Performance of Graceful Degradation for Cache Faults

Hyunjin Lee Sangyeun Cho Bruce R. Childers
Dept. of Computer Science, Univ. of Pittsburgh
{abraham,cho,childers}@cs.pitt.edu

Abstract

In sub-90nm technologies, more frequent hard faults pose a serious burden on processor design and yield control. In addition to manufacturing-time chip repair schemes, microarchitectural techniques to make processor components resilient to hard faults will become increasingly important. This paper considers defects in cache memory and studies their impact on program performance using a fault degradable cache model. We first describe how defects at the circuit level in cache manifest themselves at the microarchitecture level. We then examine several strategies for masking faults, by disabling faulty resources, such as lines, sets, ways, ports, or even the whole cache. We also propose an efficient cache set remapping scheme to recover lost performance due to failed sets. Using a new simulation tool, called CAFÉ, we study how the cache faults impact program performance under the various masking schemes.

1 Introduction

As device and wire size continue to decrease, the likelihood of permanent circuit faults in a microprocessor increases, especially at very small sizes, below $90nm$ [11]. In current processors, hard faults typically occur due to manufacturing defects. These defects are identified through rigorous testing prior to chip deployment. In memory elements, such as caches, a number of defects can be repaired by enabling spare capacity. Chip repair techniques are vital to achieving good yield and are widely used for processors [3].

Hardware faults also happen due to phenomena other than manufacturing defects, such as process variation [1] and aging [13]. Such defects happen at any time (*e.g.*, due to thermal hot spots) and may manifest themselves as *operational faults* at the microarchitecture level. Unfortunately, traditional techniques that rigorously test and physically modify the chip to make it functional are typically impractical after deployment.

Because simply testing and repairing chips during manufacture is insufficient [4, 5, 11], architectural techniques are needed to mask the inevitable operational faults. Redundancy is one way to achieve this capability [5]. Triple modular or duplex thread redundancy can be used [10]. However, a significant disadvantage to redundancy is its high resource and power/energy cost.

Ideally, the redundancy that is used for protecting against hard faults in traditional approaches should be directly utilized. *Graceful degradation* approaches do not replicate resources for fault tolerance, but use existing replication to overcome faults. Because caches are employed only to improve program performance, the graceful degradation approach can be used to mask faults, by disabling (or "deleting") the faulty portion of the cache. Deleting a portion of the cache however has a performance affect that is dependent on the fault and the frequency of access to an address that would normally be held in the disabled portion of the cache.

The goal of this paper is to explore how defects manifest themselves as faults in cache structures and how they affect performance. Based on the fault manifestations and performance impact, we examine how cache designs have to be changed to overcome faults with minimal performance penalty. From our study, we find that it is typically sufficient to disable individual cache lines, unless the whole set fails. When a set fails, it leaves a "hole" in the cacheable address space, which leads to a significant performance degradation in some cases. To address this problem, in conjunction with disabling individual cache lines, we propose a novel remapping scheme that can direct addresses from a failed cache set to a functional one.

Bruce Childers was supported in part by NSF awards CNS-0551492, CNS-0509115 and CNS-0305198.

2 Background

2.1 Fault classification

Projections suggest that future microprocessors based on an advanced nanometer-scale CMOS technology will be subject to three classes of reliability threats or *faults*: *hard faults*, *intermittent faults*, and *transient faults* [11]. Hard faults reflect irreversible physical damage, caused by imperfect material and process. Also, aging phenomena such as electro-migration, stress migration, gate-oxide breakdown, and negative bias temperature instability (NBTI) can pose serious reliability issues, leading to operational faults [13]. For example, Kumar *et al.* [7] found that NBTI can significantly degrade the read stability of SRAM cells by reducing their *static noise margin* (SNM) by as much as 9% in 3 years.

Intermittent faults happen because of unstable or marginal hardware, activated by changed operating conditions, *e.g.*, higher temperature or lower voltage. Transient, soft errors are caused by charge-assuming particles striking sensitive devices and reversing stored logic states.

The focus of this work is on hard and intermittent faults caused by *(hardware) defects*. Faults due to defects will become a more serious concern in the future. First, they lead to unavailable hardware resources and can degrade the performance continuously. Second, future processors will inevitably confront more hardware defects as the effectiveness of traditional test techniques is challenged by continued device scaling [11]. Lastly, the manufacturing economy may force chip vendors into selling processors with known defects. That is, to maintain a profitable chip yield, processors with defects may be considered "marketable" as long as the resulting faults are *maskable* in the field. This trend will become more prevalent in the future as chip lifetime reliability and manufacturing yields are critically threatened [4, 5].

2.2 Related work

Sohi [12] looked at the performance impact of line deletion. Using a trace-driven simulation method and a set of unified cache models with three sizes (256B/1kB/8kB), this work evaluated the degradation in cache performance due to defects. Faults were injected randomly into lines, and simulations were repeated several times with a different set of defective cache lines to get an estimate of expected performance degradation. Pour and Hill [9] extended Sohi's work by introducing a more systematic way to evaluate the impact of defects in a cache memory. These earlier works study a rather simple unified cache configuration using ATUM traces. More recently, Agarwal *et al.* [1] proposed a direct-mapped cache organization capable of deleting cache lines.

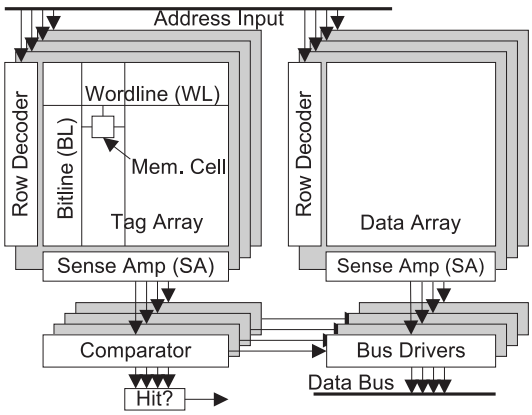

Figure 1. A 4-way set associative cache structure.

3 Fault Degradable Caches

3.1 Fault manifestations in cache memory

Figure 1 depicts a typical set-associative cache to motivate our discussion in this subsection. Major components include memory cells in tag/data arrays, wires (wordline/bitline/bus), supporting logic (decoders/hit-miss logic) and peripheral circuits (sense amps/drivers).

Though conceptually simple, a cache implementation can have many variations. For example, memory arrays can be merged together, or partitioned further into smaller sub-arrays or banks depending on speed, power, and layout requirements. Contiguous cache lines can be mapped to a single bank or distributed over multiple banks. Address decoders can be shared by a number of banks. In any case, cache operations are done by a sequence of steps, utilizing different hardware resources. For a read operation, it is ⟨address input, decoding, wordline (data array), memory cell, bitline, sense amp, data bus, data output⟩ ("data read path") and ⟨address input, decoding, wordline (tag array), memory cell, bitline, sense amp, comparator, hit-miss logic⟩ ("tag read path"). For a write operation, it is ⟨address input, address decoding, wordline (data/tag array)⟩ ("wordline select path") and ⟨data input, sense amp, bitline, memory cell (data/tag array)⟩ ("data/tag write path").

Manifestation	Causes
Faulty cache line	A memory cell in the cache line is unstable and does not retain its value; a pass transistor connected to a memory cell is faulty (*e.g.*, gate-source is short) and the attached bitline is stuck at zero/one/float.
Faulty cache set	A group of memory cells are weak; row decoder logic has a fault and a wordline is stuck at zero/one/float (when the row decoder is shared); a word line is marginal and causes a timing delay; a memory bank is marginal and affects the associated set in a CAM-tag design.
Faulty cache way	A bitline is open or marginal, barring fast signal propagation from majority of cells; bitline conditioning circuit is faulty and would not precharge bitlines properly; a sense amplifier is faulty and would not recognize valid signal levels on reads and would not enforce proper signal levels on writes; power supply to a memory bank is unstable; a bus driver in a memory bank is faulty and the resulting signal is unstable.
Faulty cache port	Address or data bus connected to a cache port has marginal wires; sense amps associated with a port are degraded and does not match the target speed.
Faulty cache	A set of faults (described above) that affect all accesses; a critical DC path from power supply to ground; IR drop causes an overall speed problem; defects in hit logic, address/data bus.

Table 1. Fault manifestations and possible causes.

All major components in cache, including memory cells, wires, logic, and peripheral circuits, are subject to defects [7, 13]. *A cache fault will occur if a defect in cache components interferes with any step in a read or write operation.* Defects in cache can manifest themselves in a number of ways, as summarized in Table 1.

3.2 Degrading strategies

A number of graceful degradation strategies can overcome the effect of various fault manifestations in cache memory. We consider general strategies here.

Line delete. When a particular cache line is faulty, it can be marked and excluded from normal cache line allocation and use. A programmable *fault map* can be provided to record the markings. As an implementation of the fault map, an "availability bit" may be attached to each cache tag and treated as the second valid bit [8].

Set delete. When a set becomes faulty, it can be marked so and deleted. In certain cases, this can be simply done by deleting all lines in the set if a line delete scheme is employed. On the other hand, since the nature of faults may not allow using the per-line or per-set fault marking schemes utilizing the tag memory, more robust fault map techniques may be needed. One such technique is to employ a second decoder logic leading to an array of fault map bits.

Way delete. One can shut down a cache way if it becomes unavailable for use due to defects. Conceptually, an N-bit fault map can tell which way(s) are unavailable in an N-way set-associative cache. Depending on the nature of defects, one may shut down a cache way by simply turning off a specific per-line availability bit in all cache sets.

Cache shutdown. When defects are major and memory accesses as a whole do not benefit from using the defective cache, it can be simply turned off.

Port delete. This strategy is somewhat different from the other strategies, in the sense that it requires changing the outside view of the cache. When a cache port is deleted, the instruction issue and steering logic must be aware of this change and should not use the pipeline leading to the faulty cache port.

Cache resizing. Considering that hard faults tend to form a cluster [14], neighboring cache resources may be affected together by a set of clustered faults. To mask them altogether, one can reconfigure a programmable decoders to exclude the faulty rows (lines or sets, depending on implementation) and use only the available resources. For example, we can tie a certain cache index bit to zero or one to delete faulty memory rows.

Row remapping. When memory rows (and cache lines in them) are lost, accesses to the faulty rows can be directed to other rows. As will be shown in Section 5, this strategy is very effective in tackling the performance loss due to a few faulty cache sets. The proposed row remapping scheme calls for a change in the decoder driver (typically a series of inverters) as shown in Figure 2(a) and (b). In addition, a set of programmable remap match registers (along with their associated logic) are also needed (Figure 2(c)). The match registers are programmed to record the (faulty) cache sets to remap. When there is a cache access hitting a "remapped" set, one of the registers will have a match, which will drive the wordline leading to the target set. By properly sizing the NOR gates, the proposed row remapping scheme will not affect the critical path in a typical pipelined cache implementation [15].

4 CAFÉ

To quantitatively assess the impact of faults in cache memory on program performance, we have developed a tool set, called *CAFÉ* (CAche Fault Evaluation tools) on top of the SimpleScalar simulation infrastructure (v4.0) [2]. CAFÉ comprises (1) a cache profiler; (2) systematic and random fault map generators; (3) an analytical performance estimator; and (4) a cycle-accurate performance simulator. The key data structure used during evaluation is *fault map*, which describes the

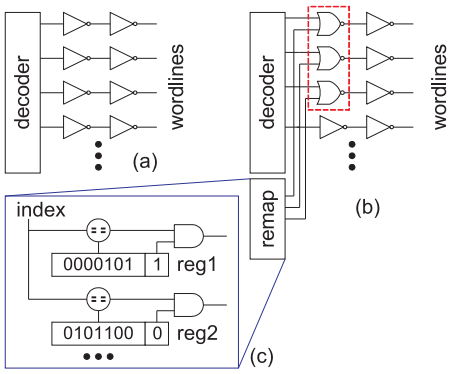

Figure 2. (a) Conventional decoder. (b) Remap-enabled decoder. (c) Remap unit.

faults (their types and locations) to be injected into cache memory under study. A fault map can be generated systematically based on profile data, randomly using the random fault map generator, or manually.

Given a program and cache configuration, the cache profiler generates a detailed cache access profile to feed the systematic fault map generator and the analytical performance estimator. The profiler collects per set hit/miss count and per LRU entry access count using a stack-based algorithm [9]. The stack-based algorithm allows us to assess the impact of removing any number of cache lines (with faults) from a set without profiling for each different configuration. The profiler also supports FIFO-based caches by profiling for each configuration with different associativity, from 1 to the original associativity. Once the cache access profile is collected, one can immediately evaluate the impact of a set of faults, of different types and numbers, in terms of the resulting miss rate and program execution time. For example, the analytical performance estimator, with help from the systematic fault map generator, can answer questions like: "what is the maximum impact of losing N cache lines due to faults?", "what is the minimum impact of losing N cache sets?", "what is the average impact of losing N cache sets?", and "what is the impact of losing a cache way?"

We employ a set of greedy algorithms and the Monte Carlo method while systematically generating fault maps to obtain accurate results. Greedy algorithms are used to create a fault map leading to a minimum or maximum performance impact, given the detailed cache profile data. The algorithm implementation is relatively straightforward for caches with an LRU replacement policy. Figure 3 exemplifies how the algorithms work, given the per block access counts. Interestingly, the choice of N cache blocks leading to a maximum miss count may not depend on the choice

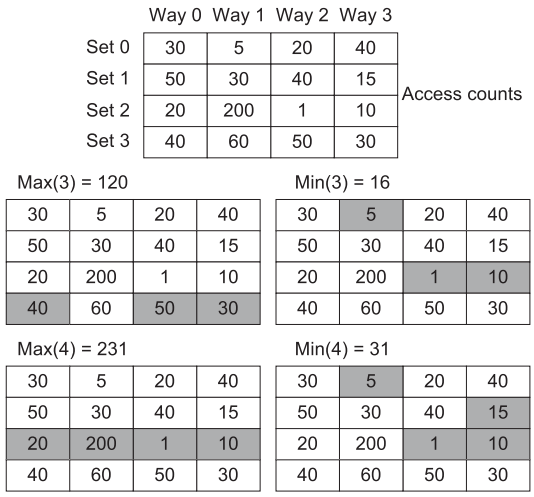

Figure 3. An example access count profile and fault maps leading to a max. or min. number of misses.

Conf.	Parameters
M1	**"Embedded Processor"**
	Single in-order pipeline 8kB 16-way I/D caches – 32B line, 1-cycle latency 50-cycle latency main memory via a 64-bit bus 2k-entry bi-mod branch predictor
M2	**"High-Performance Processor"**
	8-issue out-of-order processor with 128 ROBs 32kB 4-way I/D caches – 128B line, 3-cycle latency 2MB 8-way L2 cache, 256B line size, 18-cycle latency 240-cycle latency main memory via a 128-bit bus 4k-entry combined branch predictor

Table 2. Key machine parameters.

of $N-1$ cache blocks. Generating a fault map systematically for a FIFO-based cache is much more involved than an LRU cache, due to the well-known Belady's anomaly. We use a heuristic-based integer-programming algorithm for FIFO-based caches. The Monte Carlo method is used to quickly compute the average impact of a given number of faults.

The random fault map generator can randomly pick up N distinct cache lines or cache sets, as well as to mark a specified number of cache ways to inject faults into. During the selection process, we consider each cache line or set either *independently* or in *clusters*. Based on a simplified center-satellite model [14], we pick up cluster centers randomly. Cluster size (*i.e.*, the number of faulty lines or sets in a cluster) can be set to a constant value or based on the Poisson distribution around a mean value.

Finally, we develop an execution-driven processor simulator based on sim-outorder [2], extended with the degradable cache organization. It is used to measure detailed program performance given a fault map.

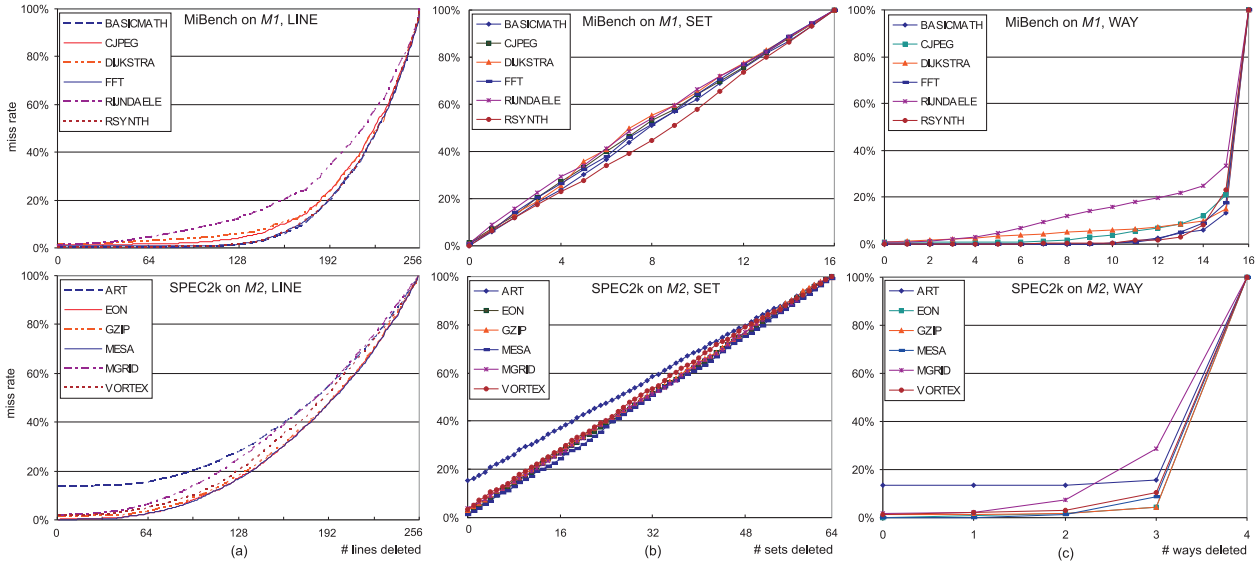

Figure 4. Impact of deleting (a) lines, (b) sets, and (c) ways on two different machine configurations.

5 Quantitative Evaluation

5.1 Machine models and workload setup

To evaluate graceful degradation, we examined two machine models: a simple embedded processor and a complex superscalar design. Table 2 summarizes the models. For workload, we used six programs (*art*, *eon*, *gzip*, *mesa*, *mgrid*, *vortex*) from the SPEC2k benchmark and six programs (*basicmath*, *cjpeg*, *dijkstra*, *fft*, *rijndaele*, *rsynth*) from MiBench [6]. We picked programs based on their different miss rates, ranging from ∼0.0% (*basicmath*) to 13.7% (*art*). Programs were compiled to target Alpha (SPEC2k) and ARM (MiBench) using Compaq Alpha C compiler (V5.9) with the `-O3` optimization flag and gcc 2.95.2 with `-O2`.

5.2 Result

Among the cache degrading strategies discussed in Section 3, we focus on line/set/way delete and row remapping in the results. We assume that faults in cache memory are detected and necessary cache reconfiguration is done *before* program execution, similar to current practices [5]. A detailed discussion on on-line cache testing is beyond the scope of this paper.

Deleting lines and sets. The impact of deleting cache lines or cache sets in terms of miss rate is presented in Figure 4(a)–(b). In general, capacity loss due to line deletions results in a smaller impact than the same capacity loss due to set deletions. This is because the remaining cache lines after deleting faulty cache lines can continuously serve cache accesses directed toward the partially deactivated cache sets. On the other hand, deleted sets cause recurring holes in the memory space that are not serviced by the cache memory at all, leading to a much larger impact.

It is likely that in a highly set-associative cache, the loss of cache lines will affect performance less, compared with a cache that is not. When half of cache capacity is lost due to deleted lines, for example, *M1* (16 ways) has miss rates up to 12%, while *M2* (4 ways) sees higher miss rates, 17% and above. The shallow region in the curves, where miss rate is less than 20%, is much wider in *M1* than *M2*. Similarly, the angular point of each curve tends to reach farther to the southeast in *M1*. If accesses in a program are distributed equally on different LRU entries in each cache set, its curve will become straighter (*e.g.*, *rijndaele*). When deleting sets, on the other hand, the expected impact grows almost linearly.

Deleting ways. It is interesting to find that the impact of deleting ways is quite limited, as shown in Figure 4(c). Even with only one way available (*i.e.*, when 15 ways are deleted in *M1* and 3 ways in *M2*), high hit rates of 67–95% were achieved.

Range of impact. Now, we turn our attention to the possible maximum and minimum impact of deleting cache lines and sets. Figure 5 has three curves showing maximum, average, and minimum impact for three selected programs.

As shown, *vortex* has a large gap between the maximum and minimum curves and it becomes more difficult to predict performance accurately given a set of

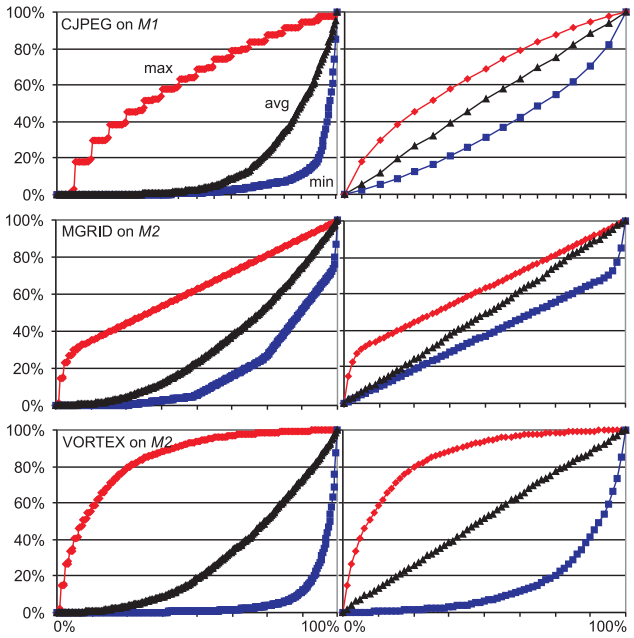

Figure 5. Max./avg./min. impact of deleting lines (left) and sets (right) on miss rate.

faulty cache lines or sets. These "big eyes" suggest that cache set usages are heavily unbalanced and clustered in these programs, and that only a few lines are actively used in those sets. On the other hand, *cjpeg* and *mgrid* have a narrower gap between the curves, especially in the set graph. Except for a few outliers, *mgrid* uses cache sets in a very uniform way, resulting in near straight-line maximum curves. Although *cjpeg* has uses cache sets relatively regularly, the minimum curve due to line deletion is much shallower initially and steeper later than that of *mgrid*. This is because cache lines within each set, as sorted in the LRU list, have much different access profiles. *mgrid* again has the most balanced usage of cache lines within each set, as shown in Figure 4(c), and the minimum impact curve for line deletion becomes in essence a 4-segment (4-way cache) piecewise linear curve.

It is further shown that there is criticality in the lines (sets) deleted. At the loss of 12.5% capacity, the maximum impact was 30%, 36%, and 59% for *cjpeg*, *mgrid* and *vortex*, respectively. At the loss of 25% capacity, the impact can be as large as 46%, 45% and 80%.

Impact on program performance. To assess the impact of faults on program performance, we measured programs' execution time as we incrementally inject faults. For this experiment, we used two fixed fault maps generated for all the benchmark programs. The first fault map (dubbed *FM-R*) is randomly generated,

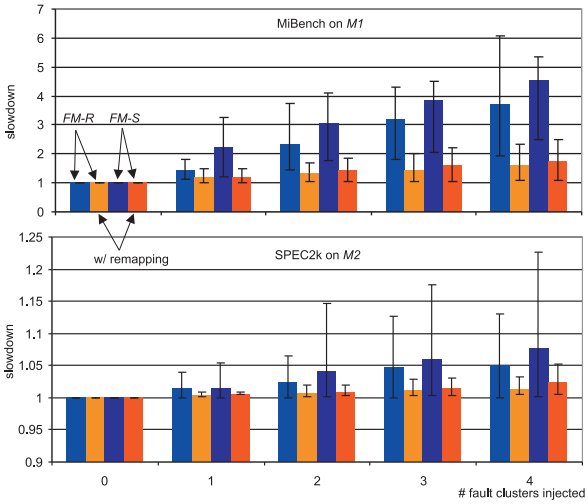

Figure 6. Impact of clustered faults on execution times. Error bars show the max./min. impact.

while the second one (*FM-S*) is systematically generated to select the cache sets that are most frequently accessed by all programs. We simulated two different caches for each machines: The first one is with conventional decoder in Figure 2(a), where as the other ("*w/ remapping*") is with the remap-enabled decoder in Figure 2(b). Figure 6 presents the result.

From the results without the remap-enabled decoder, *M1* has a large slowdown (it lacks an L2 cache) and *M2* has a noticeable slowdown. The average slowdown for *M1* is 1.5 to 3.7 (for one to four fault clusters) when faults are randomly injected. *M2*'s average slowdown is 1.01 to 1.05. The slowdown for the programs in each workload varies due to memory behavior. For two fault clusters, systematic fault injection shows that the maximum slowdown is 1.15, which is significant in an aggressive processor design.

The graphs also show how the remap-enabled decoder can overcome the performance lost due to failed sets. The lightly shaded bars show the slowdown when failed sets are remapped. Remapping can be quite effective. For *M1* with the remap-enabled decoder, the average slowdown is 1.1 to 1.5 (from one to four fault clusters), which is a significant improvement over simply disabling the failed sets. Machine *M2* also sees a benefit: the average slowdown is reduced to a maximum of 1.02 for four fault clusters and systematic fault injection (the last set of bars). From these results, we conclude that a remap-enabled decoder is beneficial, even for a small number of cluster faults.

5.3 Discussion

Based on our results, we discuss how caches in future processors will have to be designed in response to the growing threat from hard faults.

1. *Future caches must tolerate faults.* Given that defects will happen during processor operation, and these defects will lead to faults, it is imperative to develop techniques for fault-tolerant caches. Because the performance impact can be high in some programs, *caches should be designed in a way in which they offer graceful degradation* due to the faults. Further, the *whole cache hierarchy* needs to be designed with faults in mind. Our results show that the impact of faults is greatest when there is only an L1 cache. However, there can be a large impact for a cache hierarchy with multiple levels. Thus, simple disabling strategies (from Figure 6 w/o remapping), are likely insufficient for future processors that will be subject to many more hard faults.

2. *Covering the full address space is important.* When some portion of the address space cannot be cached due to a fault, the performance impact can be high. This situation may occur when a whole set (a line in a direct-mapped cache) is lost. Thus, *a cache should be designed to be "adaptable"* so that it can cache *all* addresses when faults occur. Address re-mapping (see Figure 6 w/ remapping) is one way to achieve this capability. An alternative is to provide spares, where additional resources can be selectively enabled to take the place of faulty ones. Similarly to address re-mapping, sparing ensures that the address space can be covered, but it also maintains the original cache's capacity. It has an area cost (for the spares) and a possible latency cost. Of course, both techniques can be applied together.

3. *The "criticality" of cache elements must be considered.* Different cache elements have varying criticality in terms of the program's performance. A cache line that is frequently accessed in an in-order processor is quite important to good performance. If that cache line is lost due to a fault, then performance suffers. *Criticality can be used to guide decisions* about address re-mapping and sparing.

6 Conclusion

A growing threat in the design of microprocessors is the dramatically increasing probability of a hardware defect in a deployed processor. Cache designers for a future processor will have to consider and tackle this challenge to achieve good overall product yield and reliability. This paper examined how circuit defects can cause faults in processor caches. Given the possible fault manifestations, the paper described different graceful degradation strategies that disable unreliable cache lines, sets, and ways. Our results show that losing a cache set often has the most impact because a portion of the address space cannot be cached. This paper also showed that a simple remap scheme can recover much performance loss due to faulty sets. Moreover, we show that different cache elements have varying criticality in terms of the program's performance. For a future cache design with a degree of reconfigurability, criticality can be used to guide decisions about address re-mapping and sparing. Our results provide valuable insight into the performance and design of fault-tolerant, degradable caches that will be necessary in future processors.

References

[1] A. Agarwal *et al.* "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies," *IEEE Trans. VLSI Systems*, 13(1): 27–38, Jan. 2005.

[2] T. Austin *et al.* "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, Feb. 2002.

[3] D. K. Bhavsar. "An Algorithm for Row-Column Self-Repair of RAMs and Its Implementation in the Alpha 21264," *Int'l Test Conf.*, pp. 311–318, Sept. 1999.

[4] S. Borkar *et al.* "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Tech.@Intel*, March 2005.

[5] D. C Bossen *et al.* "Power4 System Design for High Reliability," *IEEE Micro*, 22(1): 16–24, Jan. 2002.

[6] M. R. Guthaus *et al.* "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Workshop Workload Characterization*, pp. 110 – 111. Dec. 2001.

[7] S. Kumar, C. Kim, and S. Sapatnekar. "Impact of NBTI on SRAM Read Stability and Design for Reliability," *Int'l Symp. Quality Electronics Design*, March 2006.

[8] D. A. Patterson *et al.* "Architecture of a VLSI Instruction Cache for a RISC," *Int'l Symp. Computer Arch.*, pp. 108–115, June 1983.

[9] A. F. Pour and M. D. Hill. "Performance Implications of Tolerating Cache Faults," *IEEE Trans. Computers*, 42(3): 257–267, Mar. 1993.

[10] E. Rotenberg. "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Int'l Symp. Fault-Tolerant Computing Systems*, pp. 84–91, June 1999.

[11] SEMATECH. "Critical Reliability Challenges for the International Technology Roadmap for Semiconductors (ITRS)," *Technology Transfer #03024377A-TR*, March 2003.

[12] G. S. Sohi. "Cache Memory Organization to Enhance the Yield of High-Performance VLSI Processors," *IEEE Trans. Computers*, 38(4): 484–492, April 1989.

[13] J. Srinivasan *et al.* "The Impact of Technology Scaling on Lifetime Reliability," *Int'l Conf. Dependable Systems and Networks*, pp. 177–186, June 2004.

[14] B. Vinnakota and J. Andrews. "Repair of RAMs with Clustered Faults," *Int'l Conf. Computer Design*, Oct. 1992.

[15] C. Zhang, F. Vahid, and W. A. Najjar. "A Highly-Configurable Cache Architecture for Embedded Systems," *Int'l Symp. Computer Architecture*, pp. 136–146, June 2003.