Center for Reliable Computing

TECHNICAL REPORT

PADded Cache: A New Fault-Tolerance Technique for Cache Memories

Philip P. Shirvani and Edward J. McCluskey

00-6

(CSL TR # 00-802)

December 2000

Center for Reliable Computing
Gates Room # 239, MC 9020
Gates Building 2A
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract:

This report presents a new fault-tolerance technique for cache memories. Current fault-tolerance techniques for caches are limited either by the number of faults that can be tolerated or by the rapid degradation of performance as the number of faults increases. In this report, we present a new technique that overcomes these two problems. This technique uses a special *Programmable Address Decoder (PAD)* to disable faulty blocks and to re-map their references to healthy blocks. Simulation results show that the performance degradation of direct-mapped caches with PAD is smaller than the previous techniques. However, for set-associative caches, the overhead of PAD is primarily advantageous if a relatively large number of faults is to be tolerated. The area overhead was estimated at about 10% of the overall cache area for a hypothetical design and is expected to be less for actual designs. The access time overhead is negligible.

Funding:

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047.

Imprimaturi: Nirmal Saxena and Nahmsuk Oh

PADded Cache:
A New Fault-Tolerance Technique for Cache Memories

Philip P. Shirvani and Edward J. McCluskey

CRC Technical Report No. 00-6

(CSL TR No. 00-802)

December 2000

CENTER FOR RELIABLE COMPUTING

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

Abstract

This report presents a new fault-tolerance technique for cache memories. Current fault-tolerance techniques for caches are limited either by the number of faults that can be tolerated or by the rapid degradation of performance as the number of faults increases. In this report, we present a new technique that overcomes these two problems. This technique uses a special *Programmable Address Decoder (PAD)* to disable faulty blocks and to re-map their references to healthy blocks. Simulation results show that the performance degradation of direct-mapped caches with PAD is smaller than the previous techniques. However, for set-associative caches, the overhead of PAD is primarily advantageous if a relatively large number of faults is to be tolerated. The area overhead was estimated at about 10% of the overall cache area for a hypothetical design and is expected to be less for actual designs. The access time overhead is negligible.

Key Words and Phrases: fault-tolerant architectures, high availability, RAS, cache memory, on-line repair, programmable address decoder, graceful degradation, cache performance, trace-driven simulation, yield enhancement.

1. INTRODUCTION

High levels of reliability, availability and serviceability (RAS) are key design goals of high-end computer systems. High availability is achieved by using high quality components (fault avoidance) and design techniques that allow on-line detection and recovery from hard and soft failures (fault tolerance). Minimizing the downtime is very important in systems with high availability. When continuous operation is required, an on-line repair mechanism disables or replaces a faulty component without human interaction and thereby eliminates the downtime for service. On-line repair is also required for systems that are not accessible for repair, as in space applications [Siewiorek 92]. In some fault-tolerance techniques, the system continues its operation at a possibly lower performance level (*graceful degradation*) under the presence of faults. In this report, we present a new technique for tolerating permanent faults in cache memories during their lifetime.

All high performance microprocessors use a hierarchy of cache memories to hide the slow access to the main memory [Hennessy 96]. With each new generation of integrated circuit (IC) technology, feature sizes shrink, creating room for more functionality on one chip. We see a fast growth in the amount of cache that designers integrate into a microprocessor to gain higher performance. Hence, caches occupy a substantial area of a microprocessor chip and contain a high percentage of the total number of transistors in the chip. Consequently, the reliability of the cache has a big impact on the overall chip reliability. The new fault-tolerance technique presented in this report addresses this issue by exploiting the architectural features of caches. We propose a reconfiguration technique that can keep the cache operational at a high level of performance (low miss rate) even in the presence of a large number of faults. Our technique provides a way to do the reconfiguration to isolate the faulty part; it assumes the system is equipped with appropriate error detection and error recovery mechanisms.

In Sec. 2, we review the previous work on fault-tolerance techniques for memories and caches. In Sec. 3, we present the *PADded* caches — caches with *Programmable Address Decoders*. To evaluate the performance of PADded caches, we carried out simulations. The results are explained in Sec. 4. The hardware overhead of PADded caches is estimated in Sec. 5. We discuss some implementation issues and the advantages of our technique in Sec. 6 and conclude in Sec. 7. Simulation results that were not presented in [Shirvani 99] are presented in the Appendix.

2. PREVIOUS WORK

Caches are used to bridge the speed gap between microprocessors and main memory. Without them, the processor can still operate correctly but at a substantially lower performance. Many architectures are designed such that the processor can operate without a cache under certain circumstances. Therefore, the obvious solution to a faulty cache is to totally disable it. In set-associative caches, a less extreme solution is to disable one unit (way) of the cache [Ooi 92].

Most of the transistors in a cache are in memory cells. Hence, the probability that a given defect is in a memory cell (a bit in the data or tag field) is higher than the probability of it being in the logic circuitry. If the fault is in a data or tag bit, only the block containing that bit needs to be disabled. For marking a cache block as faulty, an extra bit can be added to the set of flag bits associated with each block [Patterson 83]. If this bit is zero, the block is non-faulty and will do its normal function. If the bit is one, it will indicate that the block is faulty. In case of a direct-mapped cache, if the faulty block is accessed, it will always cause a cache miss. In case of a set-associative cache, the associativity of the set that includes that block is reduced by one. In this report, we refer to this extra bit as the *FT-bit* (fault-tolerance bit). Other names used in literature are: *availability bit* [Sohi 89], *purge bit* [Luo 94], and the *second valid bit* [Pour 93]. Adding the FT-bit was initially suggested for yield enhancement [Patterson 83]. Yield models have been derived in [Nikolos 96] for processors with partially good on-chip caches—chips with up to R faulty cache blocks. It is shown that maximum yield can be achieved with small R.

Disabling faulty blocks is done even when the cache is protected by *single-error correcting, double-error detecting* (SEC-DED) codes [Turgeon 91]. For example, *Cache Line Delete* (CLD) is a self-diagnostic feature in high-end IBM™ processor caches that automatically detects and deletes the cache blocks with permanent faults [O'Leary 89].

A detailed investigation of the effect of faulty cache blocks on miss rates was first presented in [Sohi 89] and then extended in [Pour 93]. Cache blocks were randomly chosen and marked as faulty. It was shown that for direct-mapped caches, the miss rate increases linearly with the fraction of faulty blocks. For a set-associative cache, the increase is slow for a few faults and becomes faster as the number of faults increases.

Replacement techniques, such as extra rows and columns, are also used in caches for yield enhancement [Youngs 96] and for tolerating lifetime failures [Turgeon 91] [Houtt 97]. With replacement techniques, there is no performance loss in caches with faults. However, the number of extra resources limits the number of faults that can be tolerated using these techniques. A replacement scheme called the *Memory Reliability Enhancement Peripheral* (MREP) is presented in [Lucente 90]. The idea is to have a set of spare words

each of which can replace any faulty word in the memory. A technique similar to MREP is presented for caches in [Vergos 95]. A very small fully associative spare cache is added to a direct-mapped cache that serves as a spare for the disabled faulty blocks. The difference between this technique and MREP is that there can be more faulty blocks than there are spare blocks. The simulation results in [Vergos 95] show that one or two spare blocks are sufficient to avoid most of the extra misses caused by a few (less than 5%) faulty blocks. However, as the number of faults increases, a few spare blocks is not very effective. Cache memories have an inherent redundancy that can be exploited to increase this limit.

3. PADDED CACHES

In this section, we present a new technique that has much less performance loss in caches as the number of faulty blocks increases. A good measure of microprocessor performance is *CPI*, average number of cycles per instruction [Hennessy 96]. For a non-faulty cache with miss rate (misses per memory references) *m*:

$$CPI = CPI_{execution} + \frac{Memory\_accesses}{Instruction} \times m \times Miss\_penalty$$

where $CPI_{execution}$ is the average number of cycles for executing an instruction without any memory stall and *Miss_penalty* is the average extra cycles needed for accessing the subsequent memory hierarchy levels. Since *Miss_penalty* is usually an order of magnitude bigger than $CPI_{execution}$, a small change in *m* will cause a big change in *CPI*. Therefore, it is desirable to minimize the increase of miss rate under the presence of faults.

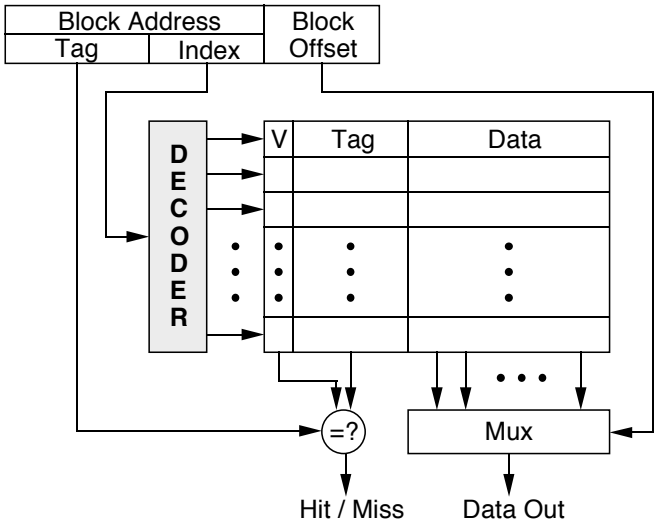


Figure 3.1 Block diagram of a direct-mapped cache.

Figure 3.1 shows a simple block diagram of a direct-mapped cache. The mapping of block addresses to the blocks in the cache is done by a decoder which is shown as a dark rectangle in the diagram. We modify this decoder to make it programmable such that it can

implement different mapping functions suitable to our technique. No spare blocks are added for tolerating faults. Instead, we exploit the existing non-faulty blocks as substitutes for the faulty ones.

Figure 3.2 shows part of the last three stages of a simple decoder. This type of decoder may not be used in speed-critical circuits. However, it is easier to illustrate our re-mapping procedure using this decoder. The same procedure applies to decoders that use gates instead of pass transistors as will be shown later.

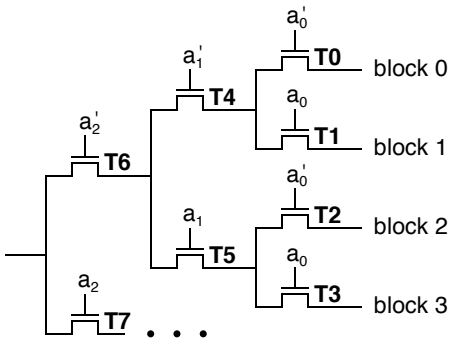


Figure 3.2 Circuit diagram of a simple decoder.

In Fig. 3.2, each *block i* output is a word line that selects a block in the cache. For example, when $a_2a_1a_0$=000, *block 0* is selected, when $a_2a_1a_0$=001, *block 1* is selected, and so forth. Now assume that *block 0* is faulty. We modify the control inputs of transistors *T0* and *T1* such that, when *block 0* is marked faulty, *T0* is always off and *T1* is always on. That is, all the references to *block 0* are re-mapped to *block 1*. Therefore, the addresses that map to *block 0* are still cacheable and will only suffer from conflict misses with *block 1*. Since one address bit information is lost due to this re-mapping, one bit is augmented to the tag bits to distinguish the addresses that may be mapped to the same block when faults are present. Similarly, if *block 1* is faulty, *T0* will be always on, *T1* will be always off, and all the references to *block 1* are re-mapped to *block 0*. Figure 3.3 shows the modified decoder.

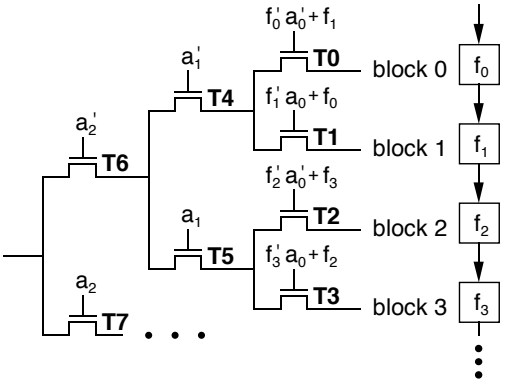


Figure 3.3 A simple Programmable Address Decoder (PAD).

Similar to the FT-bit, an extra bit is added to each block to mark it as faulty or non-faulty (the f_i's in Fig. 3.3). These flip-flops can be connected as a shift register and loaded using special instructions. The control inputs of the pass transistors are modified as shown. For example, the control for *T0* is changed from a_0' to $f_0' \cdot a_0' + f_1$. Therefore, *T0* is always on if *block 1* is faulty (f_0=0, f_1=1) and is always off if *block 0* is faulty (f_0=1, f_1=0).

We call two blocks *adjacent* if their addresses differ only in the address bit that is decoded last. For example, in Fig. 3.2, *block 0* and *block 1* are adjacent and *block 2* and *block 3* are adjacent. Depending on the reliability requirement of the application, the probability of adjacent faulty blocks may be low enough that the programmability is done only for the last level as shown in Fig. 3.3. However, the scheme can be applied to multiple levels to tolerate adjacent faulty blocks and to meet the desired level of fault tolerance. For example, the control of transistor *T4* can be changed to $g_0' \cdot a_1' + g_1$, where $g_0 = f_0 \cdot f_1$ and $g_1 = f_2 \cdot f_3$. In a case where both *block 2* and *block 3* are faulty (g_0=0 and g_1=1), *T4* and *T5* will always be on and off, respectively. Therefore, the references to blocks *2* and *3* will be re-mapped to blocks *0* and *1*, respectively. In this case, the tag portion has two extra bits. There are two points to notice here. In the last example, transistors *T2* and *T3* will be always on. However, this will not cause any problem because *T5* will be off and hence blocks *2* and *3* will never be selected. Second, if one of the blocks *0* or *1* is also faulty, the corresponding f_i will be 1 and all the references to the 3 faulty blocks will be re-mapped to the one healthy block. The same procedure can be applied to all levels.

This technique can be similarly applied to a set-associative cache. Typically, in a physical design, there is a separate memory array for each way of a set-associative cache. The decoder can be shared between the arrays, or it may be duplicated due to floor-planning or circuit issues. For example, consider a 4-way set-associative cache with one decoder for each way. Our technique can be independently applied to each decoder, i.e., re-mapping in one array will not affect mapping of the other arrays. Figure 3.4(a) illustrates a case where a set has a faulty block — we call this set the *faulty set*. With a PAD, the faulty block is mapped to a non-faulty block. We call the set that contains this non-faulty block, the *congruent set* of the faulty set (in case of a direct-mapped cache, each set has only one block). Because the re-mapping does not affect the other blocks in the set, one healthy block of the congruent set will be shared between the faulty set and the congruent set and the rest of the blocks are used as before. If there are two decoders, one for each pair of arrays, the healthy block that shares a decoder with the faulty block will

also be marked faulty and two blocks will be shared between the two sets. This is illustrated in Fig. 3.4(b).

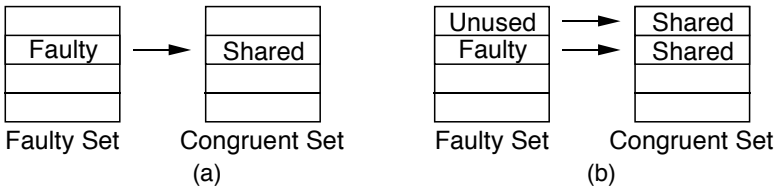


Figure 3.4 Block sharing between a faulty set and its congruent set in a 4-way set-associative cache: (a) with 4 decoders, (b) with 2 decoders.

The conflict misses that occur between a set and its congruent set can be reduced by special ordering of the address bits in the decoder. Caches exploit the spatial locality of the memory accesses. If we make a set and its congruent set far apart in the address space, there will be less conflict due this spatial locality. Since the order of the inputs to the decoder is not important in the function of the cache, we use the index bits in the reverse order. That is, the most significant bit (msb) of the array index is connected to the least significant input bit (lsb) of the decoder. The simulation results show that this design has better performance than a direct connection (normal order). As mentioned before, the tag portion of the array is widened by one bit for each level of programmability. These bits will have constant values when there is no faulty block and will start changing when the block becomes a substitute for a faulty block. Notice that these tag bits will store the same bits that are used for indexing the cache array. For example, if the cache is virtually-indexed physically-tagged, the normal tag bits will contain physical addresses and the extra tag bits will contain virtual addresses. The tag comparator is also widened by the same number of bits.

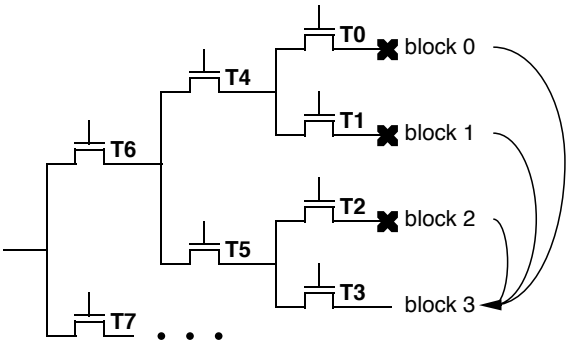


Figure 3.5 Mapping scheme 1 for three adjacent faults.

Figure 3.5 shows the mapping that occurs when there are three adjacent faulty blocks. In this example, blocks *0*, *1* and *2* are faulty. Through the programming logic (not shown in the figure), transistors *T2* and *T4* are always off and transistors *T3* and *T5* are always on. Therefore, all the references to the four blocks *0*, *1*, *2* and *3* are mapped to

block 3. To reduce the conflict misses of these references, we can assume that *block 3* is also faulty and program the decoder accordingly. The result mapping is what is shown in Fig. 3.6. This can distribute the traffic more evenly and may decrease the miss rate.

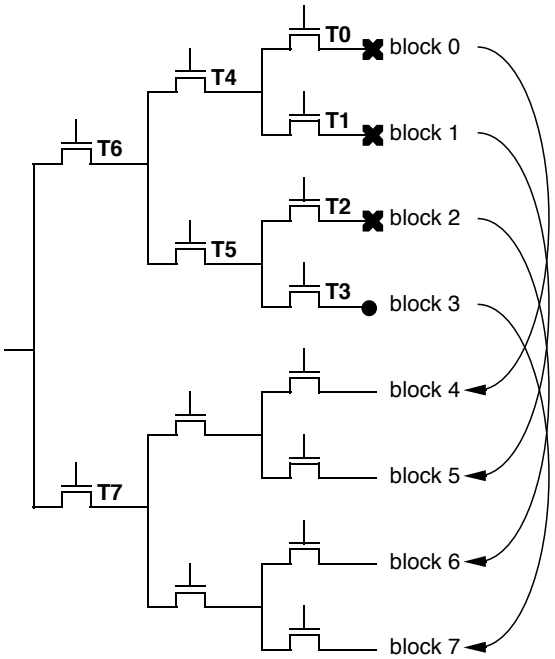


Figure 3.6 Mapping scheme 2 for three adjacent faults.

4. SIMULATION RESULTS

In this section, we discuss our simulation setup for evaluating the performance of PADded caches and present the results. Trace-driven simulation is the most popular method for evaluating the performance of cache memories. We used this method to compare the miss rates of caches with two fault-tolerance techniques, the simple block deletion using the FT-bit, and our new PAD technique. In the rest of this report, we refer to the first technique as FTB. We modified the Dinero IV cache simulator ([DineroIV 98]) to simulate these two techniques. We used different values for the following parameters: cache size (2, 4, 8, 16, and 32KB), block size (8, 16 and 32 bytes) and associativity (1, 2 and 4). Set-associative caches also have one array for the LRU (least-recently-used) bits that are used in the replacement protocol. In our technique, the decoder of this array and the replacement logic are not modified and we simulated them as such in the simulations. In the presence of faults, the shared blocks will be in different replacement order depending on which set is being accessed. Accesses to one set may replace a shared block that was the most-recently-used block in its congruent set. We did not observe any noticeable effect in the miss rates from this interference.

The simulated caches are write-through, allocate-on-write-miss, with no prefetching and no sub-blocking, and use the LRU replacement protocol. We assume that the decoders are replicated for each way of the cache, and they are programmable for all levels. The faults are injected at random locations and miss rates are calculated for different number of faulty blocks. Since the location of the faults affects the miss rate, each simulation was run several times and the minimum, maximum and average miss rates were recorded.

We collected many sets of traces for our simulations. Some of them are: the ATUM traces [Agarwal 86], traces from SPEC92 and SPEC95 benchmarks, and the IBS traces (also called the *Monster* traces) [Uhlig 95]. The total number of references in each trace set vary from about 5,000,000 for the ATUM traces, up to more than a billion for the IBS traces. The miss rate for one set of traces, e.g., the ATUM traces, is calculated by taking the weighted average of the miss rates of each trace in the set — weighted by the number of references in each trace.

In this section, we present a selection of the simulation results. More results and details are presented in the Appendix. The average miss rate of the ATUM traces is shown in Fig. 4.1 for both the FTB and PAD techniques. This graph shows the results for an 8KB cache with 16-byte blocks and with associativities: direct-mapped (DM), 2-way set-associative (SA2), and 4-way set-associative (SA4). The miss rate of PADded caches stays relatively flat and increases slowly towards the end.

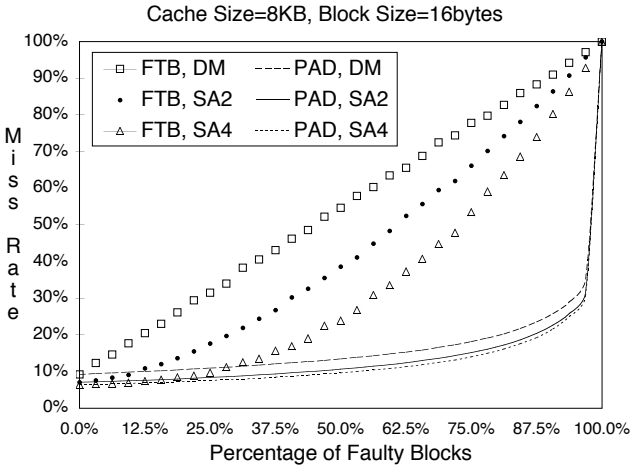


Figure 4.1 Average miss rates of the ATUM traces for different associativities.

Figure 4.2 shows the lower left corner of Fig. 4.1. Notice that for a small percentage of faulty blocks, both techniques perform equally well for the set-associative caches.

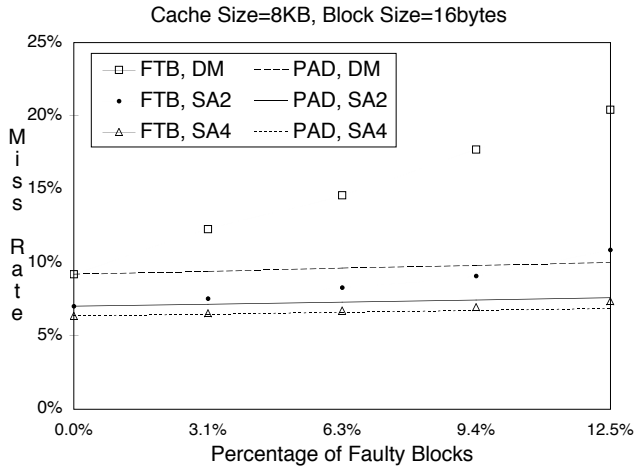


Figure 4.2 Average miss rates of the ATUM traces (lower left corner of Fig. 4.1).

Simulations of caches with different sizes show that when half of the blocks are faulty, the miss rate of a PADded cache is almost the same as a healthy cache of half the size. This can be seen in Fig. 4.3 where the miss rates of direct-mapped caches of different sizes is shown. The data labels are shown for a few points for easier comparison. For example, when half of the blocks in the 4KB cache are faulty, the miss rate is 17.8%. This is close to the miss rate of a healthy 2KB cache, which is 16.6%. The same behavior can be seen for an 8KB cache (the dark triangle in the figure). This indicates that in PADded caches, the full capacity of healthy blocks is utilized and the performance decreases with almost minimum possible degradation.

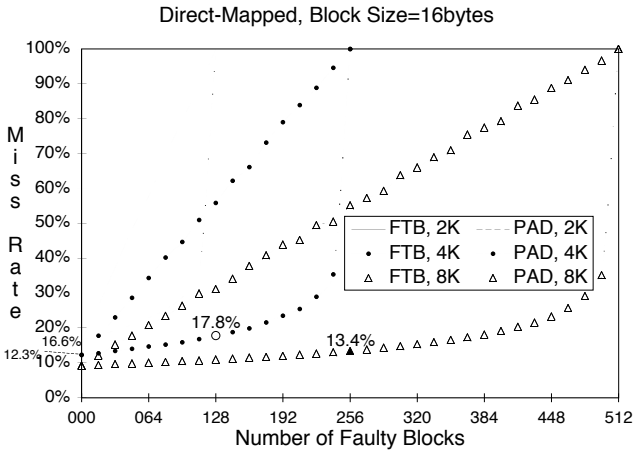


Figure 4.3 Average miss rates of the ATUM traces for different cache sizes.

As discussed in Sec. 3, for PADded caches, the order of the input bits to the decoder can be selected between normal order (lsb first) or reverse order (msb first). The effect of this option is shown in Fig. 4.4. It is clear that the reverse order has a lower miss rate as expected and should always be the choice. Therefore, in all the other graphs shown in this report, the reverse order is used for PAD.

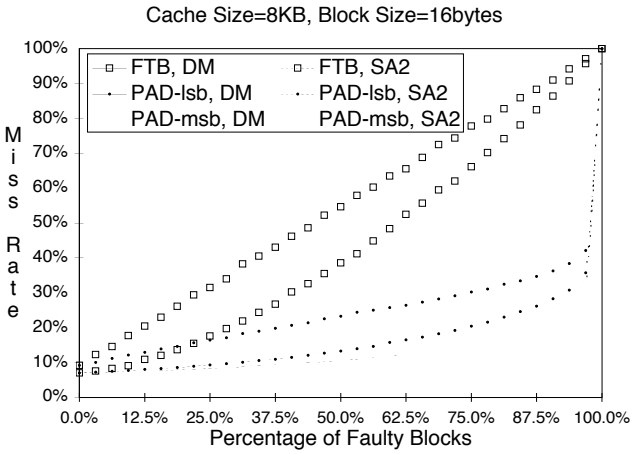


Figure 4.4 Average miss rate of the ATUM trace for different ordering of decoder bits in PAD.

Another advantage of our technique is shown in Fig. 4.5. This figure shows the minimum and maximum miss rates of the ATUM traces for different fault locations, in a direct-mapped cache. The miss rate of FTB has a noticeably big range (as was shown in [Pour 93]) while the miss rate of PAD remains almost unchanged. This shows that PADded caches are very insensitive to the location of faults. Notice that we did not do an exhaustive search for the minimum and maximum miss rates; these are the ranges observed in our simulations.

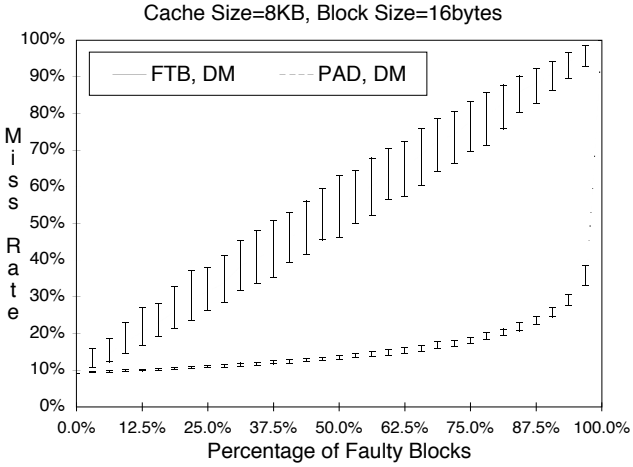


Figure 4.5 Minimum and maximum miss rates of the ATUM traces for a DM cache.

We also simulated a split cache architecture consisted of a 16KB instruction cache and 16KB data cache, and observed a small difference in the behavior of the two caches for the ATUM traces. The difference in the miss rates of FTB and PAD becomes noticeable for a smaller number of faults in the data cache than it does in the instruction cache. This shows that using PAD is more important in data caches.

5. HARDWARE OVERHEAD

In this section, we look at the circuit implementation of PADded caches to estimate the area and delay overhead of our technique. A typical decoder for a cache is implemented using several pre-decode stages and logic gates. Let us consider a small decoder for the sake of simplicity. For example, a 4-to-16 bit decoder is shown in Fig. 5.1. To emphasize the importance of using the msb of the address bits for the last stage of decode, the address bits are used in the reverse order in this example ($a_3a_2a_1a_0$, is the real index address with a_3 as the msb). Similar to the decoder in Sec. 3, we intercept the address bits to disable or enable the outputs of the gates. The modified decoder is shown in Fig. 5.2.

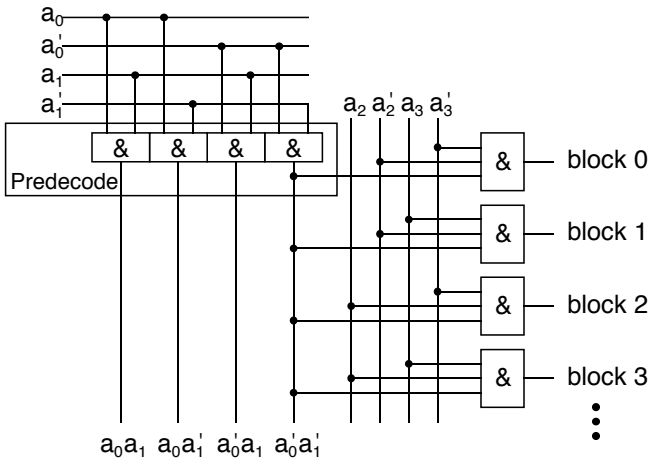


Figure 5.1 A decoder implemented by logic gates.

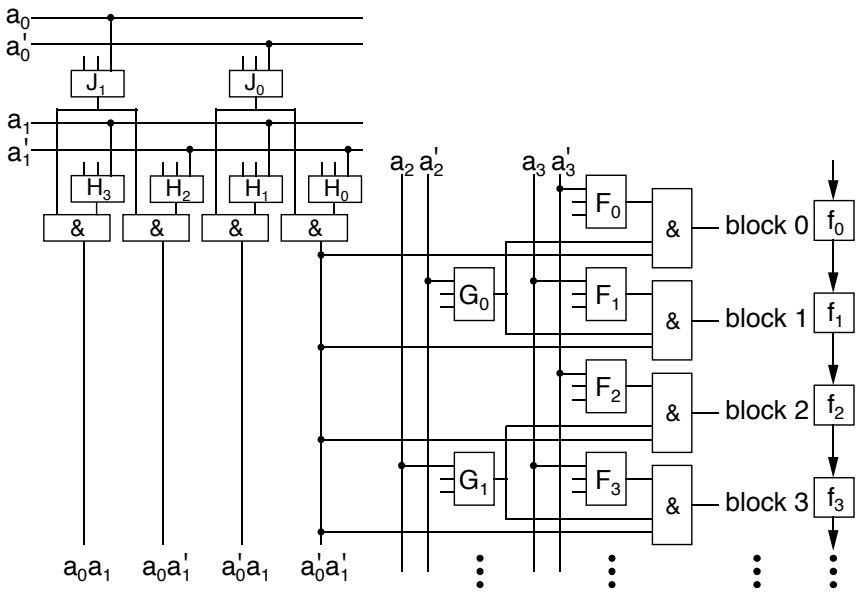


Figure 5.2 Modified version of the decoder in Fig. 5.1 for PAD.

The three input gates F_i, G_i and H_i implement functions similar to those in Fig. 3.3:

$$F_0 = f_0' \cdot a_3' + f_1 \text{ , } F_1 = f_1' \cdot a_3 + f_0 \text{ , } F_2 = f_2' \cdot a_3' + f_3 \text{ ,}$$
$$F_3 = f_3' \cdot a_3 + f_2 \text{ , } G_0 = g_0' \cdot a_2' + g_1 \text{ , } G_1 = g_1' \cdot a_2 + g_0 \text{ ,}$$
$$H_0 = h_0' \cdot a_1' + h_1 \text{ , etc.}$$

where $g_0 = f_0 \cdot f_1$, $g_1 = f_2 \cdot f_3$, $h_0 = g_0 \cdot g_1$ and $h_1 = g_2 \cdot g_3$. If only one level of programmability is required, then only the F_i functions would be required. Notice that the f_i, g_i, h_i and their complements change only during reconfiguration and are constant the rest of the time. Therefore, the corresponding gates (NANDs and inverters) can use minimum size transistors. Furthermore, these constant values form two of the three inputs of the functions F_i, G_i and H_i. The gates for F_i, G_i and H_i add one extra logic stage to the decoder and are in the critical path for accessing the cache. To reduce the gate delay of these AOI (AND-OR-Invert) functions, the transistors that are connected to the constant inputs are made 3 to 4 times the size of the transistor connected to the variable input. Figure 5.3(b) shows the transistor sizes of an AOI gate that implements function F_0, relative to the inverter in Fig. 5.3(a) —notice that only the transistors on the critical path are enlarged. The delay overhead of adding these gates can be made negligible by choosing proper sizes for the transistors in the decoding stages.

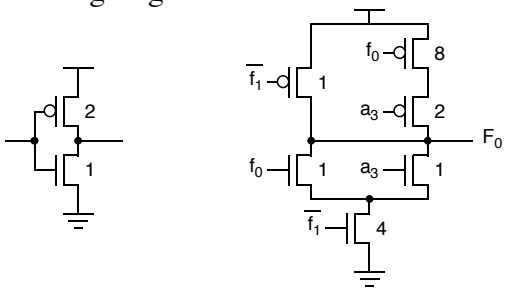


Figure 5.3 Transistor sizing for the F_i gates: (a) an inverter, (b) an AOI gate with a delay close to the inverter in (a).

Since the number of tag bits increases in PAD, the width of the tag comparator also increases. However, the delay of the comparator will increase by at most one gate delay. Therefore, the delay overhead of adding PAD to a cache is close to one gate delay of an inverter driving an identical inverter.

For a cache with 2^n blocks, there are 2^n flip-flops for the FT-bits (with area A_{ff} each). These flip-flops are formed into a scan chain and their values are shifted in serially during reconfiguration. For k levels of programmability, there are k ($k \le n$) extra tag bits for each block (with area A_b each). The total area overhead of one decoder will be:

$$A_{overhead} = S(n-1,k) \times A_g + S(n,k) \times A_G + 2^n \times A_{ff} + 2^n \times k \times A_b + k \times A_{comp}$$

where $S(n,k) = \sum_{i=0}^{k-1} 2^{n-i}$, A_g is the area of a NAND and an inverter implementing one f_i (g_i, etc.) and its complement, A_G is the area of the gates implementing the F_i (G_i, etc.) functions, and A_{comp} is the area per bit of the tag comparator (this last term is negligible).

For example, let us consider a 16KB direct-mapped cache with 16 byte blocks. A typical decoder for this cache takes about 5% of the total cache area. We estimated that the decoder would grow by 60% after modifications for PAD. That is 3% increase in total cache area. With a 32-bit address, the tags will be 18 bits wide (assume all addresses are physical). Considering two bits for flags, each block will be 16×8+18+2=148 bits long. For a PADded cache with full programmability, 10 bits will be added to each block, which is a 7% increase. The FT-bits will roughly take another extra 1%. Therefore, the total area overhead is 3+7+1=11%. Most of this overhead is in the extra tag bits. Therefore, a designer may decide to make just a few of the levels programmable.

Another way to reduce the area overhead, is to make the decoder programmable starting at the decoding level before the last. An FT-bit is added for each pair of blocks, i.e., each two blocks are disabled together, regardless of whether they are both faulty or only one is faulty. This will reduce the three first terms of the above formula by about half, and decrease the k in the last two terms, to k-1. As can be seen, our technique is very flexible and the overhead can be adjusted according to the reliability requirements.

In a cache with multiple independent ports, there are as many decoders as there are ports. These decoders should provide the exact same mapping in the presence of faults. Therefore, the FT-bits and the f_i (g_i, etc.) functions can be shared between the decoders and consequently, the overall area overhead will be less for this type of cache.

6. DISCUSSION

In this section, we discuss some implementation issues and other advantages of our technique over FTB. The advantage of FTB over PAD is low hardware overhead (we don't have overhead numbers for FTB and other techniques to do a better comparison). We initially assumed that an extra bit (the FT-bit) is added to the flags of each cache block. This bit will be an extra input to both the tag matching logic and the replacement logic. In the tag matching logic, FT-bit=1 will always produce a mismatch signal (in case of a possible match in the block number). In the replacement logic, it will prohibit this line from being chosen as the line to be replaced (for set-associative caches). These two simple modifications, will prevent the line from being used in case of a fault. Adding an extra bit to each cache block will impose a relatively small area overhead. In some caches, this extra physical bit can be avoided by using an unused combination of the available flag bits in

cache. For example, in a write-back cache, the combination *dirty*=1 and *valid*=0 can be used to mark a block as faulty.

The FTB technique relies on the availability of a path to bypass the cache when a faulty set is being accessed. Some cache organizations require blocks to be loaded into the cache before they are read by the CPU [Patterson 83]. With FTB, a data that resides in an address that is mapped to a set with all faulty blocks, does not have a place to go in the cache and has to bypass the cache. The same problem exists for write-back caches where store instructions write to the data cache and the new data is written into the memory when the block is replaced. For a faulty set, the store has to be executed in a write-through fashion. Similar problem exists for caches with allocate-on-write-miss policy. Therefore, hardware should be added to make the references to the faulty sets behave like references to non-cacheable addresses. However, with PAD, the data is always cacheable as long as there is a non-faulty block in the cache, so there is no need for extra hardware.

Our simulation results show that PAD is most beneficial for direct-mapped caches. For a small percentage of faulty blocks, both FTB and PAD perform equally well for set-associative caches. The acceptable number of faults depends on the acceptable performance loss and can be determined from the graph in Fig. 4.1 and the equation for *CPI* (given in Sec. 3). This acceptable number of faults should be greater than the expected number of faults during the minimum lifetime for which the chip is designed. The expected number of faults depends on the failure rate of individual blocks and the probability of adjacent faults. Faults were injected at random in our simulations, modeling independence between the blocks. However, the failure mechanisms may be such that the probability of adjacent faults is higher than that depicted by this model. In such case, the performance of set-associative caches will degrade faster and PAD may be chosen over FTB for a relatively lower number of faults.

7. CONCLUSION

In this report, a novel fault-tolerance technique for caches is introduced. This technique uses graceful degradation to tolerate permanent faults in cache blocks. It can be used for on-line repair in systems for high reliability and availability. The main advantage of our technique is its slow degradation of performance as the number of faults increases. This increases the lifetime of a chip and subsequently, the availability of a system, because there will be less downtime for component replacement.

The PAD technique can also be used for yield enhancement. However, as mentioned earlier, the analysis in [Nikolos 96] shows that maximum yield can be achieved by accepting a few faulty blocks. The FTB technique can provide acceptable performance for

a few faulty blocks. Therefore, if yield enhancement is the goal, FTB is probably the better choice. The PAD technique provides a better solution when: the total number of faulty blocks that have to be tolerated (i.e., the manufacturing faults and the faults that occur during the lifetime of the cache) is more than a small percentage of total number of blocks, sustaining the peak performance is very critical, or the performance should be predictable. The last case is justified by the fact that FTB has a big range for the miss rate depending on the location of the faulty block, but PAD has a very small range. In real-time applications, the system has to execute a specified program within a certain amount of time, i.e., the performance has to be predictable to a specified level. A fault-tolerant cache that can maintain its estimated miss rate for that specified program under different conditions, will be the better choice.

We estimated the area overhead of PAD for a hypothetical design at about 10%; the access time overhead was negligible. However, the area overhead depends on the parameters of the cache, the levels of programmability in the decoder, and layout optimization. Therefore, we expect the area overhead to be less for real designs.

Our technique provides a single solution for all types of caches with different policies. No extra hardware has to be added for write-back or allocate-on-write-miss caches. Application of PAD for caches that are used in multi-level cache systems and in multi-processor systems is an area for future research.

ACKNOWLEDGMENTS

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047. The authors wish to thank Nirmal Saxena, Samy Makar, Subhasish Mitra, Mehrdad Heshami and David Harris for their reviews and valuable comments. The authors also thank Timothy Slegel for helpful discussions.

REFERENCES

[Agarwal 86] Agarwal, A., R.L. Sites, M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proc. 13th Annu. Symp. Comput. Architecture*, pp. 119-127, June 1986.

[DineroIV 98] "Dinero IV Trace-Driven Uniprocessor Cache Simulator", http://www.cs.wisc.edu/~markhill/DineroIV/, Rel. 7, Feb. 1998.

[Hennessy 96] Hennessy, J.L., and D.A. Patterson, *Computer Architecture, A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Pub., Inc., San Mateo, CA, 1996.

[Houtt 97] Houtt, W.V., et al., "Programmable Computer System Element with Built-In Self-Test Method and Apparatus for Repair During Power-On," *U.S. Patent 5,659,551*, Aug. 1997.

[Lucente 90] Lucente, M.A., C.H. Harris and R.M. Muir, "Memory System Reliability Improvement Through Associative Cache Redundancy," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 19.6.1-19.6.4, May 1990.

[Luo 94] Luo, X. and J.C. Muzio, "A Fault-Tolerant Multiprocessor Cache Memory," *Proc. IEEE Workshop on Memory Technology, Design and Testing*, pp. 52-57, August 1994.

[Nikolos 96] Nikolos, D. and H.T. Vergos, "On the Yield of VLSI Processors with On-Chip CPU Cache," *Proc. 2nd European Dependable Computing Conference*, pp. 214-229, October 1996.

[O'Leary 89] O'Leary, B.J., A.J. Sutton, "Dynamic Cache Line Delete," *IBM Tech. Disclosure Bull.*, Vol. 32, No. 6A, pp. 439, Nov. 1989.

[Ooi 92] Ooi, Y., M. Kashimura, H. Takeuchi, and E. Kawamura, "Fault-Tolerant Architecture in a Cache Memory Control LSI," *IEEE J. of Solid-State Circuits*, Vol. 27, No. 4, pp. 507-514, April 1992.

[Patterson 83] Patterson, D.A., et al., "Architecture of a VLSI Cache for a RISC," *Proc. Int'l Symp. Comp. Architecture*, Vol. 11, No. 3, pp. 108-116, June 1983.

[Pour 93] Pour, A.F. and M.D. Hill, "Performance Implications of Tolerating Cache Faults," *IEEE Trans. Comp.*, Vol. 42, No. 3, pp. 257-267, March 1993.

[Shirvani 99] Shirvani, P.P., and E.J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories," *17th IEEE VLSI Test Symp.*, pp. 440-445, Dana Point, CA, April 24-29,1999.

[Siewiorek 92] Siewiorek, D.P., and R.S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2nd Edition, Digital Press, Burlington, MA, 1992.

[Sohi 89] Sohi, G. S., "Cache Memory Organization to Enhance the Yield of High-Performance VLSI Processors," *IEEE Trans. Comp.*, Vol. 38, No. 4, pp. 484-492, April 1989.

[Turgeon 91] Turgeon, P.R., A.R. Stell, M.R. Charlebois, "Two Approaches to Array Fault Tolerance in the IBM Enterprise System/9000 Type 9121 Processor," *IBM J. Res. Develop.*, Vol. 35, No. 3, pp. 382-389, May 1991.

[Uhlig 95] Uhlig, R., et al., "Instruction Fetching: Coping with Code Bloat," *Proc. 22nd Int'l Symp. Comp. Architecture*, pp. 345-356, June 1995.

[Vergos 95] Vergos, H.T., and D. Nikolos, "Performance Recovery in Direct-Mapped Faulty Caches via the Use of a Very Small Fully Associative Spare Cache," *Proc. Int'l Comp. Performance and Dependability Symp.*, pp. 326-332, April 1995.

[Youngs 96] Youngs, L., G. Billus, A. Jones, and S. Paramanandam, "Design of the UltraSPARC™-I Microprocessor for Manufacturing Performance," *Proc. of the SPIE*, Vol. 2874, pp. 179-186, 1996.

APPENDIX

As mentioned in Sec. 4, we collected many sets of traces for our simulations. The main ones are: the ATUM traces [Agarwal 86], traces from SPEC92 and SPEC95 benchmarks, and the *Instruction Benchmark Suite* (IBS) traces (also called the *Monster* traces) [Uhlig 95]. IBS attempts to expose the trend that current software-development practices produce applications that exhibit higher instruction-cache miss ratios than do the SPEC92 benchmarks. IBS includes two operating systems and eighteen application traces, nine per operating system. We used the traces for Ultrix version 3.1 operating system from DEC. Table A.1 describes the applications in this trace set.

Trace Set	Trace Name	Description
IBS	groff	GNU C++ implementation of the Unix nroff text formatting program, Version 1.09.
	gs	Ghostscript (version 2.4.1) distributed by the Free Software Foundation. Renders and displays a single postscript page with text and graphics in an X window.
	jpeg_play	The xloadimage (version 3.0) program written by Jim Frost. Displays two JPEG images.
	mpeg_play	mpeg_play (version 2.0) from the Berkeley Plateau Research Group. Displays 85 frames from a compressed video file.
	nroff	Unix text formatting program shipped with Ultrix 3.1.
	real_gcc	The GNU C compiler (version 2.6).
	sdet	A multiprocess, system performance benchmark which includes programs that test CPU performance, OS performance and I/O performance. From the SPEC SDM benchmark suite.
	verilog	Verilog-XL (version 1.6b) simulating the logic design of an experimental microprocessor.
	video_play	A modified version of mpeg_play that displays 610 frames from an uncompressed file.

Table A.1 Description of the traces in the IBS trace set.

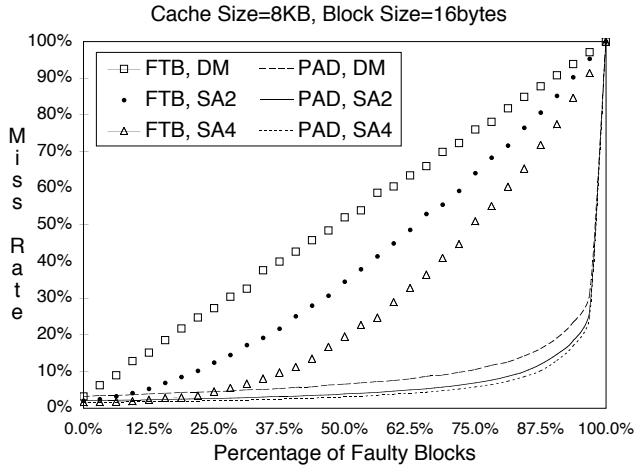
Table A.2 shows these four trace sets and the traces in each of them. This table also shows the total number of memory references in each trace. The memory references are further broken up into three types: instruction fetch, data read and data write. As can be seen in this table, the ATUM traces are relatively small and are suitable for simulations with a variety in cache parameters, some of which were presented in Sec. 4. The SPEC92 traces are moderate size and the SPEC95 and IBS traces are relatively large, presenting more realistic applications.

Race Set	Trace Name	Total # of References	Instruction Fetch	Data Read	Data Write
ATUM	dec0.000	361,982	183,023	106,459	72,500
	fora.000	387,934	199,799	108,979	79,156
	forf.003	368,212	190,915	107,969	69,328
	fsxzz.000	239,334	123,229	78,265	37,840
	ivex.000	341,968	203,510	97,335	41,123
	lisp.000	291,390	169,786	99,080	22,524
	macr.000	342,828	188,702	96,904	57,222
	memxx.000	444,849	219,050	126,660	99,139
	mul2.000	372,104	208,434	92,722	70,948
	mul8.003	429,432	199,455	141,126	88,851
	pasc.000	422,090	193,025	123,708	105,357
	spic.000	446,701	223,706	136,316	86,679
	ue02.000	357,810	199,973	98,385	59,452
SPEC92	compress	10,000,001	8,098,771	1,225,824	675,406
	ear	10,000,000	7,362,383	1,697,326	940,291
	eqntott	10,000,015	7,776,877	1,691,531	531,607
	espresso.bca	10,000,000	8,266,728	1,402,312	330,960
	gcc	10,000,002	8,019,022	1,394,620	586,360
	li	10,000,000	7,257,462	1,910,650	831,888
	sc	10,000,000	7,963,384	1,730,420	306,196
	swm	10,000,000	8,099,539	1,388,593	511,868
	tomcatv	10,000,000	7,378,857	2,157,289	463,854
SPEC95	compress	96,369,385	90,099,468	5,103,419	1,166,498
	gcc	216,257,596	198,098,561	12,171,028	5,988,007
	go	169,253,578	155,816,698	9,032,047	4,404,833
	ijpeg	369,783,397	339,027,179	19,573,014	11,183,204
	li	252,113,195	230,094,088	15,948,457	6,070,650
	perl	280,043,263	256,100,152	15,539,261	8,403,850
	vortex	247,931,137	230,273,353	12,426,174	5,231,610
IBS	groff	175,663,392	140,303,571	23,254,450	12,105,371
	gs	193,598,869	155,844,922	24,124,158	13,629,789
	jpeg_play	204,879,521	177,975,957	19,369,384	7,534,180
	mpeg_play	169,328,731	134,379,393	22,031,813	12,917,525
	nroff	195,822,738	163,036,056	22,873,679	9,913,003
	real_gcc	176,136,386	141,755,377	22,539,888	11,841,121
	sdet	67,312,658	54,682,235	7,386,723	5,243,700
	verilog	80,091,471	63,573,357	9,967,581	6,550,533
	video_play	92,278,365	72,393,212	11,221,937	8,663,216

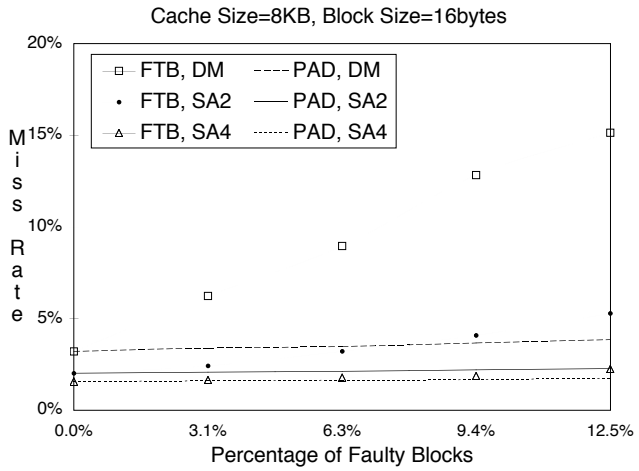
Table A.2 Summary of number of memory references in the traces used in simulations.

We used a 16KB cache for the simulation of SPEC95 and IBS traces. Figures A.1, A.2 and A.3, show the simulation results of SPEC92, SPEC95 and IBS traces, respectively. As mentioned in Sec. 4, in each simulation, faults are injected in random locations. Since miss rates depend on the locations of faults each simulation is repeated a number of times and the average as well as the minimum and maximum observed miss rate are recorded. For SPEC92 traces, we repeated each simulation 10 times. Since SPEC95 and IBS simulations are very time-consuming, this number was limited to 5 for these traces. Since similar behavior was observed for all traces, we also skipped the 4-way set associative simulation for the long traces.

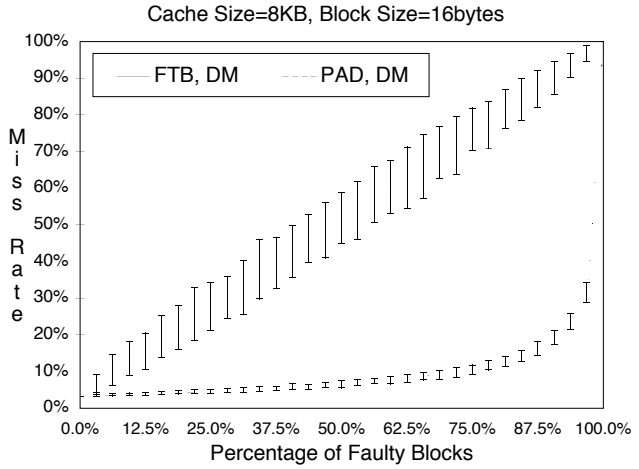
As the graphs in Fig. A.1 show, the miss rate of the SPEC92 traces starts at a lower percentage (no fault case) compared to the ATUM traces, but changes similarly. Comparing the miss rate of PADded caches in Fig. A.2(a) and Fig. A.3(a), we see a long flat section for the miss rate of the SPEC95 traces while for the IBS traces, the miss rate increases more quickly for low percentage of faulty blocks. This suggests that there is more conflict misses for the IBS traces due to references to faulty blocks and their congruent (healthy) blocks.



(a)

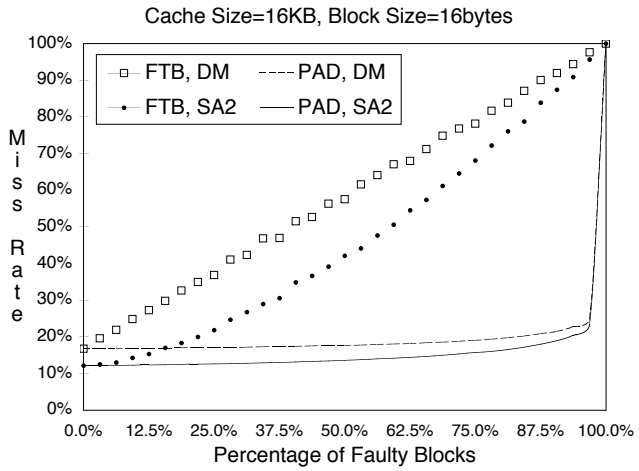


(b)

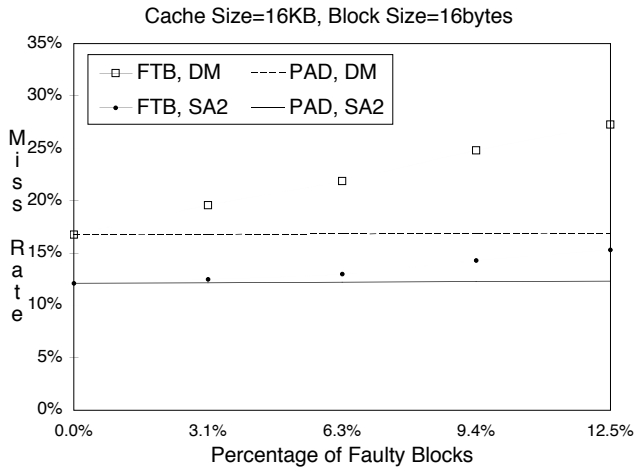


(c)

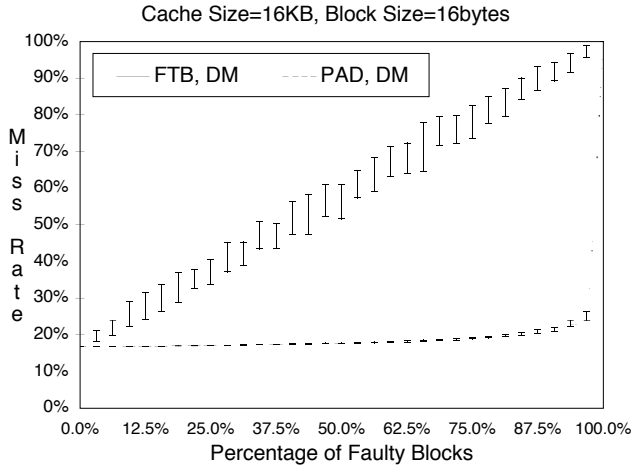
Figure A.1 Average miss rates of the SPEC92 traces: (a) different associativities, (b) lower left corner of Fig. A.1(a), (c) min. and max. miss rates for a DM cache.



(a)

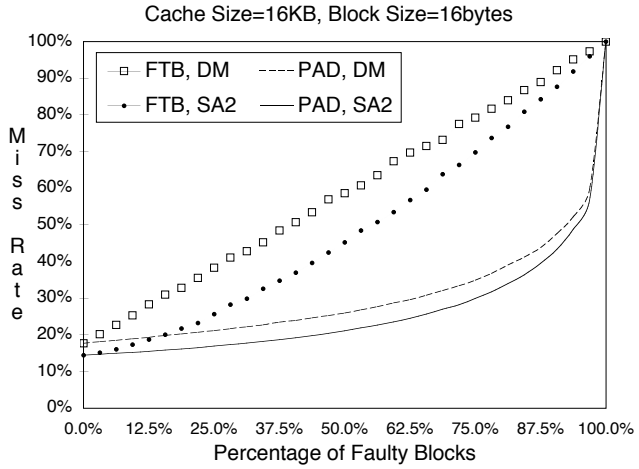


(b)

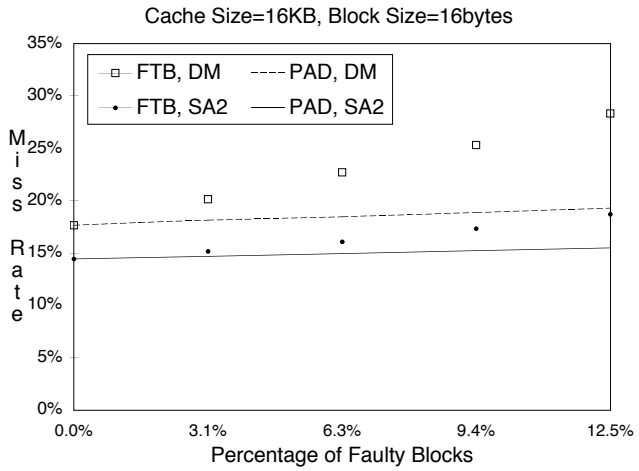


(c)

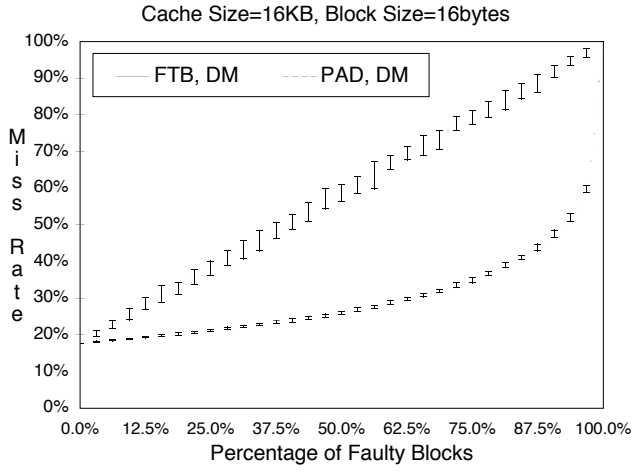
Figure A.2 Average miss rates of the SPEC95 traces: (a) different associativities, (b) lower left corner of Fig. A.2(a), (c) min. and max. miss rates for a DM cache.



(a)



(b)



(c)

Figure A.3 Average miss rates of the IBS traces: (a) different associativities, (b) lower left corner of Fig. A.3(a), (c) min. and max. miss rates for a DM cache.