible for a given number of cycles depends on the module itself and on the protocol used to communicate with other modules.

A TCU, which is dual to the CTU described above, is shown in Figure 15. The inputs are: a) the number of transactions to roll back and

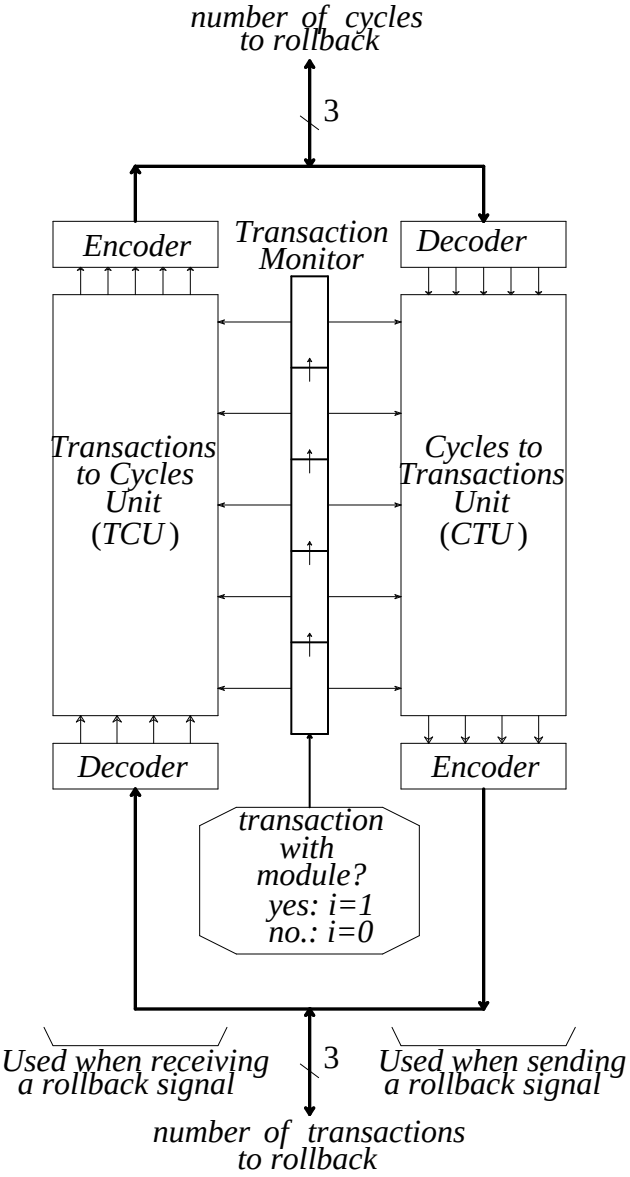

Figure 13: Transducer

b) the contents of the transactions monitor. The output is the number of cycles to roll back. In Figure 15 the thick line represents a rollback of 2 transactions requiring a rollback of 4 cycles internally. In the case where the requested number of transactions to roll back is larger than the sum of the transactions contained in the Transaction Monitor, an error signal is sent to the control unit.

We have laid out the complete circuitry for the transducer shown in Figure 13 and it measures 164000 λ^2, which represents 4% of the area of

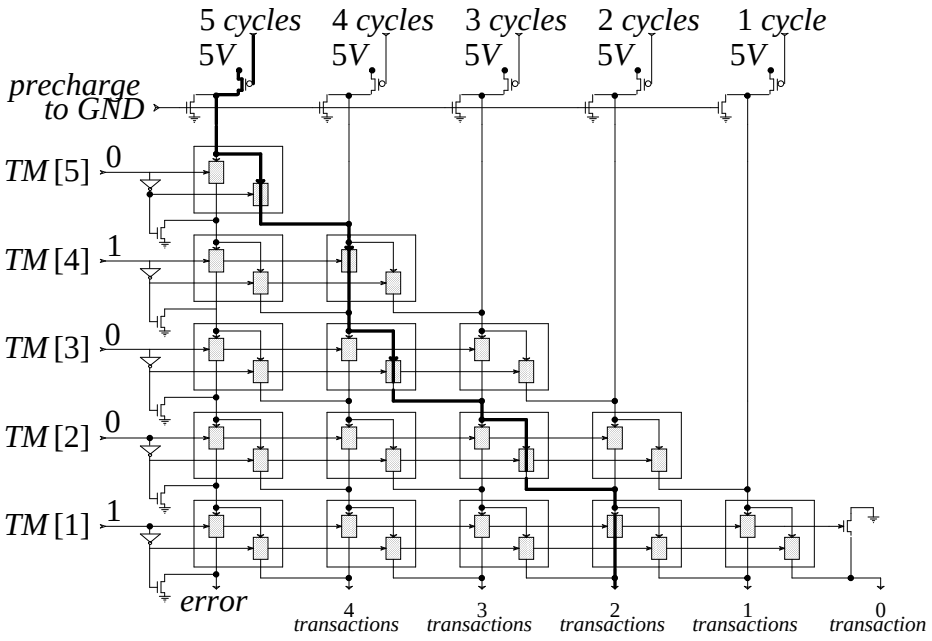

Figure 14: Circuit Converting Cycles to Transactions

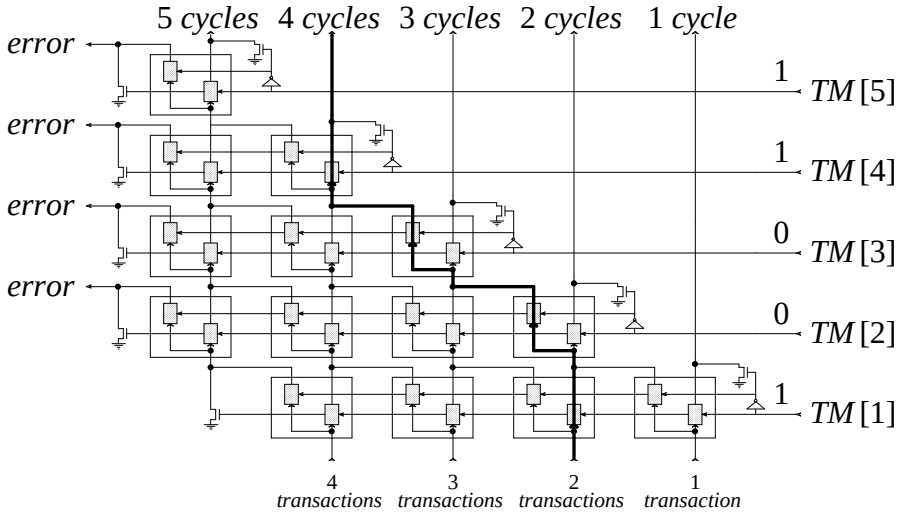

Figure 15: Circuit Converting Transactions to Cycles

a dual-port register file with 64 32-bit registers, or 22% of a simple ALU. The time required to convert cycles to transactions (and vice versa) is about 10ns. This delay, added to the delay required to roll back a large register file (13ns), makes a single-cycle rollback possible even for fast modules.

4.2. Micro Rollback in Bus-Connected Systems

Periodic checkpointing of process states and roll back to a previous state when an error is detected is a common technique for error recovery in distributed systems [9]. If each process is checkpointed independently, rolling back one process may require rolling back a second process further in time which, in turn, may cause a third process to roll back, etc. leading to an uncontrolled *domino effect* [9]. In the worst case this can result in all processes in the system rolling back to their state when the system is initialized.

In the systems discussed so far the modules are interconnected in a tree topology, where there is a unique path between every pair of nodes (Figure 11, Figure 12). In the context of micro rollback, which is done at the level of hardware modules, the domino effect cannot occur in such system. However, if the modules are connected in an arbitrary topology, where there are several independent communication paths between pairs of modules, the domino effect could, potentially, occur. Since the range of rollback is severely limited (a few cycles), this can make recovery impossible.

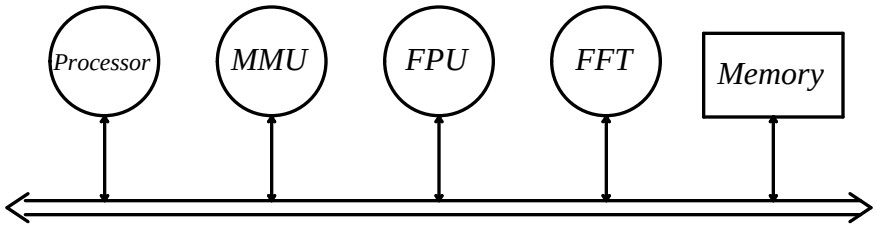

Figure 16: Bus-Based Multi-Module System

At first glance, it appears that the domino effect can be a problem when micro rollback is used in common bus-connected systems [2, 8] (Figure 16). For example in the system shown in Figure 16 the following situation could occur:

- a rollback signal is initiated in the main processor which rolls back C cycles.
- the main processor sends a number of transactions to roll back to the FPU.
- the FPU rolls back and its transducer determines that the MMU must also roll back because it interacted with the FPU during the last few cycles.
- the MMU rolls back, and its transducer sends a number of transactions to roll back to the main processor, which, in turn, is now required to roll back more than C cycles.

In a system where all modules are interconnected via a common bus, this problem can be solved by using bus transactions as a common logical clock [6]. Bus transactions can be monitored by all the modules in the system and used for synchronization. There are two possible techniques for using bus transactions to achieve a consistent state following rollback:
1) Each module has one transducer which monitors generic bus transactions. Whenever a module detects an internal rollback signal, it converts it to a number of bus transactions, and puts it on the bus. All the other modules read this number of bus transactions, convert it to an internal number of cycles, and roll back. The disadvantage of this method is that it generates unnecessary rollbacks. Modules may roll back a certain number of system bus transactions even if they haven't had any interactions with the rest of the system.
2) Each module has two transducers similar as the one described in Figure 13. The shift register (monitor) in the first transducer, shifts every time a bus transaction is executed. A one is shifted in if the bus transaction belongs to the module (*private* bus transaction). The shift register in the other transducer shifts every cycle and also shifts in a one if a private bus transaction is monitored. If a rollback signal is detected the following conversion occurs:

Generic Bus Transactions $\rightarrow$ Private Bus Trans. $\rightarrow$ Internal Cycles

In this way modules roll back only if necessary, but require twice the amount of hardware. The delays are also doubled which may make the implementation more critical for modules operating at high frequencies.

5. Summary and Conclusions

The use of concurrent error detection and/or correction in fault-tolerant systems usually results in substantial performance degradation. This performance penalty may be due to delays caused by dedicated checkers in the communication paths between each module and the rest of the system or the need to wait for the hardware to perform redundant operations that verify the validity of the ''recent'' results [10]. Micro rollback is a powerful technique that facilitates the implementation of high-performance VLSI systems which are also highly fault-tolerant. It allows a variety of concurrent error detection and correction techniques to be used with minimal performance penalty. With micro rollback, it is feasible to operate systems in hostile environments, where there is a high rate of transient faults, due to the ability of individual modules to initiate rollback and retry which can complete in a few cycles without resorting to expensive system-wide rollbacks.

The implementation of micro rollback in simple synchronous systems involves replication of small isolated registers and the use of full *delayed-write buffers* (DWBs) for storing recent state changes to large

register files [12]. If the state of a large register file does not change every cycle, it is wasteful to use a full DWB with N registers in order to be able to roll back up to N cycle. The large DWB is never fully utilized in such a module so chip area is wasted and access to the register file is delayed due to buses that are longer than necessary. We have shown that this potential inefficiency in supporting micro rollback can be overcome. Specifically, we have presented a delayed-write buffer where only one bit of storage is used for cycles in which there are no state modifications to the register file. A full DWB cell is only used for cycles in which there is a write to the register file. Using simple circuitry, rollbacks of a given number of cycles are quickly mapped into the specific register in the DWB that should be invalidated. We have produced CMOS VLSI layouts of the DWB and its control circuits and demonstrated that in some realistic situations our new design can result in savings of more than fifty percent of chip area compared to the previously used ‘‘full’’ DWB.

In a synchronous system, where all modules share a common clock and communicate via synchronous links, maintaining consistency following rollback is simple: all modules roll back the same number of cycles. In this paper we have shown that micro rollback can also be implemented in heterogeneous asynchronous systems where different modules operate with different clocks and communicate over asynchronous links. The key to implementing micro rollback in this type of system is a simple circuit, similar to that used for the optimized DWB, that translates between the number of local cycles to roll back and the number of inter-module transactions. We have presented details of the implementation of this circuit as well as techniques for applying a variation of it in bus-connected systems composed of a processor and several asynchronous coprocessors.

In this work, we show that parallel error checks in conjunction with micro rollback can be used to support fault tolerance in complex multi-module high-performance VLSI systems. We have produced CMOS VLSI layouts of key modules, evaluated their performance and area, and demonstrated that micro rollback can be supported with relatively low overhead. We are currently in the process of implementing a complete VLSI processor with support for parallel checks and micro rollback. We expect these techniques to be applied in many future systems operating in hostile environments that demand high performance and high reliability.

Acknowledgements

This research is supported by Hughes Aircraft Company and the State of California MICRO program.

References

1. M. L. Ciacelli, ‘‘Fault Handling on the IBM 4341 Processor,’’ *11th Fault-Tolerant Computing Symposium*, Portland, Maine, pp. 9-12 (June 1981).
2. Advanced Micro Devices, *Am29000 Streamlined Instruction Processor User’s Manual*, 1987.
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. G. Kane, *MIPS R2000 Risc Architecture*, Prentice Hall (1987).
5. 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).
6. L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *Communications of the ACM* **21**(7), pp. 558-565 (July 1978).
7. Motorola, *MC68881 Floating-Point Coprocessor User’s Manual*, 1985.
8. Motorola, *MC68020 32-Bit Microprocessor User’s Manual*, Prentice-Hall, Englewood Cliffs, NJ (1985).
9. B. Randell, P. A. Lee, and P. C. Treleaven, ‘‘Reliability Issues in Computing System Design,’’ *Computing Surveys* **10**(2), pp. 123-165 (June 1978).
10. D. Reynolds and G. Metze, ‘‘Fault Detection Capabilities of Alternating Logic,’’ *6th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 157-162 (June 1976).
11. K. Takeda and Y. Tohma, ‘‘Logic Design of Fault-Tolerant Arithmetic Units Based on the Data Complementation Strategy,’’ *10th Fault-Tolerant Computing Symposium*, Kyoto, Japan, pp. 348-350 (October 1980).
12. Yuval Tamir, Marc Tremblay, and David A. Rennels, ‘‘The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems,’’ *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 234-239 (June 1988).