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

Philip P. Shirvani and Edward J. McCluskey
Center for Reliable Computing, Stanford University

Abstract

This paper 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 paper, 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.*

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 [1]. 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 paper, 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 [2]. 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 paper 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.

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 [3].

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 [4]. 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 paper, we refer to this extra bit as the *FT-bit* (fault-tolerance bit). Other names used in literature are: *availability bit* [5], *purge bit* [6], and the *second valid bit* [7]. Adding the FT-bit was initially suggested for yield enhancement [4]. Yield models have been derived in [8] 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 [9]. 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 [10].

A detailed investigation of the effect of faulty cache blocks on miss rates was first presented in [5] and then extended in [7]. 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 [11] and for tolerating lifetime failures [9] [12]. 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 [13]. 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 [14]. 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 [14] 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. 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.

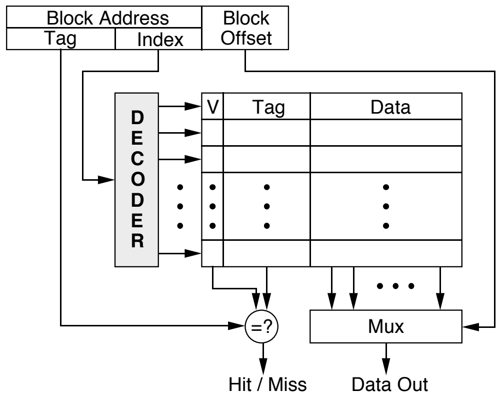


Figure 3.1 Block diagram of a direct-mapped cache.

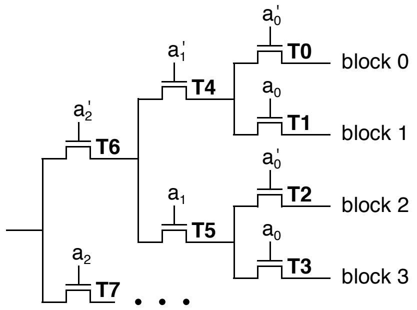


Figure 3.2 Circuit diagram of a simple decoder.

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.

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. 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).

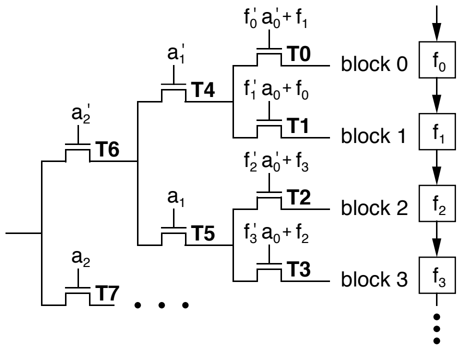


Figure 3.3 A simple Programmable Address Decoder (PAD).

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).

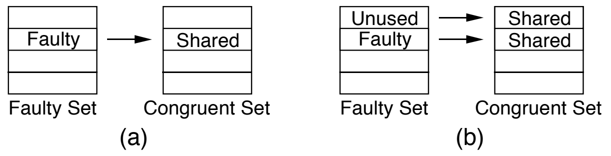


Figure 3.4 Block sharing between a faulty set and its congruent set in a 4-way set-associative cache: (a) with 4 decoders, (b) with 2 decoders.

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.

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 the paper, we refer to the first technique as FTB. We modified the Dinero IV cache simulator ([15]) 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). 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 [16], traces from SPEC92 and SPEC95 benchmarks, and the IBS traces (also called the *Monster* traces) [17]. The total number of references in each trace vary from about 300,000 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 paper, we present a selection of the simulation results. More results and details are presented in [18]. The average miss rate of the ATUM traces is shown in Fig. 4.1(a) 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. 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 indicates that in PADded caches, the full capacity of healthy blocks is utilized and the performance decreases with almost minimum possible degradation.

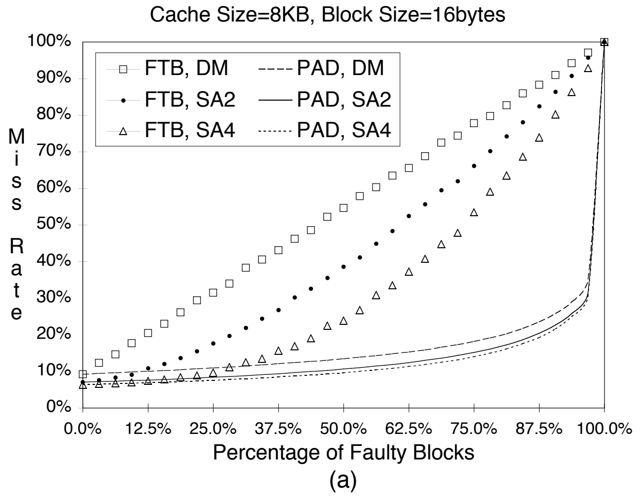
Figure 4.1(b) shows the lower left corner of Fig. 4.1(a). Notice that for a small percentage of faulty blocks, both techniques perform equally well for the set-associative caches.

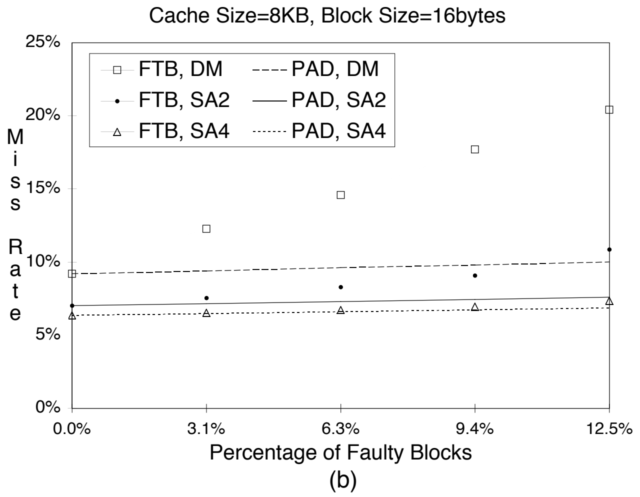
Another advantage of our technique is shown in Fig. 4.1(c). This figure shows the minimum and maximum miss rates of the ATUM traces for different fault locations, in a direct-mapped cache. The miss rate of FTB has a noticeably big range (as was shown in [7]) while the miss rate of PAD remains almost 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.

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.

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





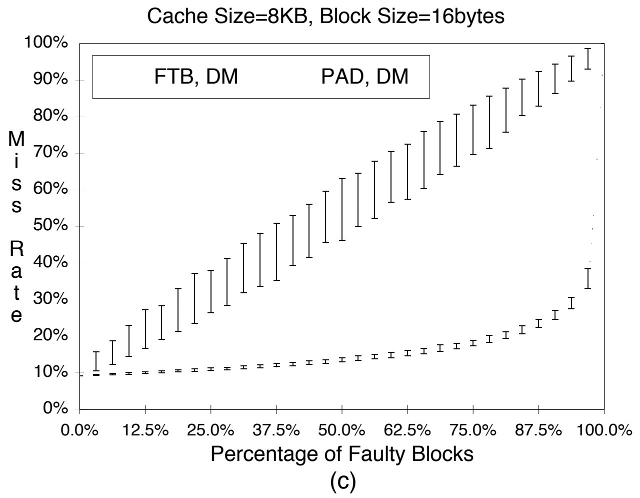


Figure 4.1 Average miss rates of the ATUM traces: (a) different associativities, (b) lower left corner of Fig. 4.1(a), (c) min. and max. miss rates for a DM cache.

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

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. The delay overhead of adding these gates can be made negligible by choosing proper sizes for the transistors in the decoding stages. 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.

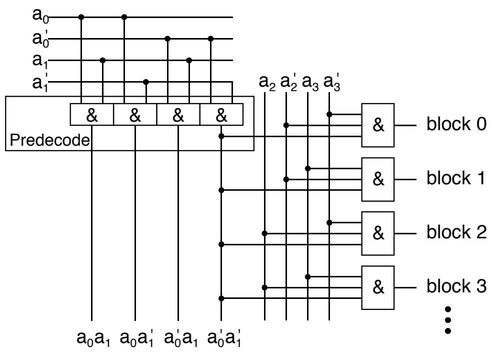


Figure 5.1 A decoder implemented by logic gates.

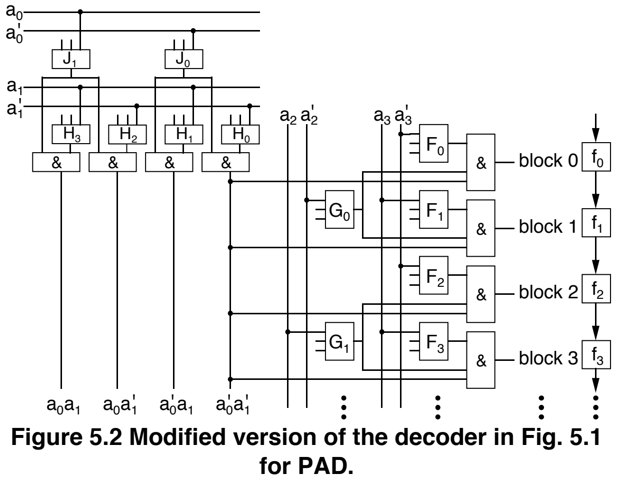


Figure 5.2 Modified version of the decoder in Fig. 5.1 for PAD.

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. 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). 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 [4]. 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.

7. Conclusion

In this paper, 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 [8] 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

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] Shirvani, P.P., and E.J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories," *CRC-TR*, Stanford Univ., in preparation.