THE EFFECTS OF FAULTS IN CACHE MEMORIES

by

Gurindar Sohi

Computer Sciences Technical Report #727

November 1987

THE EFFECTS OF FAULTS IN CACHE MEMORIES

Gurindar Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706, USA
Phone: (608)262-7985 Arpanet: sohi@cs.wisc.edu

Abstract

As processor speeds increase, on-chip caches to provide adequate memory bandwidth are becoming increasingly important. Such caches are prone to faults both during manufacturing and during normal processor operation because of the large density of active components. Since the CPU's interactions with the memory dictate the performance of the processor, it is important to evaluate the effect of faults in the cache memory system.

Faults in components such as registers, busses, control logic, etc., are critical faults because the processor will cease to operate correctly unless some action is taken to tolerate such faults. Cache memory, on the other hand, is not a critical component of the processor - it is present mainly for performance reasons. The processor will be able to operate in a correct but degraded fashion if parts of the cache memory are faulty and adequate means are provided to recover correct data through bypassing faulty cache components. Traditional techniques for tolerating faults in memory systems such as Single Error Correction and Double Error Detection (SECDED) codes may not be appropriate for a cache since they increase the latency of the cache. If the cache memory system does not have the ability to correct errors, two questions arise: (i) how does one make sure that correct data can be recovered at all times and (ii) how do faults in the cache affect performance. In this paper, we discuss the performance of cache memories under fault conditions. Through the use of simulation, we study how the performance of a cache is degraded under fault conditions.

Keywords

Cache Memories, manufacturing faults, operational faults,
error correction, cache performance, simulation.

1. Introduction

Advances in semiconductor technology and the ever-increasing demand for processing power have led to the development of high-performance single chip processors. Since such processors are generally modeled after von Neumann's view of a computer, i.e., a CPU connected to a memory, an increase in CPU speed must be coupled with an increase in memory bandwidth [1]. Several techniques for improving memory bandwidth exist. By far the most popular technique for improving memory bandwidth in general purpose processors is the use of cache memories. For a single chip processor, this translates into the use of on-chip cache memories since off-chip transactions are comparatively slower. Indeed, many recent single-chip processors use some form of on-chip caches to provide adequate memory bandwidth for the CPU [2-7].

Unfortunately, an increase in the sophistication of single chip processors is coupled with an increase in the probability of bad or faulty components in the processor. Bad components could arise during manufacturing because of yield problems or could arise during normal operation because of the large density of active components. We expect that a large fraction of chip resources will be devoted to cache memories in the near future (note that the trend towards processors that devote a large fraction of chip resources to on-chip memories has already begun[2, 4, 5]). Consequently, we expect a large fraction of faults in such chips to be present in the cache. Therefore, analyzing and evaluating the effects of faults in the on-chip cache becomes quite important.

Components of a processor such as registers, busses, control logic, the ALU, etc., are critical to the functioning of the processor. Faults in such components are *critical faults* because the fault will lead to incorrect processor operation unless some action is taken to tolerate and/or correct such faults. Consider, for example a fault in a register. Instructions that utilize the faulty register have no alternate modes of operation without violating the architectural definition of the instruction and will fail unless means are provided to tolerate the fault. Likewise, an ability to tolerate faults in the main memory must also be provided. Cache memory, on the other hand, is not a critical component of the processor. Cache memory is present in a processor mainly for performance reasons. The processor will be able to operate in a correct but degraded fashion if parts (or all) of the cache memory are unavailable and if alternate means are provided to recover and access correct data. For example, if data cannot be accessed from a faulty cache block, it can always be recovered from the memory without violating the architectural definition of the instruction. We call faults in non-critical components such as the cache, *non-critical* faults. While detecting non-critical faults is very important correcting non-critical faults may not be important if alternate means to achieve the same purpose exist.

Techniques for tolerating faults in critical components of a processing system are well-known [8]. In a memory system[1], one can tolerate faults by incorporating some error checking and correction (ECC) capability in the memory system. A typical memory system uses a SECDED Hamming code to correct single errors and detect double errors in a memory word. When the memory word is read, error correction logic determines if the data is correct, corrects the data if a single bit is in error or indicates that a double error has occurred. ECC techniques that do not use Hamming codes also exist [9]. Such fault-tolerance techniques can easily be applied to tolerate non-critical faults in the cache memory. However, the logic needed to

[1] For the remainder of this paper, the memory refers to the architecturally visible memory and the cache refers to the (generally) architecturally transparent cache memory.

implement such schemes increases the latency of each memory (or cache) access. Since a prime concern of the cache is to reduce the latency of memory requests, the use of error correcting logic in a cache is not very attractive. Indeed, the author is unaware of any processing system that uses ECC in the cache memory.

The focus of this paper is to evaluate the effects of faults in the cache memory system and thus answer the following questions: (i) are ECC techniques necessary for a cache memory, (ii) might simple parity schemes be adequate and (iii) do chips need to be discarded if the manufacturing process results in some faults in the on-chip cache. We start off with the assumption that ECC logic for the on-chip cache is too expensive to implement both in terms of chip resources and cache access time. Adequate means must, therefore, be provided to recover correct information at all times. Then we evaluate the degradation in the performance of the cache due to faults using simulation techniques. The results of our evaluation show that, in most cases, ECC in the on-chip cache is quite useless though, in some situations, a limited use of ECC might be appropriate. Finally, we apply the results of the simulations and discuss the effects of faults in the on-chip caches of some real-life processors.

2. Faults in Cache Memories

In its most general form, a cache memory consists of several *address blocks* or *address lines* of data[2]. Each cache address block is occupied by elements from an address block of the memory. An address block consists of several contiguous bytes of memory. Data from an address block is present in the *data array* of the cache. Each address block has an associated tag which is kept in the *tag array*. The tag is used to distinguish between one of several address blocks that are present in the cache. Large address blocks require less number of bits in the tag array but generate more memory traffic [10]. In order to reduce the overhead for transferring data blocks to and from the cache, each address block can be broken into several *transfer blocks*. A transfer block is the amount of data transferred from the memory into the cache on a read miss. Caches which have the same size address and transfer blocks (by far the majority of caches) are a special case.

A cache with transfer blocks smaller than address blocks operates as follows. The tag generated from the address is compared with the the tag of the block(s) stored in the appropriate positions of the tag array. If no match results, the reference is a miss. If a match results, the cache looks in the data array for the transfer block. A *present/not present* bit indicates if the transfer block is present in the cache. If the block is present, a hit results and the data is referenced from the cache, else a miss results and the block is brought in from the memory.

By having large address blocks and small transfer blocks, the cache can have the best of both worlds, i.e., fewer bits in the tag array and a smaller amount of cache-memory traffic. Apart from reducing the amount of traffic generated, smaller address blocks have other advantages specially for single-chip processors. A discussion of these advantages is beyond the scope of this paper. Suffice it to say that several processors use caches with large address blocks and smaller transfer blocks [3, 5, 11, 12].

[2] In this paper, we shall use the term *blocks* instead of *lines*.

2.1. Types of Cache Faults

Faults in the cache memory can be broken down into two classes depending upon where they occur: (i) faults in the data array or *cache data* faults and (ii) faults in the tag array or *cache tag* faults. A cache tag fault does not pollute the data stored in the cache, i.e., does not pollute the contents of the data array, but it affects the *hit* operation. If the stored tag is incorrect, the cache has no way of knowing the correct tag of the address block present in the cache. An access operation in the presence of a cache tag fault may, therefore, result in the access of possibly valid data from an incorrect address block. A cache data fault does indeed pollute the data in the cache data array. Bad data in the cache could result in incorrect computations and pollute the data in the memory if no steps are taken to rectify the problem.

Cache tag and cache data faults can either be *manufacturing* faults or *operational* faults. Manufacturing faults arise due to defects in manufacturing, i.e., problems with the yield. Operational faults arise during normal operation and can either be *permanent* or *transient*. Permanent faults arise due to hard errors during normal operation. If a manufacturing fault exists in a block, the block cannot be used to cache valid data (for the remainder of this paper, a faulty block shall refer to an address block if the fault is in the tag array and to a transfer block if the fault is in the data array). Data that would normally be obtained from the faulty cache block would have to be obtained from elsewhere. For a direct-mapped cache, data that maps onto the faulty cache block would have to be obtained directly from the memory, for a set-associative cache, the data may be present in another cache block. Therefore, if the processor is to be used in the presence of faulty cache blocks, means must be provided to bypass the cache selectively and access the data directly from the memory if need be.

Operational cache tag faults will corrupt information in the tag array. The cache must, therefore, be able to detect the fact that the tag information is indeed corrupted. This can easily be done if a *parity* bit is provided with each tag. In case of a permanent cache tag fault, the block can no longer be used to hold a valid tag, i.e., the effective size of the cache is reduced by 1 address block. In case of a transient cache tag fault, normal operation of the cache will automatically cleanse out the fault, i.e, the replacement algorithm will replace the faulty tag with the tag of another block, automatically correcting the error. No other corrective action needs to be taken if the cache management hardware translates a fault indication into a cache miss.

Operational cache data faults are slightly more cumbersome. First, such faults must be detected. This is easily accomplished by using a parity scheme. Next, since such faults will corrupt the data in the cache and eventually in the memory, means must be provided to recover the correct copy of data in the cache at all times. If ECC is provided in the cache, the data can be corrected automatically. If ECC is not provided, then correct data must be recovered from elsewhere. An alternate copy of the data can exist in the main memory or another cache in the processing system. Since a fault can occur at any time, an alternate copy must be made with each cache update and, therefore, the on-chip cache must be a *write through* cache[13] (unless a rollback scheme is invoked and the program restarted from the rollback point). Note that a read only cache (for example an instruction cache) is a special case since no writes are allowed into the cache anyway and a correct copy always exists elsewhere. A permanent cache data fault will reduce the effective size of the cache by one transfer block. Note that the entire address block containing a faulty transfer block need not be declared faulty since valid data can still be accessed from the other transfer blocks of the address block. If a correct copy of the data can be obtained from elsewhere, a transient cache data fault will automatically be cleansed out by normal cache operation.

2.2. Write Through or Write Back?

A write through cache, can become a bottleneck in a computing system since it generally requires more memory bandwidth than a *write back* or *copy back* cache. Unfortunately, a write back cache cannot guarantee that the latest and correct copy of data can be recovered from elsewhere in the presence of faults in the cache and, therefore, cannot be used in the absence of ECC in the cache data array. Fortunately, a write through caching policy for the on-chip cache is not inconsistent with a high-performance single-chip processors' view of its memory system. Such processors need 2 levels of cache in the memory hierarchy: (i) a small on-chip or level 1 cache and (ii) a larger off-chip or level 2 cache. The main purpose of the level 1 cache is to reduce the latency for the memory requests generated by the processor; the purpose of the level 2 cache is both to reduce the latency for off-chip memory requests and to reduce the traffic generated between the level 2 cache and the main memory [3, 5, 14]. This is specially important if the processor is to be used in a *multi* system in which several processors are connected together through a common bus as shown in Figure 1 [15]. If the bandwidth of the level 2 cache-memory interconnection is important, the level 2 cache must be a write back cache. Therefore, in order to guarantee the integrity of data in the level 2 cache, some form of ECC must be provided in the level 2 cache. The level 1 cache must be a write through cache if it does not have ECC. Making the level 1 cache a write through cache to ensure fault-tolerance is not an overhead; indeed in the absence of adequate data-coherence algorithms, the level 1 cache is generally a write through (or a read only) cache.

For such a memory system, the overall access time of the memory as seen by the CPU is:

$$T = h_1 T_1 + (1-h_1)h_2 T_2 + (1-h_1)(1-h_2)T_m$$

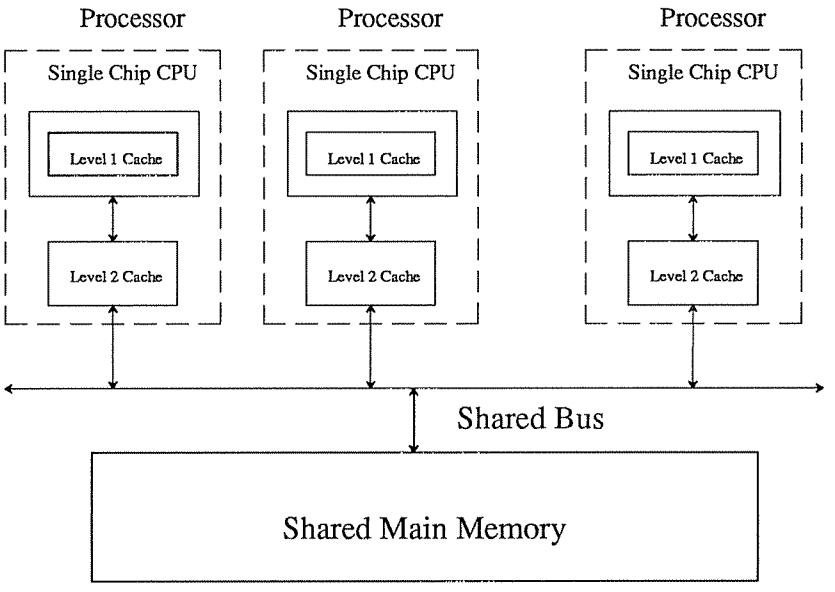

Figure 1: A Multiprocessor System with High-Performance Single-Chip Processors

where h_1 is the hit ratio and T_1 is the access time of the level 1 cache, h_2 is the hit ratio and T_2 is the access time of the level 2 cache and T_m is the access time of the memory (this includes the bus waiting time if any). An increase of δT_2 in the access time of the level 2 cache will increase the overall memory access time as seen by the processor by $\delta T = (1-h_1)h_2\delta T_2$ while an increase δT_1 in the access time of the level 1 cache will increase the overall memory access time by $\delta T = h_1\delta T_1$. Since h_1 would generally be greater than 0.5, the degradation in the average memory access time due to a degradation in the access time of the level 2 cache (due to ECC) is not very significant. Also, the degradation in the access time T due to a degradation in T_2 will be less significant than an equivalent degradation in T_1.

To summarize the above discussion, we made the following observations: (i) the on-chip cache must be a write through (or read only) cache; the off-chip cache can be write back and can use ECC logic without degrading the overall memory access time significantly, (ii) an ability to detect errors in the on-chip cache must be provided, (iii) an ability to bypass faulty blocks in the on-chip cache must be provided, (iv) transient faults in the on-chip cache are automatically corrected by normal caching operation and (v) permanent and manufacturing faults reduce the effective size of the on-chip cache. Now let us see how the cache is able to identify faults.

2.3. Identifying Faults

Each cache block (both the address block in the tag array and the transfer block in the data array) is appended with an extra bit called the *faulty* bit. This is similar to the *fault tolerance* bit used in the RISC-1 instruction cache [16]. If this bit is set, the block is permanently faulty and cannot be used to cache valid information. This is in addition to any other bits that might be provided for such blocks. For example, the address block may have a *valid/invalid* bit and the transfer block may have a *present/not present* bit. Each block also has a set of parity bits that are used to detect errors. For the tag array, a single parity bit is sufficient since the entire tag is used when determining if a reference is a hit or a miss. For the data array, more than one parity bit may be necessary. This is because parity must be computed when each datum is referenced by the processor. Since the size of a datum reference is not guaranteed to be the same as the size of the transfer block (for example, the processor may reference a byte and the transfer block may consist of 4 bytes), a single parity bit for the entire transfer block may be inefficient. A simple solution is to keep a parity bit for each of the smallest addressable data units in a transfer block. For example, we could keep a parity bit for each byte in a transfer block.

Initially, the faulty bit of each block that does not have a manufacturing fault is set to zero. The faulty bit of a block with a manufacturing fault (a fact that can be established by a testing procedure) is set to 1 to indicate that such a block cannot be used to cache information. During normal processor operation, when a data reference is made, the parity is computed (this may require the computation of several parity bits depending upon the size of the referenced data). If the parity bits indicate no error, the cache supplies the data normally. If the parity bits indicate an error, the cache considers the reference a miss and reads the data from the memory into the same cache block. The parity is computed again. If the recomputed parity indicates no error, the fault was a transient fault which was automatically corrected by the read operation. If the recomputed parity still indicates an error, we can assume that a permanent fault exists in the cache block. If a permanent faults exists, the faulty bit of the block is set to 1. A faulty block is now excluded from the cache management algorithms, i.e., any reference to a faulty block is a miss and the faulty block is never chosen by the cache replacement algorithms.

3. Behavior of Caches under Fault Conditions

In this section, we study the behavior and evaluate the degradation in performance of a cache memory under fault conditions. As is obvious from the previous discussion, a transient cache data fault causes exactly one extra miss for each fault (the transfer block has to be fetched from memory). A transient cache tag fault will cause at most P extra misses for each fault, where P is the number of transfer blocks in each address block. This is because all the transfer blocks of the address block will be present in the cache after P misses. Since these numbers are relatively insignificant compared to the total number of misses experienced by the cache, we shall exclude them from further consideration. Permanent and manufacturing faults which result in the loss of entire blocks in the cache result in misses to memory blocks that might have occupied the non-faulty cache blocks. Indeed, any long-term degradation in the cache hit ratio will arise because of permanent and manufacturing faults.

3.1. The Sensitivity of a Cache Organization to Faults

In a direct mapped cache, a memory block can be present in only one cache block. If the memory size is K times the cache size, K memory blocks map onto each cache block. If a cache block is faulty, K memory blocks that map onto the faulty cache block cannot be present in the cache. For example, consider the cache-memory system of Figure 2. The cache has 4 blocks and the memory has 16 blocks. If the cache were direct mapped, under normal operation memory blocks $\{M_0, M_4, M_8, M_{12}\}$ map onto cache block C_0. A fault in the tag of cache block C_0 will, therefore, exclude memory block $\{M_0, M_4, M_8, M_{12}\}$ from the cache. A set associative cache is less restrictive. A single fault does not automatically exclude any memory block from being present in the cache. If all the blocks from a set are faulty, then some memory blocks cannot be present in the cache. Suppose that the cache of Figure 2 were 2-way set associative and cache blocks $\{C_0, C_1\}$ comprised set 0 of the cache. Under normal operation, memory blocks $\{M_0, M_2, M_4, M_6, M_8, M_{10}, M_{12}, M_{14}\}$ could be present in either cache blocks C_0 or C_1. Though a fault in cache block C_0 will not exclude any memory block from the cache completely, the probability of interference amongst the memory blocks that map onto set 0 of the cache will increase whereas the probability of interference for the other sets will remain unchanged. A fully associative cache always allows every memory block to be cached (unless the entire cache is full of faulty blocks). Furthermore, all memory blocks are treated equally; no set of memory blocks experiences a greater interference that another set.

Thus, we expect that a direct mapped cache will be quite sensitive to faults, a set associative organization less sensitive and a fully associative organization quite insensitive to faults. Since we could not develop any analytical models to predict the performance of different cache organizations under fault conditions, we decided to carry out extensive trace driven simulations.

3.2. Simulation Methodology

We simulated three different cache sizes: (i) a 256 byte cache, (ii) a 1K byte cache and (iii) an 8K byte cache. These cache sizes were considered to be typical cache sizes for on-chip caches of high-performance processors of the near future. A direct mapped, a two way set associative and a fully associative organization was simulated for each cache size. An LRU replacement strategy was used for the set and fully associative organizations. The block size was also varied for each cache. Faults were injected at random. A fault has the effect of preventing any data from being cached in the faulty block. Since the various blocks of the cache are not accessed precisely in the same fashion, two different caches with the same number of faulty

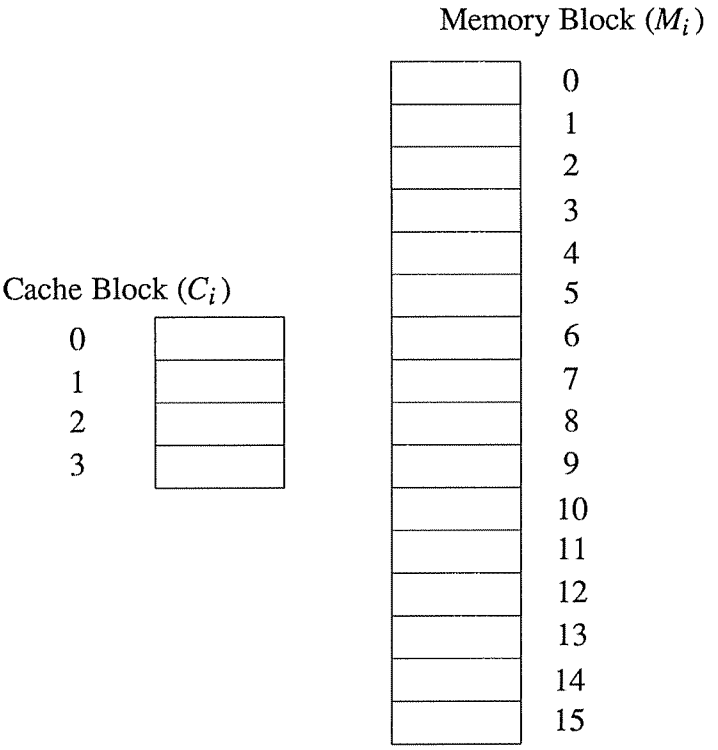

Figure 2: An Example Cache-Memory System

blocks (but different faulty blocks) may differ in performance. In order to overcome this problem, we simulated each cache organization several times for the same number of faulty blocks but with a different set of faulty blocks for each run. The set of faulty blocks was chosen at random. The miss ratios for each run were then averaged out. Care was taken to perform a sufficient number of runs so that the variance in the miss ratio for a particular number of faulty blocks was not too high.

In our simulations, we assumed that the address block size is the same as the transfer block size and, therefore, a faulty transfer block causes the same degradation in performance as a faulty address block. If the transfer block size is smaller than the address block then, on the average, the performance degradation due to a faulty transfer block is bounded from above by the performance degradation due to a faulty address block.

The benchmark programs used to simulate the caches were taken from trace tapes provided to us by DEC. The traces were generated for a VAX-11/780 using the ATUM trace technique [17]. The specific traces chosen were: (i) *fora.000*, (ii) *forf.000*, (iii) *ivex.000*, (iv) *lis2.000*, (v) *lisp.000*, and (vi) *mul8.000*. Each trace was run up to a maximum of 300,000 memory references. Each cache organization was simulated with these traces running back to back, i.e., each cache organization was simulated for approximately 1 million references. The caches were unified instruction and data caches. For the cache sizes simulated, we do not expect the results for split instruction and data caches to be significantly different. The simulation results are presented in Figures 3-5. The figures plot the average cache miss ratio versus the percentage of

blocks that are faulty for direct mapped (DM), 2-way set associative (TW) and fully associative (FA) caches with various block sizes (BS in bytes). For the 256 byte and 1K byte caches, we have plotted the complete range of faulty blocks. For the 8K byte cache, we have truncated the curves at 50% faulty blocks to allow for a better look at the miss ratio degradation, specially for a fully associative cache. We plot the miss ratio as opposed to the absolute degradation in *processor performance* since we did not want to make any assumptions about the cost of cache misses, i.e., the number of cycles that it takes to service a cache miss. Before the results can be applied to a particular system, the degradation in cache performance would actually have to be converted to a degradation in processor performance. Also note that the results have been presented for VAX traces. We expect results for other traces to follow the same general pattern.

3.3. Discussion of the Simulation Results

The results of the simulations are quite interesting. Consider a 256 byte cache of Figure 3. If the cache were organized as a direct mapped cache, the miss ratio would degrade almost linearly with the number of faulty blocks. Thus, if the block size was 16 bytes, four faulty blocks would degrade the miss ratio from about 0.325 to about 0.493. If, however, the cache were organized as a fully associative cache, the miss ratio would degrade only from about 0.259 to about 0.304. The degradation in miss ratio for a two-way set associative cache would be in between the two limits. The results for 1K and 8K bytes caches are also quite similar. As expected, associative caches can tolerate the loss of cache blocks better than direct mapped caches.

A reader interested mainly in cache performance might also find the results of Figure 5 interesting. For a fully associative 8K byte cache, a loss of 50% of its blocks would only degrade the miss ratio from 0.054 to 0.064 if the block size is 8 bytes. This result is not unexpected since a fully associative 8K byte cache with 50% of its blocks faulty is essentially the same as a fault-free, fully associative 4K byte cache.

From the figures we also see that, for an arbitrary cache organization, the loss of a cache block is more disastrous if the block size is larger. Consider, for example, a 256 byte direct mapped cache. If the block size were 4 bytes, 4 faulty blocks would translate into a loss of 6.25% of the total blocks in the cache, i.e., a degradation in the miss ratio from about 0.420 to about 0.456. If, however, the block size was 16 bytes, a loss of 4 blocks would translate to a loss of 25% of the cache blocks, i.e., a degradation in the miss ratio from about 0.325 to about 0.493. This means that if the transfer blocks are sufficiently small, faults in the data array will not degrade the performance of the cache significantly (though they would increase the fault-free miss ratio). The use of ECC logic to correct faults in the data array will, therefore, be unnecessary in a cache design that has small transfer blocks. Of course, the cache must be designed so that it can bypass a faulty block and provide data to the CPU directly from the memory. If the address block is large, as it indeed should to save on the number of tags, faults in the tag array can degrade the performance of the cache. This degradation will be quite insignificant if the cache has a high degree of set-associativity. If the cache is direct mapped, one might choose to provide some ECC in the tag array to tolerate some of the faults. We expect that the degradation in the cycle time of the cache access operation will not be as significant if ECC is provided in the tag array of a direct mapped cache since the tag matching process may not be in the critical path. However, this would have to be verified for the individual cache design.

To summarize the above, caches that have a high degree of set-associativity and/or caches that have smaller block sizes are more tolerant to faults, i.e., faults do not affect the performance

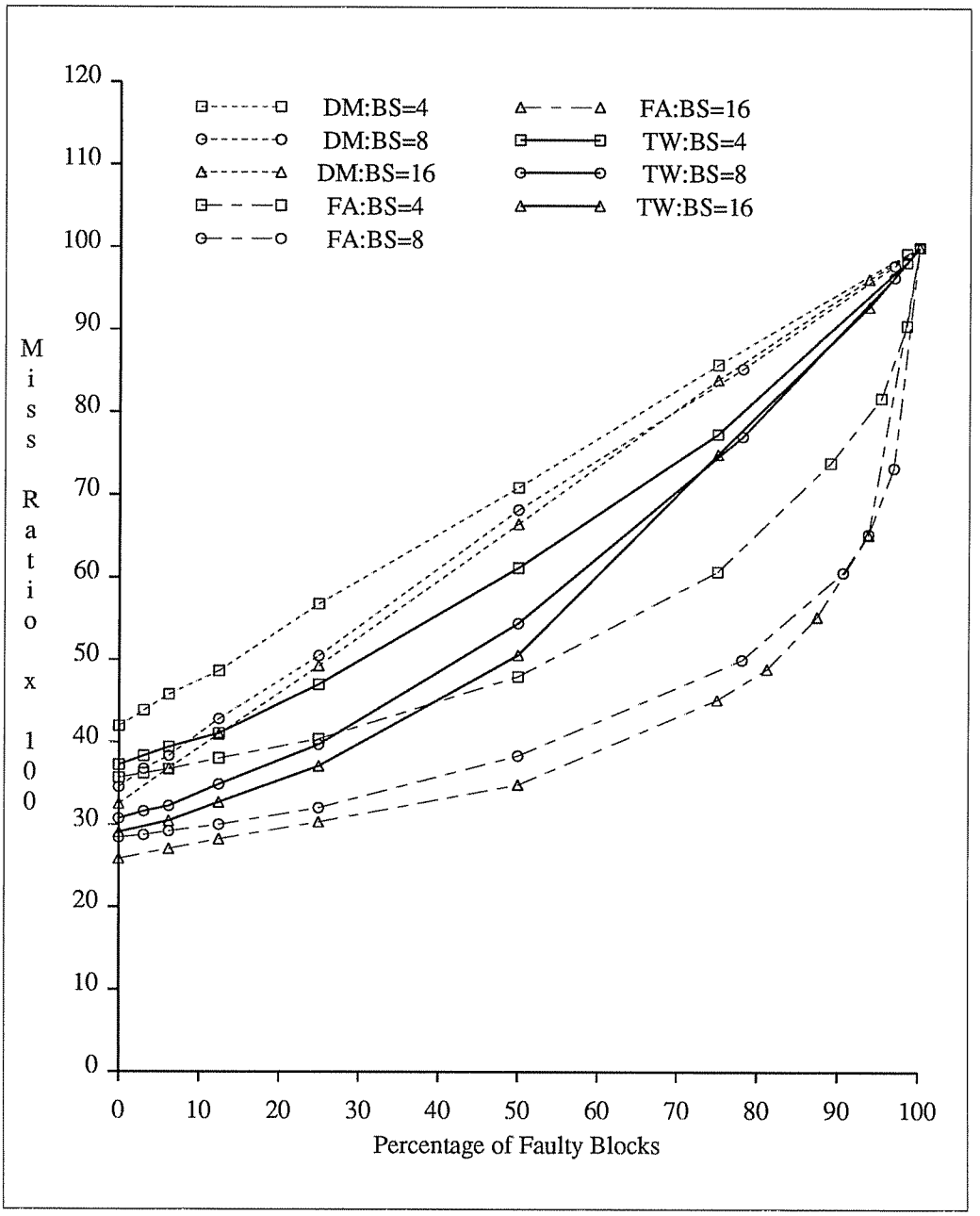

Figure 3: Miss Ratio for Cache Size = 256 bytes.

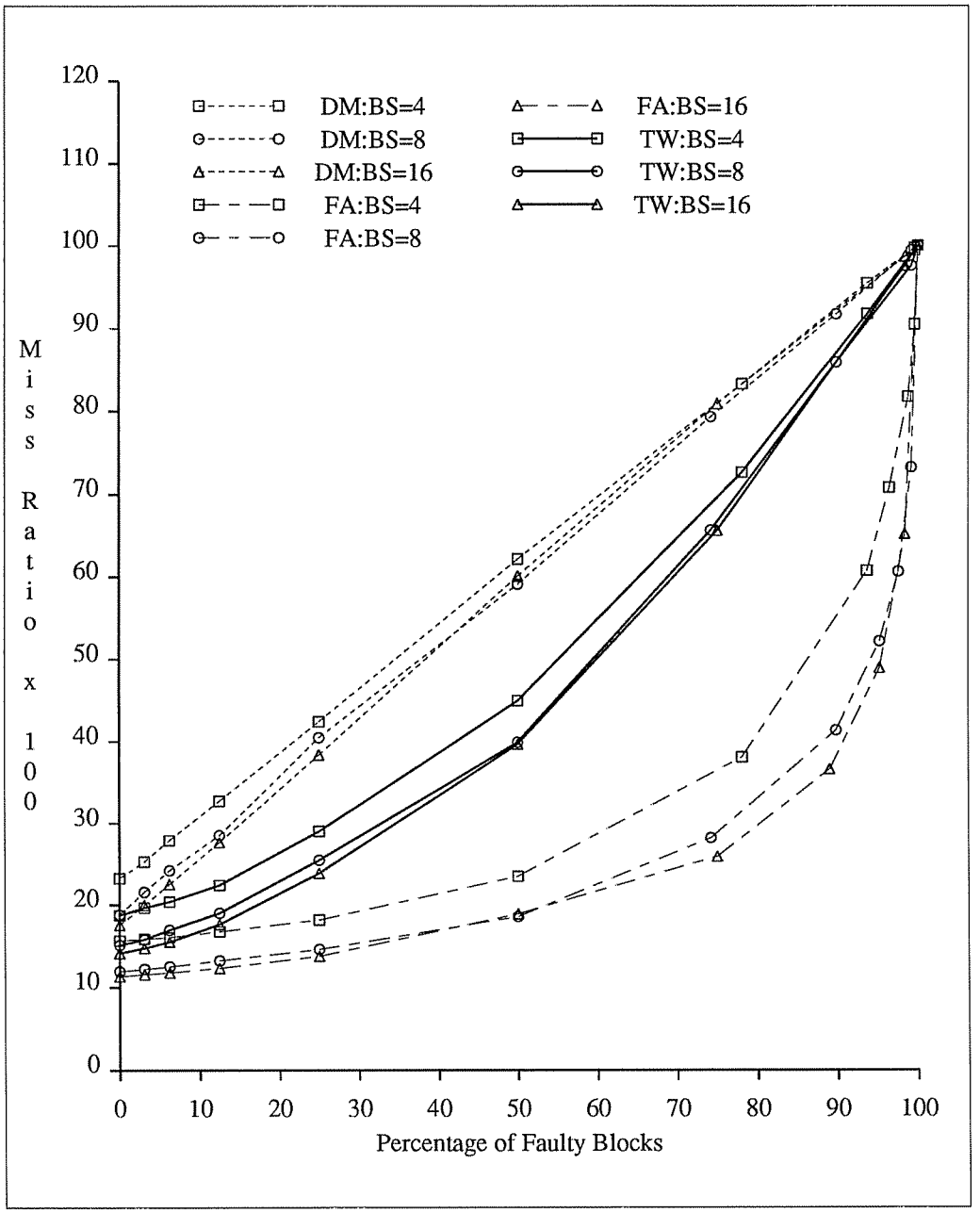

Figure 4: Miss Ratio for Cache Size = 1K bytes.

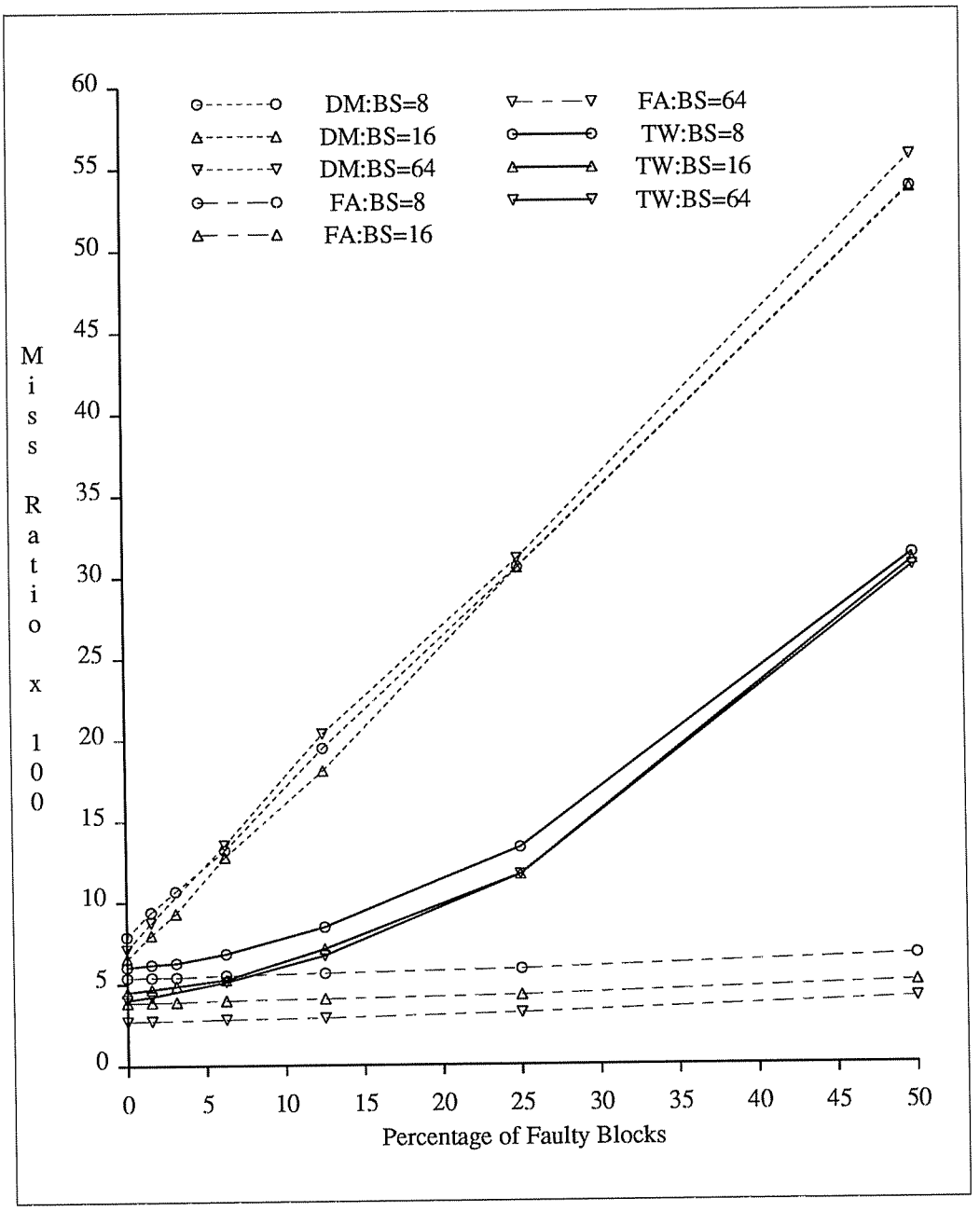

Figure 5: Miss Ratio for Cache Size = 8K bytes.

of such cache organizations significantly. Faults in the data array can, therefore, be tolerated by having smaller transfer blocks. To tolerate faults in the tag array, one might choose to have smaller address blocks and/or increase the set-associativity. If neither solution is acceptable, one could provide a limited amount of ECC in the tag array.

4. An Application of the Simulation Results

In this section, we apply the results of our study to the on-chip caches of two commercial single-chip processors: (i) the Motorola 68020[6] and (ii) the Zilog Z80,000 [11]. We are mainly interested in the cache organizations; other aspects of the cache are of secondary importance. Assume that the caches allow selective disabling of cache blocks (this is not true for either cache) and, therefore, each cache can function even in the presence of faulty blocks.

The MC68020 on-chip cache is an instruction-only cache. It is a direct mapped cache with 64 address blocks of 4 bytes each for a total of 256 bytes of on-chip cache. Each transfer block is also 4 bytes, i.e., the address and transfer blocks are of the same size. A single fault in either the tag or the data array, therefore, has the same effect on cache performance. The fault-free on-chip cache has an average hit ratio of 0.64 [18]. We expect that for each fault the hit ratio will decrease by 0.01. Clearly, a chip with a single fault in the cache need not be discarded if this decrease in hit ratio does not translate into a significant decrease in processor performance.

The Z80,000 cache is a fully associative cache [11]. It has 16 address blocks of 16 bytes each, again for a total of 256 bytes. The size of the transfer block is 4 bytes, i.e., there are 4 transfer blocks in each address block. The cache can be used to hold instructions only, data only or both instructions and data. It uses a write through policy to ensure the coherence of data. In spite of the large address block size, the cache is quite tolerant to cache tag faults because of its associativity. A single fault in the tag array would degrade the hit ratio from about 0.741 to about 0.729 (assuming that the Z80,000 traces display the same behavior as the VAX traces). Since each data block has 4 transfer blocks, we expect the degradation due to a fault in the data array to be even less severe than a fault in the tag array. The use of ECC in either the tag or the data array may, therefore, be quite unnecessary. If, however, the cache were direct mapped, a single fault in the tag array would have degraded the hit ratio from 0.675 to 0.631. In such a case, if the use of associativity was not a viable option, one might have considered the use of ECC in the tag array.

5. Conclusions

In this paper, we have discussed the effects of faults on the performance of cache memories. Faults in cache memories can arise both due to defects in manufacturing and due to transient/permanent errors during normal operation. We discussed the nature of such faults, saw how such faults could be identified and evaluated the performance degradation of various cache organizations due to faults using a detailed simulation analysis. We saw that in most cases, caches can be organized so that the performance degradation of the cache due to faults is quite insignificant. In other cases, a restricted use of ECC might be appropriate. There are two major conclusions of this research: (i) by choosing an appropriate cache organization, one need not use ECC for the on-chip cache memory and (ii) one need not blindly discard processors that have some flaws in their on-chip caches. With a proper cache organization, such chips could still function correctly with possibly an insignificant loss in cache performance.

References

[1] D. J. Kuck and B. Kumar, "A System Model for Computer Performance Evaluation," *Proc. International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 187-199, March 1976.

[2] D. W. Archer, et al, "A CMOS VAX Microprocessor with On-Chip Cache and Memory Management ," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp. 849-852, October 1987.

[3] A. D. Berenbaum, et al, "CRISP: A Pipelined 32-bit Microprocessor 13-kbit of Cache Memory ," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp. 776-782, October 1987.

[4] P. W. Bosshart, et al, "A 553K-Transistor LISP Processor Chip ," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp. 808-819, October 1987.

[5] M. Horowitz, et al, "MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache ," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp. 790-799, October 1987.

[6] D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro*, pp. 101-118, August 1984.

[7] A. Patel, "An Inside Look at the Z80,000 CPU: Zilog's New 32-Bit Microprocessor," *Proceesings of the AFIPS National Computer Conference*, pp. 83-91, July 1984.

[8] D. K. Pradhan, *Fault Tolerant Computing: Theory and Techniques*. Englewood Cliffs, New Jersey: Prentice Hall, 1986.

[9] J. Yamada, "Selector-Line Merged Built-In ECC Technique for DRAM's ," *IEEE Journal of Solid-State Circuits*, vol. SC-22, pp. 868-873, October 1987.

[10] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.

[11] D. Phillips, "The Z80000 Microprocessor," *IEEE Micro*, pp. 23-36, December 1985.

[12] J. S. Liptay, "Structural Aspects of the System/360 Model 85 Part II: The Cache," *IBM Systems Journal*, vol. 7, pp. 15-21, 1968.

[13] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

[14] Mark Hill, et al, "Design Decisions in SPUR," *Computer*, vol. 19, pp. 8-22, Nov. 1986.

[15] C. G. Bell, "Multis: a New Class of Multiprocessor Computers," *Science*, vol. 228, pp. 462-467, April 1985.

[16] D. A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. Van Dyke, "Architecture of a VLSI Instruction Cache for a RISC," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 108-115, June 1983.

[17] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," in *Proc. 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, pp. 119-127, June, 1986.

[18] J. Rubinstein and D. MacGregor, "A Performance Analysis of MC68020-based Systems," *IEEE Micro*, pp. 50-70, December 1985.