Fault-Containment in Cache Memories for TMR Redundant Processor Systems

Chung-Ho Chen, *Member*, *IEEE Computer Society,* and Arun K. Somani, *Fellow*, *IEEE*

Abstract—Cache data errors read by a processor may cause CPU control flow error and force the system to enter a CPU-cache reintegration process in redundant processor systems. The reintegration process degrades the system performance and reliability. To reduce the occurrences of such an event, we propose a real-time error recovery scheme that provides effective fault-containment for data errors in cache memories. The scheme is based on cache data broadcasting of a dirty line after modification. It effectively exploits the redundancy of a fault-tolerant system using hardware voting. The scheme recovers from erroneous cache data written by a processor with full coverage. This error recovery feature remedies the insufficiency of error-correcting codes that are unable to prevent such an error. In addition, more than 60 percent of cache lines are fully covered for recovery due to errors originated from the cache itself, including unrecoverable ECC errors. The protocol can also be used to speedup the CPU-cache reintegration process for a temporarily failed processor. The performance overhead of the protocol is to broadcast only 2-3 percent of the total memory references.

Index Terms—Caches, error detection and recovery, fault-containment, redundant systems, transient faults.

1 Introduction

Highly reliable real-time applications, such as life critical missions, aircraft control, or transactional processing, usually employ synchronized redundant systems that use voting to mask hardware failure. Such systems require a failure probability to be less than 10^{-9} for a 10-hour mission [1], [2]. To use voting, at least three redundant processing elements must be used. Fig. 1 depicts a typical TMR (triple modular redundancy) system using redundant processors (or channels) and memory modules. In this system, each memory operation is performed after a majority of the processors agree on the operation. The bus interface unit that implements the voting/synchronization scheme detects the transient faults. If the system employs no cache memory, the performance of the system suffers because every memory cycle must go through the voting process. The voting/synchronization mechanism and memory latency become bottlenecks, degrading the system throughput and process response time [3], [4].

To remedy the performance loss, a private cache memory can be used. In a system with cache memories, if the memory cycle is a cache hit (except for the write-through cycles), then there is no need to go through the voting process. Without the voting delay and the long memory latency, the performance improves. Introduction of caches, however, increases the probability of fault occurrence and the probability of latent faults. A fault in a processor or in a cache memory may cause the processor execution state to diverge from the majority computation sequence. The fault may corrupt the cache memory or lead to an erroneous internal CPU state. The fault could be permanent, intermittent, or transient. Studies in [3], [9] showed that a large fraction of errors detected are caused by transient faults. The occurrence of transient faults can be 5 to 100 times that of the permanent faults. An error may appear in a cache line by a direct transient fault in the cache memory or because a faulty processor writes incorrect data into the cache memory, or both. A processor using the erroneous data may run away from its normal course, for instance, taking a different branch in the program. Fast and early recovery of the erroneous cache data is crucial for the overall system reliability. The transient restoration of erroneous cache data is important for many reasons, as discussed below.

The early recovery of faulty data in a cache line can prevent an error from propagating in the cache memory due to subsequent executions. This real-time recovery prevents a processor from losing the synchronization in a TMR system due to the use of incorrect data. It reduces the probability of corrupting the main memory if there is more than one faulty processors. Reloading the corrupted main memory and resynchronizing a temporarily failed processor are very expensive in real-time processing and often need special mechanisms to speedup the process [3], [7]. In addition to performance loss, the time required for global memory reintegration becomes the major reliability bottleneck [7]. Resynchronization or reintegration can bring a failed channel back to operation if the fault is transient. If the early recovery for transient faults succeeds, the costly resynchronization process for the CPU-cache or the global memory is not required. A transient fault may disappear before the resynchronization is finished, however, which means the resynchronization process was done in vain and the system wasted its computation time. To prevent the processor from using erroneous cache data, an error-

- *C.-H. Chen is with the Department of Electronic Engineering, National Yunlin University of Science and Technology, Touliu, R.O.C. on Taiwan. E-mail: chen@el.yuntech.edu.tw.*
- *A.K. Somani is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011. E-mail: arun@iastate.edu.*

Manuscript received 28 Apr. 1997; revised 8 Dec. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 104959.

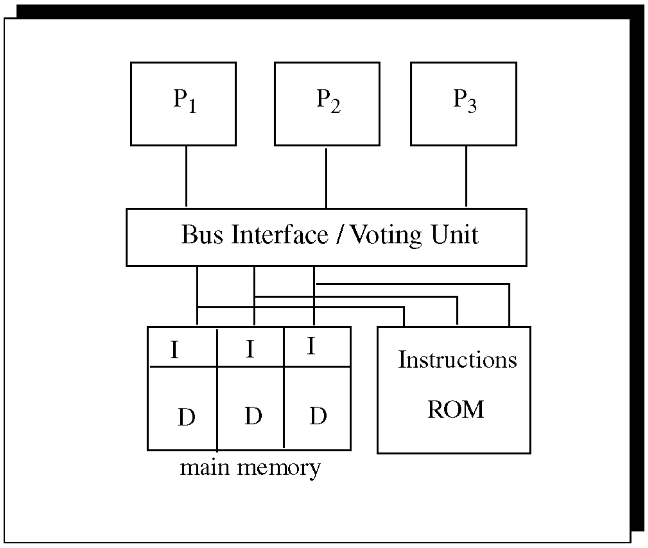


Fig. 1. A triplex fault-tolerant computer system.

correcting code (ECC) can be employed in the cache memory [10]. However, these ECC codes are only able to handle a limited number of bit errors. Erroneous data written by a processor can still corrupt the cache, even with the protection of an ECC code. Clearly, an error recovery scheme that is able to remedy the insufficiency of using ECC codes will greatly improve the system reliability or mean time between failures.

Many schemes have been proposed by several researchers using caches to assist error recovery [11], [12], [13], [15], [14]. One of the distinctive differences among these proposals is whether the error detection mechanism is assumed to exist or not. In [11], a cache-based rollback mechanism is proposed to recover a processor from transient faults. The main memory and cache memory are assumed to be reliable, so the processor can rollback to a previous checkpoint state resident in the cache memory and main memory. This method is modified for the shared memory multiprocessor systems in [12], [13] by considering the cache coherency schemes. In [14], error detection capability is provided by the comparison of data from a smaller shadow cache and from a normal cache. The recovery of error upon detection is achieved by a rollback based scheme. The Sequoia shared-bus system [16] uses software-controlled cache flush and invalidation to achieve rollback recovery. Compared with these rollback recovery schemes, the cache reloading process in TMR systems is used for the purpose of invalidating potentially corrupted cache lines and putting all processors back in synchronization.

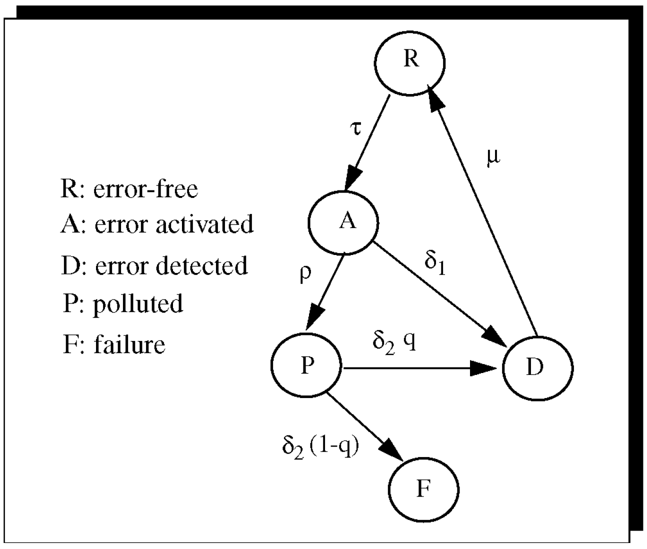


Fig. 2. A cache error model for transient faults.

In this paper, we develop a *broadcasting* protocol to realize real-time restorations of erroneous cache data for redundant TMR processor systems using hardware voting. Because various error recovery and system reconfiguration techniques are required to address system-wide issues [17], we limit the scope of the paper to the issues of fault-containment in cache memories. The proposed cache protocol emphasizes the early detection of cache errors, prevents the pollution of cache data, and thereby avoids possible resynchronization. The key feature of our protocol is to broadcast a modified dirty line to the hardware voter the first time the line is read. This straightforward approach detects and recovers a data cache error written previously by a processor due to transient fault and provides fault-containment. The scheme remedies the insufficiency of an error-correcting code when faced with processor transient faults in a TMR system. Moreover, errors generated by cache transient faults may also be recovered. The scheme is simple in implementation and tailored directly for TMR redundant processor systems using hardware voters. In summary, this paper presents a low-cost and effective real-time error detection and recovery mechanism that:

- Allows the processor or the cache memory or both to be liable for transient faults.
- Maintains cache data consistency in redundant copies.
- Supports fault-containment by preventing a processor from continuing to work on erroneous data written previously.
- Restores a polluted cache and avoids costly resynchronization due to errors in the cache.
- Eliminates the cache flush overhead when resynchronizing a processor-cache for reintegration.

The rest of this paper is organized as follows: We discuss a cache error model in Section 2. We present the proposed schemes in Section 3 and Section 4. Section 5 discusses the use of the proposed scheme in a two-level cache system. In Section 6, we present the results of protocol overhead and cache line status distribution and usage. We conclude this paper in Section 7.

2 Model of Cache Errors

In Fig. 2, we show an error handling model to characterize the processor and cache behavior in a redundant system. In this model, a system is normally in an error-free state *R*. An error activation either in the processor or in the cache memory is represented by state *A*. When the processor continues to execute the instructions, it may corrupt the cache memory and the system enters the polluted state P. In the error activated state A or the error polluted state P, the error may be detected and recovered by some detection/recovery mechanisms employed in the cache memory. In this case, the system enters the error detected state, D. After recovery in the detected state D, the system returns to state R. The detection/recovery mechanisms must be able to prevent the processor from working further with the contaminated cache data.

If the system enters the polluted state P, the employed detection/recovery mechanisms must be able to detect and recover all of the errors gradually; otherwise, the system may enter the temporarily failed state, F, where resynchronization is required. The error detection and recovery mechanism used by the cache memory or the processor must be effective so that the detection rates, δ_1 and δ_2, are high and the pollution rate, ρ, is as low as possible. To achieve this goal, it is helpful to examine the types of errors and their possible impacts on the execution of a processor. Specifically, if a transient fault occurs in the processor or the cache memory or both, the cache memory (assuming write-back protocol) may be affected in the following ways:

- Type I: One or more bits change in a cache line. This fault could happen in a dirty (modified) or in a clean (unmodified) cache line. This error is latent until the processor reads and uses the erroneous data.
- Type II: The processor writes *computationally incorrect* but *valid* data into a cache line. This may be caused by the processor's internal transient faults or by the transient faults on the processor-cache bus.
- Type III: The processor writes data using a bad address to a cache line which is not supposed to be written in a fault-free operation.

An erroneous cache line becomes a source of error which is propagated further when the processor reads from that line again. Three scenarios exist when errors are activated in a processor during the execution.

1. The processor remains synchronized with other redundant processors. In this case, the processor performs computation as in normal operation but generates incorrect results.
2. The processor may remain in synchronization for a finite time, and the erroneous data may be propagated into other cache lines.
3. The processor runs away. When the processor performs computation on the faulty data or program, this may cause a trap on some exception conditions such as an illegal instruction or result overflow. In another possible situation, the processor may take an incorrect branch of the program. The failed processor thus loses the synchronization and is out of the voting process.

Besides the scenarios described when errors are activated in a processor during the execution, there is also the possibility that the error does not affect the processor operation, for instance, when the error is masked. In view of these discussions, when the error occurs, it is desirable to keep the processor synchronized as long as possible so that the error correction is possible, and the pollution rate can be reduced. The cache recovery can be performed without invoking the resynchronization process. However, if a resynchronization process is inevitable because a processor has run away, the reintegration time should be minimized. Both of these two metrics improve the system's mean time between failures.

3 Single Level Write-Through Caches

For the system we consider, the bus interface unit that supports the voting process and the main memory modules are assumed to employ their own fault detection mechanisms and they remain fault free for cache recovery.

In a write-through cache, for each write operation, the bus interface unit conducts a bit-by-bit comparison from the outputs of all the redundant processors. A correct output is determined and written to the main memory by a majority vote. If a disagreement is detected, and the write operation is a cache hit, to recover from the error, the processor must be put in a hold state. A hold state prevents the processor from initiating any further memory references; however, the processor can continue its execution as long as it does not request a read/write access. The correct data from the voting process are then used to replace the erroneous cache data. Since the system is synchronized, all processors are required to enter the hold state whenever a write hit occurs. The hold state is released when either the voting is completed without detecting any error, or the faulty cache data are replaced.

For error recovery purposes, the cache controller responds not only to the caching requests from the processor but also to the replacement requests from the bus interface unit. To enforce the run-time detection and recovery of the erroneous cache data, the processor may suffer from a high performance penalty for using a write-through cache since any following writes or reads are blocked until the hold state is released. Besides, a cache fault may still occur in the newly written data despite of the recent verification. The processor may use the erroneous data when it reads that data again. For these reasons, we look for an alternative that uses a write-back cache to interface with the voting bus. This is addressed in the following section.

4 An Effective Fault-Containment Scheme for Write-Back Caches

In a write-back cache, the data are written into the cache to reduce main memory traffic. A cache line is chosen to be replaced using a predefined replacement policy when the need occurs to make room for a read or a write miss. When an error appears in a write-back cache, it may be propagated to other cache lines by the read/write access of the processor. This data error, if not recovered in time, may eventually cause control flow errors as reported by the results of a fault injection simulation [8]. A data error written by a processor is detected only when the line is flushed back to the main memory in a TMR system. At this time, the bus interface unit will detect the error by majority voting. However, a write hit in a write-back cache does not

```
If this is the first read hit on a dirty line after modification
    then
        each cache controller broadcasts the accessed line
        bus interface unit votes on the broadcasted lines
        if error detected
            then
                the erroneous line is replaced by the voted output
                voted output is given to all processors
            else /* no error is detected */
                each processor proceeds using its own cache line
else if cache flushing
    then
        bus interface unit votes on the outputs of all processors
        voted output is written to the memory
```

Fig. 3. The Read-First-Dirty-Broadcasting protocol for write-back caches.

invoke either the detection or the recovery procedure by the voting hardware. To prevent a processor from using any erroneous cache data, cache line verification and recovery can be performed in a write cycle or in a read cycle. If it is done on the write cycle, then every write hit even in the same line has to be verified. The processor may be held too long, losing the benefit of using a write-back cache.

Consider a scenario where a transient fault corrupts cache line X at t_1. The error stays in that line until it is read again at t_2 and, possibly, the error is propagated to the other cache line, e.g., X_1 at t_3 by a write-hit. We propose a *read-first-dirty-broadcasting* (RFDB) scheme to detect and recover such an error. Fig. 3 illustrates the RFDB algorithm. The RFDB algorithm requires each cache controller in the redundant system to broadcast a dirty line after modification to the bus voting unit when the line is read for the first time. If an error is detected in a line, the erroneous line is replaced by the majority result from voting. Therefore, the processor is prevented from using the erroneous data. If no error is detected, each of the redundant processors proceeds using its own line. It is observed that, to reduce possible frequent broadcasts resulting from a sequence of consecutive reads and/or writes on a cache line, only the first read on a dirty line after modification is broadcast.

The line state transition of a write-back cache (write allocate for a write miss) with the RFDB protocol is shown in Fig. 4. A read miss causes the line state to change from the invalid state to State A after the completion of the line fill process. Similarly, for a write miss, the line is first read in and then the write is performed. The resultant state is valid and dirty as shown in State B in Fig. 4. A read for a line in State B causes the line to be broadcast and the state transits from State B to State C. In other words, a cache line is verified when 1) the cache line is fetched into cache from the main memory as a result of a read miss, and 2) the cache line is broadcast to the bus interface unit due to the first read in state B.

We illustrate how the RFDB protocol provides fault-containment and recovers an error appearing in a cache line due to a transient fault in processor, processor-cache bus, or cache memory. For the RFDB protocol to perform correctly, the following conditions are required: 1) the majority voting mechanism works, 2) the redundant processors remain in synchronization.

Definition 1. *The set of lines which are contaminated due to the use of erroneous data in line L is denoted by* C_L.

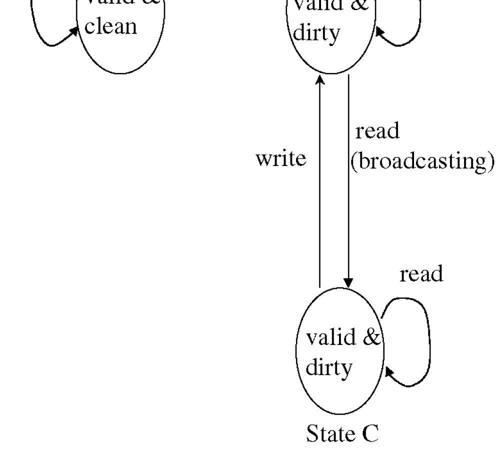


Fig. 4. Line states for the RFDB protocol.

For instance, if a processor reads from an erroneous line L and propagates errors to other cache lines, i.e., $X_1, X_2, \ldots, X_j$ through write-hits, the contaminated set is given by $C_L = \{X_1, X_2, X_3, \ldots X_j\}$.

Theorem 1. *Given that the error in line L is written with a correct address by a processor, the contaminated set does not exist. That is, $|C_L| = 0$. The RFDB protocol provides 100 percent recovery coverage for this type of data errors.*

Proof. For this type of data errors, the error source is the faulty processor or the processor-cache bus. The error in the dirty line L is recovered by the RFDB protocol when the line is read again or when the line is replaced before it is read. Thus, the RFDB protocol provides 100 percent recovery coverage for this type of data errors. □

Theorem 2. *Associated with data line L, if $|C_L| \neq 0$, then the error source is the cache itself.*

Proof. For an unmodified line, the reason is obvious. For a modified line, a data error can only produce a nonzero contaminated set if the error occurs after the read-first-dirty-broadcasting is performed. Therefore, the error source comes from the cache itself. □

Theorem 3. *The read-first-dirty-broadcasting protocol provides fault-containment by preventing further propagation of errors in C_L to other cache lines.*

Proof. With the RFDB protocol, since all lines in C_L are modified by the processor, the first read reference to any of these lines results in the broadcasting of the line and, therefore, the erroneous line is recovered by the majority voting. The RFDB algorithm prevents further error propagation and provides fault-containment. □

Definition 2. *$RD(L)$ is the sequence representing the order of lines read from C_L for the first time. $RD(L)_i$ is the ith element of $RD(L)$.*

For example, if the sequence is $X_2, X_5, \ldots$, then $RD(L)_1 = X_2$, $RD(L)_2 = X_5$, and so on. The subsequent consecutive reads on the same line are not part of the $RD(L)$.

Definition 3. *Let $CRW(L)_i$ denote the set of dirty lines which are in C_L flushed back to the main memory due to read miss or write miss between reading cache line $RD(L)_i$ and $RD(L)_{i+1}$. Those lines in C_L flushed before $RD(L)_1$ is read are denoted by $CRW(L)_0$.*

Theorem 4. *The RFDB protocol recovers the contaminated cache lines when $(i + \sum_{j=0}^{i-1} |CRW(L)_j|) \geq |C_L|$.*

Proof. After the line $RD(L)_i$ is read, there are i lines recovered by the RFDB algorithm. At that time, $\sum_{j=0}^{i-1} |CRW(L)_j|$ lines are also recovered due to cache flushing. Therefore, when $(i + \sum_{j=0}^{i-1} |CRW(L)_j|) \geq |C_L|$, the error contaminated lines due to line L are recovered. It is essential to note that those lines in C_L do not further propagate error as proved in Theorem 3. □

4.1 On the RFDB Protocol

In the transient fault injection study for a 32-bit RISC processor [8], Ohlsson et al. reported that about 60 percent of the effective errors resulted in the data errors while 33 percent of the effective errors are control flow errors. Ohlsson et al. called an error that causes a load or store instruction to reference an incorrect address or causes a store instruction to write incorrect data a data error. From Ohlsson et al.'s report, we note that the data errors are the major portion of the errors due to processor transient faults, and it takes some time (about the order of 1,000 clock cycles) for a portion of the data errors to generate control flow errors. This experimental result indicates that 1) cache data can be polluted by a processor, 2) real-time recovery and fault-containment for cache errors will help prevent the processor from entering control flow errors.

The RFDB protocol detects and recovers the incorrect data written by a processor and prevents the possible control flow errors which could have been caused due to incorrect use of data. The coverage of recovery for this type of errors is 100 percent using the RFDB protocol. This feature can be used to remedy the insufficiency of using an ECC code. On the other hand, when the erroneous data in the dirty line are originated from the cache memory itself, the RFDB protocol recovers the error when the processor performs a read from the cache line due to the demand of the program. The overall recovery coverage for this type of cache data errors is about 60 percent for the set of the programs we simulated, since around that amount of cache lines are dirty (simulation results shown later.)

Moreover, when the majority of the processors perform the read-first-dirty-broadcasting operation, if a processor uses an incorrect load address accessing a different dirty line which is modified previously, this error can be detected and recovered by the RFDB protocol. An erroneous address, if it is a cache hit, is very likely to access a dirty line which is modified previously (in state B as shown in Fig. 4) instead of a clean line, since more than 60 percent of the cache lines are dirty and in state B. In this situation, the RFDB protocol is further effective in helping to keep the processor in synchronization. The data errors discussed here are beyond the detection and recovery capability of using any ECC code in a TMR system.

If the processor accesses a clean line with a bad address while the majority of the processors perform the read-first-dirty-broadcasting operation on a dirty line, or vice versa, this error can be detected. In this case, the system can determine to enter a fast cache resynchronization process that is proposed in later section. If an error appears in a clean line, the error may remain after the transient fault has disappeared. The erroneous data in a clean line read by a processor may not cause the processor to lose synchronization directly or immediately as indicated by Ohlsson et al.'s study. A control flow error may be caused by the secondary errors which are written by the processor using the primary erroneous data. The RFDB protocol recovers these secondary errors in the cache memory and may prevent the processor from losing synchronization for this type of errors. In a later section, we describe approaches to improve the recovery performance for errors originated from clean cache lines. Table 1 summarizes the error recovery performance of the RFDB protocol discussed in this section. The proposed protocol achieves full recovery coverage of

TABLE 1
Error Recovery Capability of Using the RFDB Protocol

Types of errors	Cache error written by processor	Error from cache itself	Incorrect address errors on loads
Recovery performance	100% recovered.	60% recovered.	recovered or fast RFDB re-integration.

TABLE 2
CPU Data Errors and Control Flow Errors for a TMR System

Signal Behavior	Symptom	Remedy
2 RFDB, 1 clean	lack of one vote	re-integration
2 D Miss, 1 clean	"	"
1 D Miss, 1 RFDB, 1 clean	"	"
2 I Miss, 1 I hit	"	"
1 RFDB, 2 clean	lack of two votes	re-integration
1 D Miss, 2 clean	"	"
1 I Miss, 2 I hit	"	"
3 RFDB	1/3 inconsistency	recovered
3 I/D Miss	1/3 inconsistency	re-integration
3 RFDB	3/3 inconsistency	re-integration
3 I/D Miss	"	"
2 RFDB, 1 I/D Miss 1 RFDB, 2 I/D Miss	access type inconsistency	re-integration

2 RFDB, 1 clean: Two cache controllers issue the RFDB cycle while one cache controller issues an internal access on a clean line.

errors written from a faulty processor and, in addition, it recovers 60 percent of errors originated from the cache itself, and assists CPU-cache reintegration in run-time.

Adding the RFDB protocol into a normal cache system does not complicate the voting design of the original TMR system since a RFDB cycle can be easily distinguished from a normal miss cycle by issuing a control signal from the cache controller. Table 2 summarizes the actions to take when the system is faced with various CPU data errors and control flow errors seen from the voting interface. For instance, if all of the three cache controllers output the RFDB signal (3 RFDB in Table 2) and if one of the data broadcast is different from the other two (1/3 inconsistency), then this data error can be recovered by the voting mechanism. The cases when a faulty processor has a read hit/miss in the I-cache and other processors don't have a similar symptom as those for the data errors are shown in Table 2.

4.2 On the RFDB with ECC Codes

If an ECC code is used in the cache memory, it offers the recovery capability up to its codeword limit. Beyond the codeword limit, the processor may still use the incorrect data. Systems using the SEC-DED code, for example, in the main memory usually require software-assisted memory scrubbing to minimize the probability of multiple-bit errors. The contents of the memory are read, and rewritten back at regular, predefined time intervals [17].

Memory scrubbing can also be used in the cache memory. However, one of the major differences between the main memory and the cache memory is that the data inside the cache memory can be replaced at a rate much faster than the main memory. This is because many data items in the main memory are mapped onto the same location in the cache memory. In our trace-driven simulation for the SPEC programs, we observed that there are less than 10 percent of cache data which can be classified as *stale* for their residence in the cache memory. Thus, using memory scrubbing in the cache memory is likely to be much less effective than in the main memory.

On the other hand, the RFDB protocol recovers the unrecoverable ECC errors up to 60 percent in coverage in addition to detect and recover the incorrect data written by a processor. These features make the RFDB protocol attractive to be used with an ECC code in TMR processor systems.

4.3 On RFDB with Cache Line Invalidation

Another way of preventing a processor from using the erroneous data in a cache memory is to flush and invalidate the cache memory at regular intervals. In a write-back cache, cache flush can take considerable time to complete

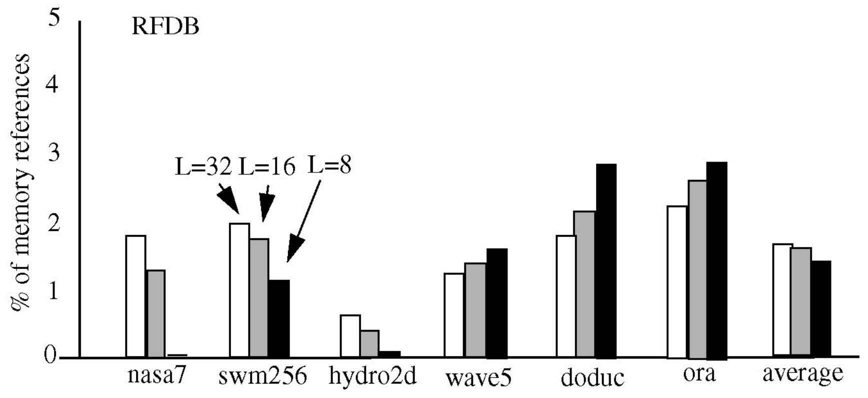


Fig. 5. Number of broadcasts (32 Kbytes, write allocate cache, set-associativity = 2).

because a large portion of cache lines are dirty. In order to enhance the detection and recovery capability for the clean cache lines and the dirty lines in state C as shown in Fig. 4, we can explore the combination of clean line invalidation and dirty line broadcasting. In this approach, the clean lines are invalidated and the dirty lines are changed into state B. With this scheme, there is no need for cache flushing during invalidation because any erroneous dirty line is recovered by the RFDB protocol before the processor uses it. Later, we show that the number of cache lines which remain clean and are occupied only once is only a small portion of the total data cache lines used. Together, this reduces the probability of error staying in the clean line which may be referenced by a processor. Also, with a small number of clean cache lines, invalidating the clean lines at, say, each iteration of a flight control program causes insignificant performance overhead.

4.4 A Cache-Based Resynchronization Process

When a transient fault occurs in a processor, it may manifest itself as an error by corrupting the processor internal states or altering the normal sequence of instruction execution. This is a control flow error accounting for 30 percent of effective errors for a fault injection study on a RISC processor according to Ohlsson et al.'s results [8]. The damage to the control flow of a processor may cause the processor to run away to an unknown state. In such cases, the faulty processor may leave the synchronization from the remaining redundant processors.

To bring the running away processor back to synchronized operation, it is necessary to reintegrate the temporarily failed processor (channel) with the healthy processors (channels). Upon the detection of the failed channel (see Table 2), a voted interrupt can be directed to all redundant channels for invoking a recovery process. In recovery, the processors need to flush the cache lines and invalidate the cache memories since the faulty processor may have corrupted the cache memory. If only one channel fails at a time in a TMR system, flushing of caches from the healthy channels produces correct output to the main memories by majority voting. The recovery time can be relatively long if the cache size is large.

To reduce this resynchronization latency, the RFDB protocol can be used to speed up the process. The approach to reduce the penalty of flushing all cache lines at a time is to allow the processor to change all of the cache lines into state B (ref. Fig. 4). Once all of the cache lines are in state B, any cache line by a subsequent first read from the processor is broadcast for verification using the RFDB protocol. The corrupted cache can be gradually recovered while the resynchronization cost is minimized. However, there may be some extra cost for flushing the dirty lines which are changed from the clean state when the dirty lines are replaced. Nevertheless, this extra cost is small since the number of clean lines is generally smaller than that of the dirty lines in the data cache.

This RFDB-assisted CPU-cache reintegration approach takes advantage of distributing the cache flush overhead into the computation phases. This cache-based resynchronization process requires very small overhead and can always be used as the first attempt in reintegrating the failed channel. For the process to be successful, the cache tag memory for the associated faulty processor and the main memory must be intact. If the cache-based resynchronization is unsuccessful by the detection of the voting

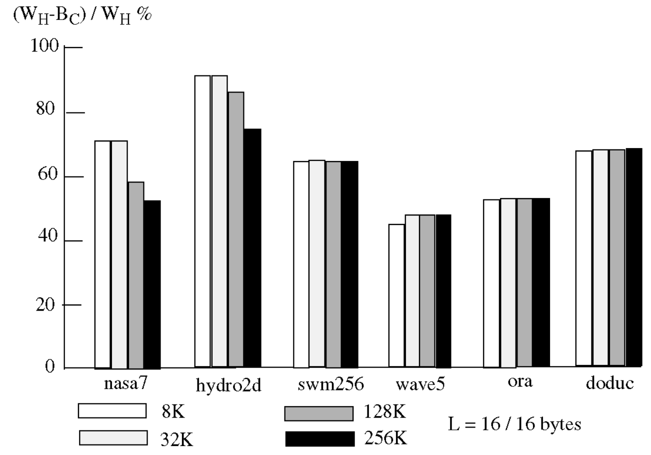


Fig. 6. Saving of write-hit memory cycles using the RFDB protocol (L = 16 bytes).

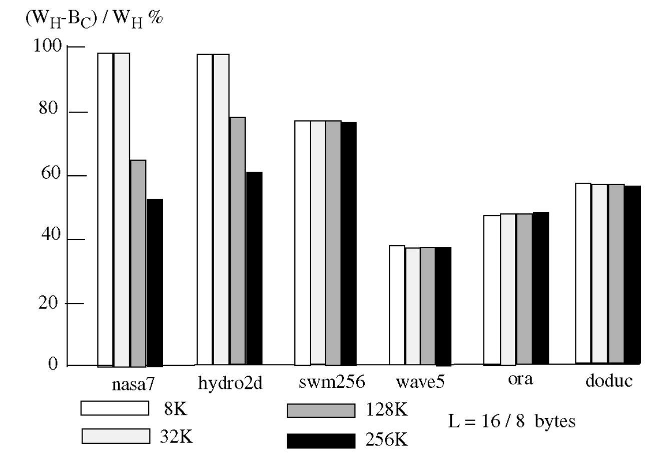


Fig. 7. Saving of write-hit memory cycles using the RFDB protocol (L = 16 bytes, subblock size = 8 bytes).

mechanism, the process of reloading the entire main memory is required for the channel reintegration.

5 Other Design Considerations

Many of the current processors have on-chip (L1) cache memories. To reduce memory penalty due to the misses in the L1 cache, an L2 cache can be used. The RFDB protocol can also be used in a two-level cache system. In such a configuration, for most of the cases, the L2 cache uses the write-back mode while the L1 cache uses the write-through mode. Also, the data in the L2 cache will be the super set of those in the L1 cache. These assumptions are used when applying the RFDB protocol in a two-level cache system.

5.1 RFDB in Two-Level Caches

For a two-level cache system, the RFDB protocol is applied in both of the two caches. In the L1 cache, the cache uses a state called "written" to implement the behavior of the read-first-dirty-protocol in the write-through mode. The written state is the "valid & dirty" state in B or C, as shown in Fig. 4. In fact, the same state transition in Fig. 4 is used for the L1 write-through cache. The line being read in state B is broadcast for verification and error correction. The performance overhead of broadcasting the L1 cache lines will be similar to that of the single level cache system we have evaluated.

When incorrect data are written into both of the L1 and L2 cache, the RFDB protocol in L1 will perform the recovery as the processor requests the data. At the same time, the voting result is also used to recover the same line in L2. If the request is an L1 miss because that line is replaced, then the RFDB protocol in L2 will perform the recovery. In either case, the processor uses the correct data by the fault-containment feature of the RFDB protocol. The recovery coverage for this type of data errors is 100 percent.

In a two-level cache system, it is possible that the data in the L1 and L2 cache may be incoherent because of cache transient faults. For instance, the L1 cache has correct data, whereas the data in L2 are corrupted. In this case, the processor uses the correct one because of read hits on L1. On the other hand, if the L1 has the incorrect data while the L2 has the correct one, the error in L1 may be recovered by the RFDB protocol as the case for a single level cache. The error recovery performance of using the RFDB protocol in both of the caches is illustrated as follows.

Definition 4. *In both of the caches, the set of lines which are contaminated due to the use of erroneous data in line L is denoted by C_{L12}. Line L can be either in the L1 cache or in the L2 cache.*

Corollary 1. *Given that the error in line L is written with a correct address by the processor, $|C_{L12}| = 0$ and the RFDB protocol provides 100 percent recovery coverage for this type of data errors.*

Proof. See Theorem 1. □

Corollary 2. *If the data error line L is in a modified state and $|C_{L12}| \neq 0$, or the data error line is in an unmodified state, then the error source is the cache itself.*

Proof. See Theorem 2. □

Definition 5. *$R(L12)$ is the sequence representing the order of lines read from C_{L12} for the first time. $R(L12)_i$ is the ith element of $R(L12)$.*

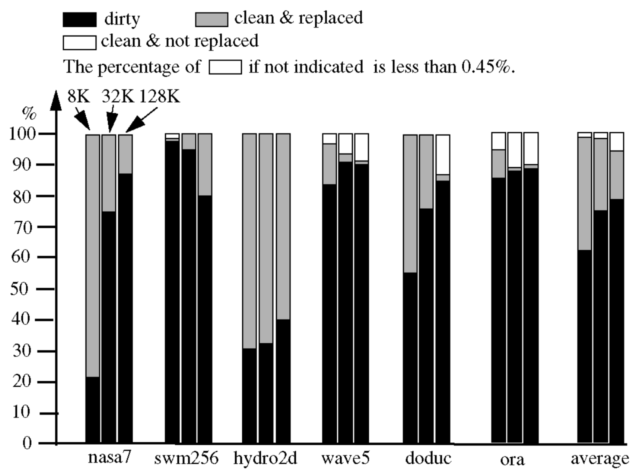


Fig. 8. Cache line usage for the SPEC programs (L = 16 bytes, write allocate cache, set-associativity = 2).

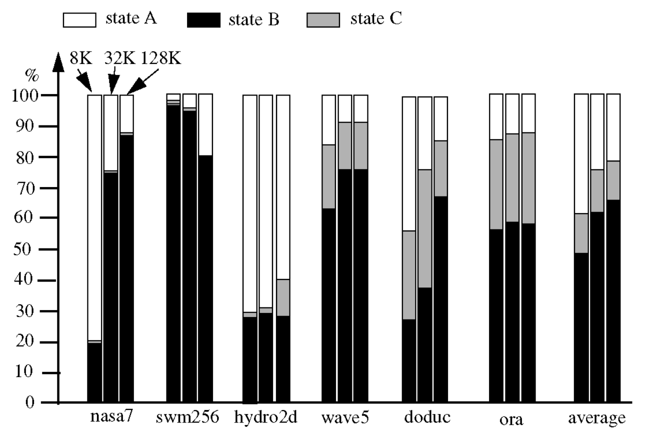


Fig. 9. Cache line state distribution (L = 16 bytes, write allocate cache, set-associativity = 2).

TABLE 3
Data Cache Performance Statistics for the Flight Control Programs

	L = 4 bytes		L = 8 bytes	
Program	$\frac{W_H}{B_C}$	$\frac{W_H-B_C}{W_H}$	$\frac{W_H}{B_C}$	$\frac{W_H-B_C}{W_H}$
roll-cmd	1.48	0.32	1.51	0.34
pla	1.98	0.49	1.98	0.49
numeric	1.46	0.32	1.46	0.32
intmm	1.02	0.02	1.03	0.03
dhry	1.57	0.36	1.60	0.37
comp	1.33	0.25	1.33	0.25
cmdch	1.71	0.41	1.71	0.41
state	1.52	0.34	1.52	0.34
am2	1.33	0.25	1.33	0.25

Subblock size = 4 bytes, set-associativity = 2, W_H = number of write hits, B_C = number of broadcasts.

Definition 6. *Let $CR(L1)_i$ denote those L1 cache lines which are in the written state, belong to the C_{L12}, and are replaced due to a read miss or a write miss between reading cache line $R(L12)_i$ and $R(L12)_{i+1}$.*

Definition 7. *Let $CR(L2)_i$ denote those L2 cache lines which are in the dirty state, belong to the C_{L12}, and are flushed back due to a read miss or a write miss between reading cache line $R(L12)_i$ and $R(L12)_{i+1}$.*

Theorem 5. *The RFDB protocol provides fault-containment and recovers the contaminated cache lines when $(i + \sum_{j=0}^{i-1}(|CR(L1)_j| + |CR(L2)_j|)) \geq |C_{L12}|$.*

Proof. There are i lines recovered by the RFDB algorithm employed in both of the caches after the line $R(L12)_i$ is read. At that time, $\sum_{j=0}^{i-1}|CR(L1)_j|$ lines in L1 are recovered due to replacement and $\sum_{j=0}^{i-1}|CR(L2)_j|$ lines in L2 are recovered due to copy back. Thus, when $(i + \sum_{j=0}^{i-1}(|CR(L1)_j| + |CR(L2)_j|)) \geq |C_{L12}|$, the error contaminated lines due to line L are recovered. □

From the above, we have illustrated how the RFDB protocol can be easily applied in a two-level cache system and maintain the same fault-containment capability as used in a single write-back cache system.

6 Protocol Overhead Evaluation

We evaluated the performance overhead of the RFDB protocol with trace-driven simulations. The programs used include the SPEC92 benchmark and several flight control programs. The flight control programs implement iterative operations that usually have multiple nested loops. The programs generate traces on a DECstation 5000 (R3000 MIPS CPU) with the *pixie* facility. The cache memory simulated is a split data and instruction cache organization. The data cache implements the write-back policy with write-allocate mode for handling write misses. The cache line replacement uses the LRU (least recently used) policy. In addition to simulating various cache line sizes, we also evaluated the case where a cache line is implemented using subblocks. A write modified bit or a dirty bit is used for each subblock in a line. Our experiment evaluated two aspects of the RFDB protocol. On the performance overhead, we examined the number of broadcasts, and the number of write-hit memory cycles in a write-through cache that can be saved by using the RFDB protocol in a write-back cache. On the effectiveness, we examined the percentage of clean lines and dirty lines and the state distribution of cache lines. The results are used to address the effectiveness of various fault-containment schemes for cache data errors, as discussed in Section 4. The simulation results are obtained for the first 100 million instructions executed.

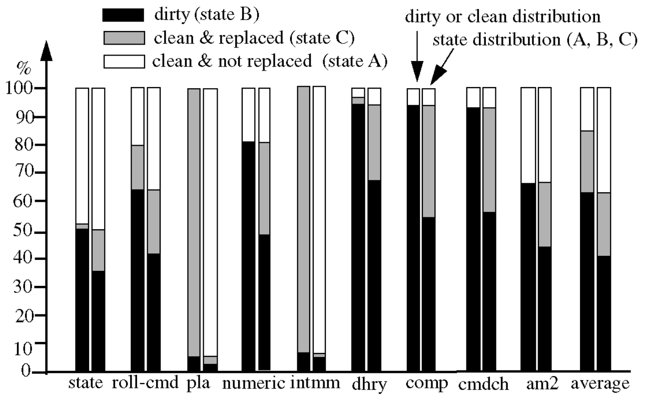


Fig. 10. Cache line usage for the flight control programs (L = 4 bytes, write allocate cache, set-associativity = 2).

6.1 SPEC92 Programs

The performance overhead paid to achieve the error recovery capabilities mentioned in Table 1 is to broadcast about 2 percent of the total number of memory references. Fig. 5 illustrates this result. For performance overhead, this is equivalent to increasing the miss ratio by 2 percent.

The saving of write-hit memory cycles is illustrated in Fig. 6 for a line size of 16 bytes. The comparison is made between the number of write-hits and the number of read-first-dirty-broadcasts since, in a write-through cache, each write-hit is also a memory cycle. The simulation results indicate that using the RFDB protocol in a write-back cache for error detection and recovery significantly reduces the number of write-hit memory cycles as required with a write-through cache. For most of the cases simulated, the ratio of write-hits to broadcasts is much larger than two. Thus, more than 50 percent of the write-hit memory cycles (ratio $\frac{WH-BC}{WH}$, where WH is the number of write-hits and BC is the number of broadcasts) in a write-through cache are eliminated in a write-back cache using the RFDB protocol.

To further reduce the cost of broadcasting an entire cache line, we studied the RFDB protocol on a subblock basis. The advantage of broadcasting a subblock is that the voting latency can be reduced. The broadcasting frequency is reduced if the first read is done on a clean subblock while

TABLE 4
Cycle Time (CT) Parameters

instruction CT = 25 nsec,	read miss per line CT = 160 nsec,	read/write hit CT = 50 nsec.

any of the other subblocks in the same cache line is modified by a write. However, first-reads on the modified subblocks in the same cache line may increase the frequency of broadcasts. Both of these two cases are observed from the simulation. Fig. 7 illustrates the saving of write-hit memory cycles for the RFDB protocol with subblock broadcasting. The subblock size is 8 bytes in this study.

The subblock protocol may increase or decrease the write-hit to broadcast ratio, depending on how good the program can fit in the cache. On an average, more than 50 percent of the write-hit memory cycles required in a write-through cache are exempted in a write-back cache using the RFDB protocol. Because a whole cache line is usually two or four times the size of a subblock, the voting latency of a subblock based RFDB scheme is much smaller than that of a full line-based broadcasting.

We examined how many cache lines are dirty and how much data may stay in clean cache lines which are only occupied once during the execution of a program. We recorded the number of dirty lines and clean lines which are never replaced for every 100 instructions executed. The average numbers of line usage obtained for simulating the first 10 million instructions are shown in Fig. 8. It is possible to examine the status of each cache line for each instruction executed; however, the simulation time is very long for large caches.

We observed that most of the programs exhibit either a high percentage of dirty lines and/or a lower percentage of clean lines that are never replaced. The SPEC programs, on an average, have more than 60 percent of the cache lines that are dirty for the 8KB cache while more than 70 percent of the cache lines are dirty when the cache size is larger than 32 Kbytes (ref. Fig. 8). With a large portion of cache lines being in dirty state, considerable cache flush overhead is saved using the approach which combines clean line invalidation and the RFDB protocol. Most notably, the percentage of the clean lines which are never replaced after they are brought into the cache is very small (less than 5 percent). Thus, the probability of erroneous data resident in a clean cache line which is only occupied once is minimized by the execution of the program itself. In general, a larger cache has a higher percentage of dirty lines.

The distribution of cache line state is shown in Fig. 9. As observed, about 60 percent of the cache lines are in state B. This result shows that any data error that originated from these cache lines themselves can be fully recovered by the RFDB protocol. For the programs we simulated, caches of 128KB and 256KB have very close hit ratio. This is why we only examined the cache size from 8KB to 128KB. We expect that the results should be similar with large caches above 256KB.

6.2 Flight Control Programs

Because most of the flight control programs are small in size, and all of them are iterative, we use a cache size of one Kbytes for the study. For such a cache size, the working set of these iterative programs fits into the cache very well. Table 3 lists the results of the write-hit to broadcast ratio for different line sizes. The write-hit to read broadcast ratios are generally smaller than those of the SPEC programs. This is because these control programs tend to fit into the cache memory very well, even for a small cache.

The line usage and state distribution obtained for the flight control programs are shown in Fig. 10. There are about 60 percent of the cache lines, which are used (valid), modified by the processor. The flight control programs have a higher percentage of clean lines which are never replaced. This is due to the iterative nature of the programs.

To further ensure the integrity of data in clean lines and the dirty lines in state C, we explore the approach by invalidating the clean lines and, at the same time, changing the dirty lines into state B. For the flight control programs, this can be done by invalidating the clean lines at the end of each iteration of a program so that any erroneous data are not carried over to the next iteration. One can expect that the performance overhead from the invalidation on clean lines is not significant because of the high percentage of dirty lines. We computed the extra time per iteration due to the invalidation by using the average number of clean lines multiplied by the difference of the read miss cycle time and the read hit cycle time. The extra miss cost due to invalidating the clean lines increases about 2 percent on an average in the execution time using the cycle time parameters in Table 4.

The recovery performance of the RFDB protocol and various recovery schemes are summarized in Table 5. Considering the processor and cache transient faults and judging from the results of cache line usage, it appears that the most robust cache error recovery scheme is an ECC-based RFDB protocol that provides both memory protection and fault-containment.

7 Conclusion

An effective design for the real-time error detection and recovery in fault-tolerant computing systems is crucial to the system's reliability. Traditionally, an ECC code and memory scrubbing are used to provide cache data recovery and fault-containment. However, such schemes do not possess the recovery capability for any CPU data errors, the majority of CPU errors shown by a previous transient fault injection study. Therefore, in a TMR system, an effective cache error recovery mechanism has to emphasize the recovering of data errors resultant from processor transient

TABLE 5
Features of Various Error Recovery Protocols in a TMR Redundant Processor System

Features	ECC	ECC + scrubbing	ECC + inv. (clean)	RFDB	RFDB + ECC
Recover from erroneous data written by the processor?	No.	No.	No, unless the data are invalidated before used.	Yes. Provide fault-containment.	Yes. Provide fault-containment.
Recover from data errors originated from cache itself?	Yes, but limited by codeword capacity. No full coverage.	Yes, multibit recovery improved. No full coverage.	Yes, better than using ECC only. No full coverage.	Yes, but limited to modified lines with full coverage.	Yes, improved for clean lines.
Recover from secondary errors or ECC unrecoverable data errors?	No.	No.	No, unless the data are invalidated before used.	Yes, but limited to modified lines with full coverage.	Yes, improved for clean lines.
Assist CPU-cache re-integration?	No.	No.	No.	Yes.	Yes.
Performance overhead	Read/Modify /write cycles for each cache access.	R/M/W cycles + scrubbing cycles.	R/M/W cycles + invalid. + flushes.	2-3 % of memory accesses are broadcasts (SPEC92)	broad-casts (2-3 %)
Data cache area increased	25%: 40-8 code	25%	25%	Virtually none.	25%

faults, since the majority of cache lines are modified by a processor.

The RFDB protocol provides fault-containment in a cache memory and prevents the possible control flow errors which could have been caused due to incorrect use of data. The recovery coverage for the errors written with correct addresses is 100 percent using the RFDB protocol. This fault-containment feature is best used to remedy the insufficiency of using any ECC-based schemes in the cache memories of a TMR system. In addition, the RFDB protocol is able to recover errors from a bad read access on a modified cache line. The performance overhead of the RFDB protocol is very small, only adding about 2-3 percent of the total memory references for broadcasts. The simulation results also indicate that most of the programs exhibited either a high percentage of dirty lines or only a small percentage of clean lines that are never replaced. About 60 percent of the data errors originated from the cache itself can be recovered by the RFDB protocol.

The RFDB protocol can be easily applied in a two-level cache system and provides the same fault-containment capability as in a single level cache. In some situations, a control flow error may cause a processor to leave the synchronization operation eventually. Speedup on the CPU-cache reintegration becomes important. The RFDB protocol reduces this reintegration overhead by first restoring the cache memory in the computation phase of the application task. Because the RFDB protocol introduces only a small performance overhead, the protocol, when used with an ECC-based scheme, is a very compelling CPU-cache data error recovery scheme for redundant processor systems.

Acknowledgments

This work was supported in part by the funding of NSC-84-2213-E-224-007, Taiwan, Republic of China. This paper is an updated version of the paper previously appeared in the Proceeding of the 24th International Symposium on Fault-Tolerant Computing [18].

References

[1] A.L. Hopkins Jr., T.B. Smith, and J.H. Lala, “FTMP-A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft,” *Proc. IEEE,* vol. 66, no. 10, pp. 1,221-1,239, Oct. 1978.

[2] J.H. Wensley et al. “SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control,” *Proc. IEEE,* vol. 66, no. 10, pp. 1,240-1,255, Oct. 1978.

[3] S.J. Adams, “ Hardware Assisted Recovery from Transient Errors in Redundant Processing Systems,” *Proc. 19th Symp. Fault-Tolerant Computing,* pp. 512-519, 1989.

[4] R.E. Harper and B.P. Butler, “Rapid Recovery from Transient Faults in the Fault-Tolerant Processor with Fault-Tolerant Shared Memory,” *Proc. IEEE/AIAA/NASA Ninth Digital Avionics Systems Conf.,* pp. 355-359, 1990.

[5] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems,* second ed. Digital Press, 1992.

[6] D. Jewett, “Integrity S2: A Fault-Tolerant Unix Platform,” *Proc. 21st Int’l Symp. Fault-Tolerant Computing,* June 1991.

[7] K.K. Goswami, R.K. Iyer, and L. Young, “DEPEND: A Simulation-Based Environment for System Level Dependability Analysis,” *IEEE Trans. Computers,* vol. 46, no. 1, pp. 60-74, Jan. 1997.

[8] J. Ohlsson, M. Rimen, and U. Gunneflo, “A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog,” *Proc. 22nd Symp. Fault-Tolerant Computing,* pp. 316-325, 1992.

[9] X. Castillo, S.R. Mcconnel, and D.P. Siewiorek, “Derivation and Calibration of a Transient Error Reliability Model,” *IEEE Trans. Computers,* vol. 31, no. 7, pp. 658-671, July 1982.

[10] C.L. Chen and M.Y. Hsiao, “Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review,” *IBM J. Research and Development,* vol. 28, no. 2, Mar. 1984.

[11] D.B. Hunt and P.N. Marinos, “A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique,” *Proc. 17th Symp. Fault-Tolerant Computing,* pp. 170-175, 1987.

[12] R.E. Ahmed, R. Frazier, and P.N. Marinos, “Cache-Aided Rollback Error Recovery Algorithms for Shared-Memory Multiprocessor Systems,” *Proc. 20th Symp. Fault-Tolerant Computing,* pp. 82-88, 1990.

[13] K. Wu, W.K. Fuchs, and J.H. Patel, “Error Recovery in Shared Memory Multiprocessors Using Private Caches,” *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 2, pp. 231-240, Apr. 1990.

[14] A.K. Somani and S. Kim, “Transient Fault Detection in Cache Memories by Employing a Small Shadow Cache,” *Proc. Sixth Ann. Int’l Symp. Dependable Computing for Critical Applications (DCCA-6),* Mar. 1997.

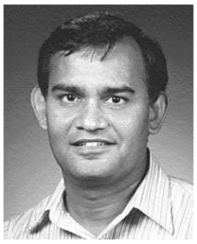
[15] M. Banatre and P. Joubert, “Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor,” *Proc. 20th Symp. Fault-Tolerant Computing,* pp. 89-96, 1990.

[16] P.A. Bernstein, “Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing,” *Computer,* vol. 21, no. 2, pp. 37-45, Feb. 1988.

[17] J. Sosnowski, “Transient Fault Tolerance in Digital Systems,” *IEEE Micro,* pp. 24-35, Feb. 1994.

[18] C.-H. Chen and A.K. Somani, “A Cache Protocol for Error Detection and Recovery in Fault-Tolerant Computing Systems,” *Proc. 24th Symp. Fault-Tolerant Computing,* pp. 278-287, 1994.

Chung-Ho Chen received his MSEE degree from the University of Missouri-Rolla and his PhD degree in electrical engineering from the University of Washington, Seattle, in 1989 and 1993, respectively. He has been an associate professor with the National Yunlin University of Science and Technology in Taiwan since 1993. His research works are in the field of computer architecture, data network switches, and parallel processing systems.



Arun K. Somani earned his MSEE and PhD degrees in electrical engineering from McGill University, Montreal, Canada, in 1983 and 1985, respectively. He is currently the David C. Nicholas Professor of Electrical and Computer Engineering at Iowa State University. He worked as a scientific officer for the government of India, New Delhi, from 1974 to 1982 and as a faculty member of electrical engineering and computer science and engineering at the University of Washington, Seattle, from 1985 to 1997 (as a full professor from 1995-1997). Professor Somani’s research interests are in the area of fault-tolerant computing, computer interconnection networks, optical networking, computer architecture, and parallel computer systems. He has taught courses in these areas and published more than 140 technical papers. He was elected a fellow of IEEE for his contribution to the theory and application of computer networks. He has served on several program committees of various conferences in his research areas and was the general chair of IEEE FTCS-97.