DEFCAM: A Design and Evaluation Framework for Defect-Tolerant Cache Memories

HYUNJIN LEE, SANGYEUN CHO, and BRUCE R. CHILDERS, University of Pittsburgh

Advances in deep submicron technology call for a careful review of existing cache designs and design practices in terms of yield, area, and performance. This article presents a Design and Evaluation Framework for defect-tolerant Cache Memories (DEFCAM), which enables processor architects to consider yield, area, and performance together in a unified framework. Since there is a complex, changing trade-off among these metrics depending on the technology, the cache organization, and the yield enhancement scheme employed, such a design flow is invaluable to processor architects when they assess a design and explore the design space quickly at an early stage. We develop a complete framework supporting the proposed DEFCAM design flow, from injecting defects into a wafer to evaluating program performance of individual processors on the wafer. Using DEFCAM, interesting interactions between architectural, organizational, and layout/defect related parameters can be easily evaluated. Moreover, we propose practical set remapping schemes to contain hard faults in cache memory. In a set remapping scheme, accesses that would go to an unusable faulty set are directed to a sound set. Case studies are presented to demonstrate the effectiveness of the proposed design flow and developed tools. Experimental results show that a set remapping is the most efficient fault covering method among prevailing strategies.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance analysis and design aids; B.3.4 [**Memory Structures**]: Reliability, testing, and fault-tolerance; C.4 [**Performance of Systems**]: Fault tolerance

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Process Variations, Yield

ACM Reference Format:
Lee, H., Cho, S., and Childers, B. R. 2011. DEFCAM: A design and evaluation framework for defect-tolerant cache memories. ACM Trans. Architec. Code Optim. 8, 3, Article 17 (October 2011), 29 pages.
DOI = 10.1145/2019608.2019616 http://doi.acm.org/10.1145/2019608.2019616

1. INTRODUCTION

At feature sizes below 65nm, more frequent hard faults due to random defects and process variations pose a serious burden to processor design and yield management [SEMATECH 2003]. As device size scales downward, a smaller defect size makes it easier to introduce faults that cause critical failures. Process variations also become problematic due to limitations in lithography and process control. The primary effect is on device length and threshold voltage, which can adversely impact timing and leakage. Amplified process variations require that operational margins widen, making it difficult to construct functioning chips with worst case design.

This work is supported in part by the National Science Foundation, under grant CCF-0702236, grant CCF0811295, grant CCF-0811352, grant CCF-0952273, grant CCF-1059283, and grant CCF-1064976.
Author's address: email: abraham@cs.pitt.edu.

This problem is particularly pronounced in the memory hierarchy of a processor chip. As the march continues toward large on-chip memories, more of the total transistor budget is devoted to memory. For example, in Intel's Nehalem-EX processor [Rusu et al. 2009], the memory arrays for the L2 and L3 caches account for well over 50% of the total chip area. There is a greater likelihood of defects and variations in memory as these structures grow in size because memory transistors are some of the smallest and most timing sensitive. Traditional techniques for masking memory faults use redundancy, where functional spare elements (e.g., columns and/or rows) take the place of defective ones. However, this approach will have to devote a large chip area to the spares in future nanometer-scale technology [Agarwal et al. 2005], leading to an adverse interplay with yield.

In response to the problems with redundancy in nanometer technology, new yield-enhancing techniques based on *graceful degradation* are gaining much attention [Lee et al. 2007; Agarwal et al. 2005; Sohi 1989]. In graceful degradation, all memory capacity is exposed to the microarchitecture (i.e., there are no spares). Graceful degradation can disable failed components and possibly reconfigure operational ones to serve as substitutes. For caches, faulty components, such as cache lines, sets, and ways, can be disabled without changing the processor functionality, possibly with a performance penalty. Programs that are cache intensive may suffer a larger penalty than programs having few memory accesses. Even the specific memory access pattern in a program can have an impact—a failure in a cache line that is heavily accessed may result in more performance loss than a fault in a less frequently used line. This approach has been applied to recent processors [Rusu et al. 2009; Bossen et al. 2002], and will become more commonplace, even in embedded and SoC designs.

An advantage to traditional redundancy approaches is that cache microarchitecture and yield can be treated independently, especially at an early stage during architecture specification. Assuming no faulty components exist, a processor architect can focus on designing key processor components to achieve the target performance regardless of yield management schemes. In this approach, yield should be maintained at the lower level (e.g., circuit level) which determines the physical layout of the processor's elements.

However, with graceful degradation, there is a strong interdependence between cache microarchitecture and yield. In this case, the cache microarchitecture and the yield management scheme must be considered *simultaneously* to make appropriate design decisions. For example, one graceful degradation mechanism might have good defect coverage but a high performance penalty on a benchmark workload. Another scheme might have a negligible performance impact but cover only certain faults in a few cache resources. A processor architect needs to evaluate the choices to select the most appropriate one, given the expected workload and defect and process variation characteristics of the target technology.

In this article, we propose a design methodology, called "Design and Evaluation Framework for defect-tolerant Cache Memories" (DEFCAM), that has fault models based on defects and metrics that can effectively evaluate the trade-off between yield and performance with different fault masking mechanisms. Using DEFCAM, performance impacts of the processor cache can be estimated quickly with faults caused from randomly or systematically injected low-level defects.

The methodology relates the occurrence of defects at the circuit level to faults at the cache organization level. The occurrence of process variations or defects in the circuit can cause many different fault manifestations, including the failure of cache lines, sets, and ways. Based on the failures, microarchitecture schemes that disable faulty components may be modeled and workloads simulated to evaluate the performance impact. Our methodology introduces a new family of metrics, called *yield-area-performance* (YAP), for comparing alternative yield management schemes. YAP can be used to evaluate different graceful degradation approaches, as well as traditional redundancy schemes. It can be used to quantify yield at a particular performance goal. Importantly, our methodology lets a processor architect quickly and early in the design process evaluate the trade-offs.

This article makes several important contributions. First, it presents a new high-level evaluation methodology for various yield management strategies. Second, it describes a model that can relate circuit-level defects and process variation in caches to faults at the organizational level. Third, it gives a novel metric (YAP) for evaluating different graceful degradation and redundancy approaches. Finally, the article presents a case study of using our methodology and the YAP metric. We find that a set remapping graceful degradation scheme can achieve a higher yield than redundancy approaches at a slight performance cost.

The rest of this article is organized as follows. Section 2 presents background and explains in detail fault classification and manifestations in cache memory. Section 3 explains our proposed design flow that generates physical level defects and maps defects to faults. Section 4 illustrates delete schemes to cover faults at the architectural level and analyze the minimum and maximum performance impact. Our proposed set remapping schemes are demonstrated in Section 5. Using the tools and strategies, we describe case studies in Section 6. Related works are summarized in Section 7, and concluding remarks are presented in Section 8.

2. BACKGROUND

2.1. Fault Classification

A *fault* is an event and cause of an *error* which can eventually lead to a *system failure*. For instance, a fluctuation in the power supply that is larger than the design margin and an alpha particle hitting the chip circuit are faults. If a circuit state is affected by a fault (e.g., a bit flip in an SRAM cell), an error is said to occur. When the changed circuit state is eventually propagated to the program state, a system failure can happen. Not all errors lead to a system failure, however, since faults and errors may occur to a circuit portion that is not currently in use.

Projections suggest that future microprocessors based on advanced nanometer-scale CMOS technology will be subject to three classes of faults: *hard faults*, *intermittent faults*, and *transient faults* [SEMATECH 2003]. Hard faults reflect irreversible physical damage (defect), caused by imperfect material and process. Intermittent faults happen due to unstable or marginal hardware, activated by changed operating conditions, e.g., higher temperature or lower voltage. Both high temperature and low voltage cause circuit speeds to slow. Transient, soft errors are caused by charge-assuming particles striking sensitive devices and reversing stored logic states. Cosmic rays and alpha particles generated from chip packaging material are known to cause soft errors.

As circuit technology enters the sub-65nm regime, much attention has been paid to the impact of process variation and lifetime reliability issues [Borkar 2004]. Process variation refers to the fluctuations in process parameters observed after fabrication. These variations result from a wide range of factors during the fabrication which determine the ranges of variations and can lead to designs that deviate significantly from their specification. Furthermore, lifetime reliability issues become more pronounced with technology scaling which increases power densities in the processor. Aging phenomena such as electro-migration, stress migration, gate-oxide breakdown, and negative bias temperature instability (NBTI) can also give rise to hard faults while the chip is operational [Srinivasan et al. 2004]. Kumar et al. [2006] find that NBTI can

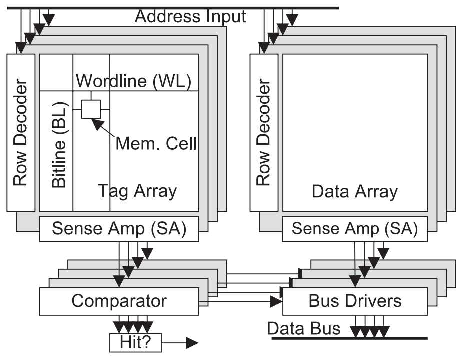


Fig. 1. A 4-way set associative cache structure.

degrade the read stability of SRAM cells by reducing their static noise margin (SNM) by as much as 9% in 3 years.

The focus of this work is on hard and intermittent faults caused by *hardware defects* and *process variations*. Faults due to defects and process variations will become a more serious concern in the future.

2.2. Fault Manifestations in Cache Memory

When a processor architect considers faults in cache memory, low-level defects and their immediate effect, such as a single bit failure, may not be seen at the architectural level. It is beneficial for an architect to identify faults at the architectural level instead of low-level especially when evaluating the impact of such faults at an early design stage. This section briefly discusses how low-level defects and process variations can manifest themselves at the cache organizational level [Lee et al. 2007]. Figure 1 depicts a typical set-associative cache for the following discussion. Major components in a cache include memory cells in the tag/data arrays, wires (wordline/bitline/bus), supporting logic (decoders/hit-miss logic) and peripheral circuits (sense amps/drivers).

All major components in a cache, including memory cells, wires, logic, and peripheral circuits, are subject to defects [Kumar et al. 2006; Srinivasan et al. 2004]. A cache fault will occur if a defect in a cache component interferes with any step in a read or write operation. The critical cache access path is: ⟨address input, decoding, wordline (data array), memory cell, bitline, sense amp, data bus, data output⟩ and ⟨address input, decoding, wordline (tag array), memory cell, bitline, sense amp, comparator, hit/miss logic⟩.

Defects in cache can manifest themselves in a number of ways at the cache organizational level. First, *individual cache lines* can become faulty and unusable. For example, if a memory cell in a tag or data array has a defect, the corresponding cache line will be unusable. This cache line fault model is commonly used in previous studies [Agarwal et al. 2005; Pour and Hill 1993; Sohi 1989]. We note that SRAM read stability is aggravated with scaling and cache lines whose cells suffer from reduced SNM will be more frequent in the future. Second, an *entire cache set* may fail. For example, process misalignment can lead to a large number of unstable memory cells, which may cause the loss of cache sets. When there is a defect in the row decoding logic, specific wordlines stick to ground/V_{DD} or they may float. As a result, an entire cache set becomes unusable. In certain cases, defective memory banks can lead to losing a group of cache sets. Third, an *entire cache way* can become faulty due to marginal bitlines or degraded sense amplifiers. Lastly, the *entire cache* can be lost as a result of a few critical defects in shared resources. For example, defects in the hit logic, the address bus, or the data bus will interfere with every cache access. Consequently, it becomes impossible to use the cache memory.

3. DEFCAM

The design of critical processor components like cache memory at the architectural level is complicated by the requirement of meeting a number of important, yet potentially conflicting design goals. Although performance is a key goal in high-performance processor designs, the cost of a design, typically measured in chip area, is also an important aspect to ensure a competitive product. Yield is another important design consideration that affects the cost structure of the product. To achieve the best design, a processor architect has to make judicious trade-offs between these interacting goals: performance, area, and yield. In this section, we present a new Design and Evaluation Framework for defect-tolerant Cache Memories (DEFCAM) that allows a processor architect to study and evaluate these trade-offs for the on-chip cache.

Evaluating a cache design in terms of different goals is complicated by two factors, which DEFCAM has to consider. First, the evaluation needs to be done early in the design flow. However, not all information needed to perform the evaluation may be available at an early design stage. Second, the interaction among yield management and the cache organization is complicated because a selected cache mechanism simultaneously impacts performance, area, and yield. For example, a mechanism might help yield, but harm performance. Indeed, architecturally visible yield-enhancement schemes, such as set remapping as proposed in Section 5, are challenging in this regard because they trade run-time performance for improved yield. DEFCAM considers these two factors by deriving high-level fault manifestations at the cache organizational level from low-level specification. As a result, fault-aware evaluation and behavioral simulation can be done with the derived fault information. DEFCAM can be used to iteratively evaluate design decisions and how they interact with performance, area and yield.

3.1. Integrated Design Flow

DEFCAM provides a processor architect with an integrated framework to evaluate a cache design's yield, area, and performance.[1] Figure 2 shows the DEFCAM framework. The design flow can be easily extended to include other traditional metrics, such as power and energy consumption. However, power is not the focus of this paper because good power estimation tools already exist, which can be integrated into the framework [Tarjan et al. 2006; Ye et al. 2000].

There are three obstacles in current methodologies that have to be addressed by DEFCAM. First, during early design exploration, a processor architect needs to understand how defects and process variations affect the cache to select a good yield management scheme. There are complex trade-offs among yield, area, and performance, especially when yield control mechanisms with different area and defect coverage are considered [Lee et al. 2007; Thomas and Anthony 1999]. Second, defects and process variations manifest themselves at the physical level and need to be related to the cache microarchitecture under consideration. However, the physical design is unknown during early design exploration. Finally, because a particular yield management scheme can affect several design layers from the application level to the physical level, efforts from independent design groups have to be integrated.

DEFCAM addresses these problems in the following way. From a "cache microarchitecture specification," the design flow automatically derives a "physical cache model." The physical cache model approximates the anticipated implementation of the given

[1]Yield usually refers to the proportion of dies on a wafer that perform properly up to a design specification. In this paper, we extend the use of "yield" to a cache within a processor chip.

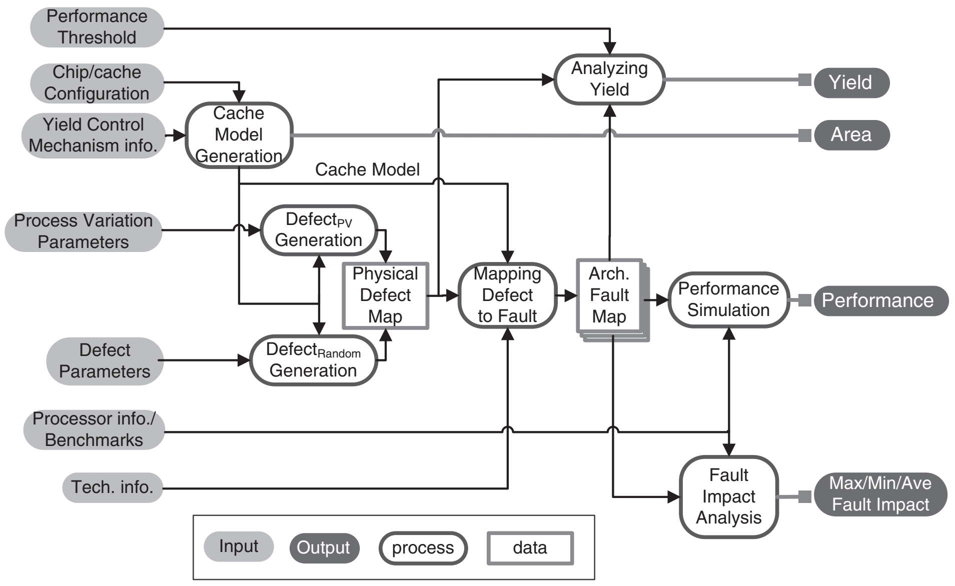


Fig. 2. DEFCAM design flow overview.

cache specification. For instance, the number of wordline segments (in Table I) can be used to calculate the number of row decoders in the physical cache model. Defects and process variations are automatically placed in the physical model according to a "defect model." The defect model is based on the characteristics of the target technology. The defects and process variations at the circuit level are then mapped to the organizational level to determine how they affect the cache (e.g., which cache lines failed). Because some yield management schemes do not guarantee program performance, our methodology simulates—at the architecture behavioral level—a cache with a set of given faults. The simulation is done automatically only in cases when the yield management scheme can impact program performance (e.g., when the number of faults is bigger than the number of redundant rows). To ensure statistically valid coverage of many possible fault manifestations, the methodology uses extensive sampling at the virtual wafer level to derive and evaluate different defect and process variation scenarios (e.g., 100 wafers per each scenario).

To use the design flow in Figure 2, several inputs have to be specified. Table I lists these inputs. The processor architect would typically specify the cache organization, program workload and processor architecture. Information about the target technology can be provided by circuit and process engineers. The output of the design flow includes yield, area, and program performance. In our current implementation, obtaining the output from the input is fully automated.

3.2. Generating Physical Defects

Procedures for generating and projecting defects into physical layer were developed. We use a 300mm silicon wafer model and the die area model from the specified processor model. A hard defect is a spot defect on the physical layout of the wafer. Based on the physical information input to DEFCAM, we randomly generate physical defects with defect density and clustering effect factors. One can also set defect sizes from the ITRS projection data [ITRS 2005]. After generating defects, we inject them into a wafer according to the location of defects in the wafer. We use a Gaussian random

Table I. Input Parameters

Cache configuration	Description
A	Set associativity
B	Block size (in bytes)
C	Cache size (in bytes)
N_{dwl}	Number of wordline segments
N_{dbl}	Number of bitline segments
N_{spd}	Number of sets mapped to a single wordline

Yield control mechanism	Description
Redundancy	Number of redundant rows
ECC	Number of correctable bits
Disabling	Line/set/way disabling
Set remapping	Number of target set candidates

Defect	Description
Density	Number of defects in unit area (1mm^2)
Clustering	Average number of clustered defects
Size	Average size of defects

Process variation	Description
Inter-die variation	$\Delta V_{th-inter}$ (in V)
Intra-die variation	$\Delta V_{th-intra}$ (in V)

Technology information	Description
Wafer size	Diameter (mm)
Threshold voltage	V_{th} (in V)
cx, cy	SRAM cell dimension (in nm)

Processor	Description
Pipeline	In-order/out-of-order, issue width, number of ROBs
Cache	L1/L2 cache architectural configurations
Branch predictor	Number of entry, bi-mod/combined/g-share
Memory	Access latency, bus width

distribution model for the defect locations and sizes in this work, but any distribution model can be specified. Clustering factors can represent the nature of the real processor's defects [Friedman et al. 1997]. Two defect generation blocks (i.e., physical defect generation and process variation generation) and the physical defect map in Figure 2 illustrate these processes in the DEFCAM design flow. After generating physical defects, we translate low-level defects into architectural level faults using the methodology described next.

3.3. Mapping Defects to Faults

To accurately translate physical defects into architecturally visible faults, a detailed cache specification is necessary. The most important cache design parameters are architectural parameters, organizational parameters (e.g., banking), and physical layout information such as array geometry and SRAM cell layout. Our design flow provides an interface to describe a cache's geometry inside a chip.

Architectural parameters are a 3-tuple: A (associativity), B (block size), and C (cache size). We consider three organizational parameters: N_{dwl}, N_{dbl}, and N_{spd}, which determine the internal sub-banking by specifying how wordlines and bitlines are partitioned [Tarjan et al. 2006; Wada et al. 1992]. They are listed as cache configuration parameters in Table I. They also define cache line to subbank mapping. Cache geometry is modeled as a hierarchy of non-overlapping rectangles, from the wafer and chip down to cache and cache components. Cache components include memory arrays and control logic. To ease illustration, we use derived parameters in Table II. These parameters are used for calculating faulty way/set numbers (i.e., architectural fault information) from defect parameters (i.e., circuit-level defects or process variations). For the sake of

Table II. Derived Parameters for Mapping Defects to Faults

Parameter	Description
px, py	Coord. of a given physical defect ϕ from physical defect map (in nm)
W_b	Block width; $8 \cdot cx \cdot B$ (in nm)
W_{sb}	Sub-bank width; $A \cdot W_b \cdot N_{spd}/N_{dwl}$ (in nm)
W_{ssb}	Set width in a sub-bank ($N_{spd} \geq 1$); W_{sb}/N_{spd}
px_{sb}	Rel. X-coord. of ϕ in a sub-bank; $px \ mod \ W_{sb}$
N_{set}	Number of sets; $C/(A \cdot B)$
N_{sbl}	Number of sets per bitline; N_{set}/N_{dbl}
N_{pr}	Index of memory row having ϕ; $\lfloor py/cy \rfloor$
N_{xsb}	X-index of sub-bank having ϕ; $\lfloor px/W_{sb} \rfloor$

presentation, we consider only the data array in the following discussion. Our techniques apply equally well to the tag array.

A defect or process variation in the physical model is mapped to the cache microarchitecture based on the affected component. In this mapping, our methodology considers three component types: bitline, wordline, and memory cell. Bitline and wordline faults include sense amplifier faults and row decoder faults, respectively. Once lower level defects are converted to architectural faults, different defects are indistinguishable at the organization level.

3.3.1. Bitline and Sense Amplifier. A defect in a bitline or a sense amplifier causes faulty cache lines. Depending on the cache organization, all the lines in a way can be lost as in Figure 3(a) or only a portion of them as shown in Figure 3(b) and (c). The left part of the figure shows some defects placed in the cache at the circuit level, and the right part architectural faults according to the defects. We consider two cases depending on the value of N_{spd}. When $N_{spd} \geq 1$, total $(N_{set}/(N_{spd} \cdot N_{dbl}))$ lines are affected. Within a selected cache way (*way num* in below equations), one cache line in every N_{spd} consecutive cache lines is faulty. In other words, given a defect $\phi = (px, py)$ which shows two dimensional location information, all the sets with their number equal to $(initial\ set\ num + n \cdot N_{spd} \ mod \ (N_{set}/N_{dbl})) \geq 0$, are faulty. *way num* and *initial set num* are computed as:

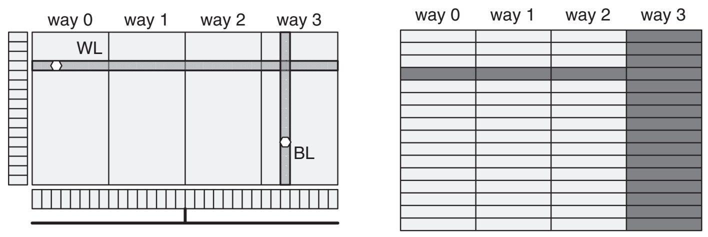
$$\begin{aligned} way\ num &= N_{xsb} \cdot W_{ssb}/W_{sb} + \lfloor (px_{sb} \ mod \ W_{ssb})/W_b \rfloor \\ initial\ set\ num &= N_{pr} \cdot N_{spd} + \lfloor px_{sb}/W_{ssb} \rfloor . \end{aligned}$$

When $N_{spd} < 1$, total (N_{set}/N_{dbl}) consecutive cache lines become faulty. They are identified with:

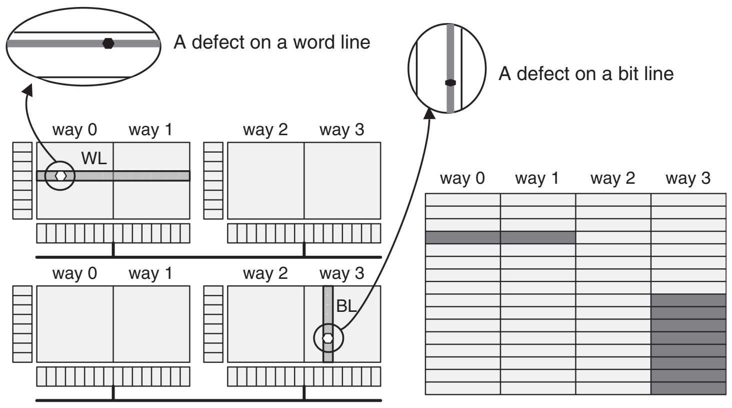
$$\begin{aligned} way\ num &= \lfloor px/(W_b \cdot N_{spd}) \rfloor \\ set\ num,\ start &= \lfloor (N_{pr} \cdot N_{spd})/N_{sbl} \rfloor \cdot N_{sbl} . \end{aligned}$$

3.3.2. Wordline and Row Decoder. Like a defective bitline, a defective wordline or a defective row decoder (i.e., an unusable decoder line) can cause unavailable cache lines in a number of different cache ways. All lines in a set can be lost as in Figure 3(a), a few cache lines in a set may become unavailable as in Figure 3(b), or cache lines in multiple cache sets can become faulty as in Figure 3(c). When $N_{spd} \geq 1$, faulty cache lines are clustered in a rectangle (N_{spd} by A/N_{dwl}) whose origin can be computed as follows:

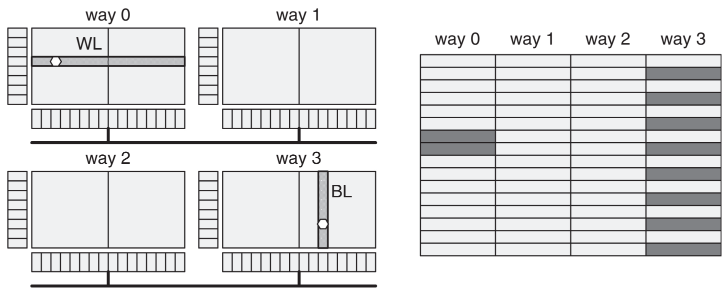
$$\begin{aligned} set\ num,\ start &= N_{pr} \cdot N_{spd} \\ line\ num,\ start &= N_{xsb} \cdot A/N_{dwl} . \end{aligned}$$



(a) A=4, N_{dwl}=1, N_{dbl}=1, N_{spd}=1



(b) A=4, N_{dwl}=2, N_{dbl}=2, N_{spd}=0.25



(c) A=4, N_{dwl}=4, N_{dbl}=1, N_{spd}=2

Fig. 3. Defective wordline or bitline (left) results in various architectural faults (right).

When $N_{spd} < 1$, (A/N_{dwl}) consecutive lines become faulty within a single set. The initial line is identified with the following:

$$\begin{aligned} \textit{set num} &= \lfloor N_{pr} \cdot N_{spd} \rfloor \\ \textit{line num, start} &= \left\lfloor \frac{\lfloor px/(W_b \cdot N_{spd}) \rfloor}{A/N_{dwl}} \right\rfloor \cdot \frac{A}{N_{dwl}}. \end{aligned}$$

3.3.3. Memory Cell. A faulty cache line due to a defective memory cell is determined by the following equations. Again, we consider two cases based on the value of N_{spd}. When $N_{spd} \geq 1$, calculating the set and line number is identical to finding the initial set and way number of the bitline defect case.

$$\begin{aligned} set\ num &= N_{pr} \cdot N_{spd} + \lfloor px_{sb}/W_{ssb} \rfloor \\ line\ num &= N_{xsb} \cdot W_{ssb}/W_b + \lfloor (px_{sb}\ mod\ W_{ssb})/W_b \rfloor \end{aligned}$$

When $N_{spd} < 1$, this is also very similar to the bitline defect case.

$$\begin{aligned} set\ num &= \lfloor N_{pr} \cdot N_{spd} \rfloor \\ line\ num &= \lfloor px/(W_b \cdot N_{spd}) \rfloor \end{aligned}$$

Besides the effect of random defects, we consider the impact of process variations on memory cell reliability. Failures caused by process variations are heavily dependent on operating voltage, frequency, and temperature. Since these conditions are varying factors, the worst case operation condition (e.g., low voltage and high temperature) is routinely considered at the manufacturing time. Thus, we assume the worst operating condition as a fixed design point. While there are systematic variations and random variations, we only consider random variations in this study. Process variations are caused by imperfect control over the channel length, width, oxide thickness, and placement of dopants. Among them, random dopants are more important than the others because it causes a threshold voltage mismatch even between nearby transistors [Tang et al. 1997]. Different threshold voltages vary the transistor's response time and lead to SRAM cell's access time failures and read and/or write failure. These three failures (access, read, and write time failures) compose the probability of single cell failure. This problem is magnified when adjacent transistors in one SRAM cell have different threshold voltages [Agarwal et al. 2005]. The probability of failure increases as the variation of threshold voltages grows. For example, effective probability of failure becomes 1×10^{-3} in 45nm technology where ΔV_{th} is 30mV [Agarwal et al. 2005]. We treat process variations similar to random defects based on the failure probability.

3.4. Area Calculation

In our current implementation, DEFCAM uses CACTI [Tarjan et al. 2006] to calculate the baseline cache area for a given technology. Further, there are three elements that affect the area: row redundancy, ECC, and an availability bit for graceful degradation. For redundant rows, data area grows proportionally to the number of redundant rows. We assume conventional SECDED code (i.e., 8-bit ECC code for 64-bit data) to evaluate the area overhead from ECC. We consider one bit overhead for graceful degradation, which comes from each block's availability bit [Sohi 1989].

3.5. Yield Calculation

DEFCAM computes cache yields by simulating fault occurrences from the information about cache configuration, low level defects, process variations, and yield enhancing techniques employed. Low-level defects are randomly injected to a user-specified number of dies, which eventually become bit line, word line, cell, or peripheral errors based on the cache configuration. Process variations are calculated for individual cells using the methods described in Section 3.3.3. These two types of errors are projected to higher level architectural fault information. With the architectural fault information, the impact of different yield enhancing techniques is evaluated. If a given yield enhancing technique is unable to cover all faults in a die (e.g., more faults than the number of redundant rows), yield is sacrificed.

3.6. Performance Simulation

DEFCAM simulates workloads to measure the performance for different architectural fault maps, as shown in Figure 2. In our current implementation, the performance simulator is based on SimpleScalar [Austin et al. 2002] with an additional module to simulate the effects of cache faults with fault masking strategies—delete schemes and remapping schemes—which are described in Section 4. This performance simulator enables the user to analyze how much performance is impacted from faults. The performance result also becomes an input to the Yield Analyzer (see Figure 2). When a disabling or remapping scheme is used, a cache achieving performance equivalent to a user-specified threshold or better is regarded as a sound cache. In this article, we use either a 95% or a 99% performance threshold for the caches using a graceful degradation scheme. For the 95% performance threshold, if any benchmark's performance on a processor show more than 5% degradation from disabling faulty blocks in the cache, it is discarded.

3.7. YAP: Yield, Area, and Performance

The yield of a cache memory depends on its area and organization as well as defect distribution and process variation effects. A cache memory with a defect which hinders its normal operation is considered a failed component, unless a yield enhancement technique masks the effect of the defect. The area of a cache depends on its baseline design (specified by design parameters) and the yield enhancing scheme employed. The performance of a cache memory is primarily determined by its architectural parameters. Note that these metrics interplay. For example, a larger cache design may lead to lower yield. If a disabling scheme is used, the resulting yield may be high, but the average performance obtainable from the degraded cache may be low. Therefore, these metrics should be made available together to a designer at an early design phase so that good design decisions can be made.

Our design flow reports these metrics together in a 3-tuple (Y (yield), A (area), P (performance)). To conveniently compare multiple design points with a single number, the three terms can be combined in various ways. For example, when Y, A, and P are normalized to a baseline cache design with no yield enhancement, $Y^l \cdot A^{-m} \cdot P^n$ gives a single number which ranges between 0 and 1. The negative exponent for A comes from the fact that a smaller area cache presents a better design with the same performance and yield. The choice of l, m, and n depends on the emphasis of analysis. If a processor architect concerns more on area, m should be bigger than other two numbers. $Y^{0.7} \cdot A^{-0.9} \cdot P^{0.4}$ is an example, which stresses area rather than performance.

4. FAULT MASKING STRATEGIES: DELETE SCHEMES

DEFCAM can model many different fault schemes for cache memory, as described in Section 3.3. In this section, we use DEFCAM to explore some of these schemes, as well as new ones. We start with "delete schemes" that disable portions of the cache. These schemes are already present in some modern processors [Rusu et al. 2009; Bossen et al. 2002] and are based on the observation that programs run correctly as long as the processor retrieves (from a defective cache) correct data regardless of cache access latency. However, any performance degradation due to masking the defects is a critical issue for processor usability and yield analysis/management. There are general strategies that can be used to trade performance for guaranteed correct operation, including line, set, and way delete.

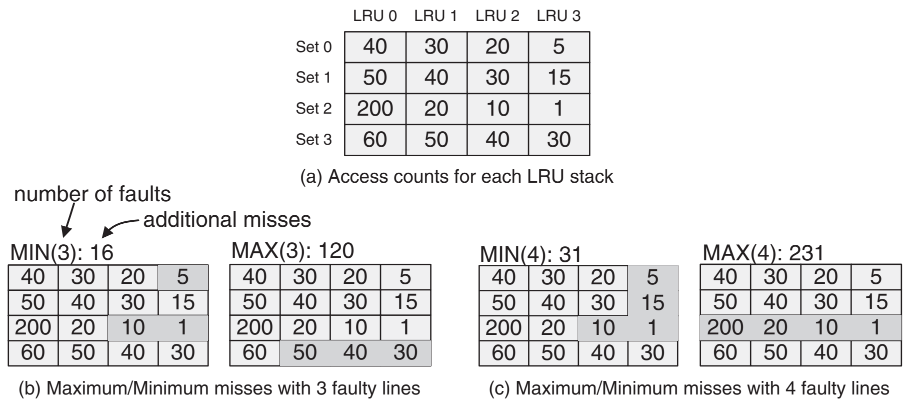

LRU 0
LRU 1
LRU 2
LRU 3
Set 0
Set 1
Set 2
Set 3
(a) Access counts for each LRU stack
number of faults
additional misses
MIN(3): 16
MAX(3): 120
MIN(4): 31
MAX(4): 231
(b) Maximum/Minimum misses with 3 faulty lines
(c) Maximum/Minimum misses with 4 faulty lines

Fig. 4. An example access count profile and fault maps leading to most or fewest misses.

4.1. 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 second valid bit [Patterson et al. 1983]. Once the system is turned on, a memory BIST engine performs testing, and sets the availability bits. Any cache line with the availability bit turned off is faulty and not used. A similar strategy has been employed in IBM's Power4 and Power5 processors, which can delete up to two cache lines in each L3 cache slice [Bossen et al. 2002]. For this scheme to work, cache line allocation (e.g., LRU or FIFO) has to be aware of the fault locations in a target set. Several circuit techniques have been studied for such LRU logic [Lamet and Frenzel 1993; Ooi et al. 1992].

4.2. Set Delete

When a set becomes faulty, it can be marked as deleted. In certain cases, set deletion can be 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 per-line or per-set fault marking schemes utilizing the tag memory (e.g., a specific entry in the tag memory is not accessible), more robust fault map techniques may be needed. One such technique is to employ a second decoder leading to an array of fault map bits.

4.3. Way Delete

A cache way can be shut down if it becomes unavailable due to defects. Conceptually, an N-bit fault map can tell which ways are unavailable in an N-way set-associative cache. Depending on the nature of defects, a cache way may be shut down by simply turning off a specific per-line availability bit in all cache sets. As in the case of set delete, a more robust fault map scheme than a per-line available bit scheme may be needed. For instance, one may introduce N faulty way bits and on each cache access, treat those bits as the third valid bit (just like the per-line availability bit), common to all sets. There are also microarchitectural techniques to shut down cache ways to save power consumption [Albonesi 1999], which can also be used to delete defective cache ways.

Table III. Key Machine Parameters

Benchmarks	Parameters
MiBench	"ARM based 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
SPEC2000	"High-Performance Superscalar 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

4.4. Evaluation Methodology: Greedy Algorithm

We used DEFCAM to assess the impact of line and set delete schemes. We do not include the impact of disabling a way because it does not have significant impact except when the whole cache is turned off [Lee et al. 2007]. For more informative assessment, DEFCAM needs to evaluate the best- and worst-case scenarios for line and set deletion. For example, the deletion of a frequently used cache line may lead to different performance than the deletion of a rarely used line. Thus, we develop an algorithm that finds the worst/best miss counts (miss rate) when the most/least frequently used lines or sets are deleted. Figure 4 illustrates how the algorithm works for a four-way set-associative cache with four sets. The figures show the number of accesses per cache line for an example workload. Figure 4(a) shows the access counts for each set and LRU stack without any faults. LRU 0 means the most recently used line in the LRU stack. For example, the number for LRU 0/Set 1 is the total hit counts from LRU 0 stack in set 1 (50 from the figure).

In Figure 4(b), MIN(n) shows the minimum additional misses that result when there are n faulty lines. The minimum is computed based on the best case (i.e., the least heavily used lines are faulty). For example, "MIN(3):16" means there are only 16 additional misses when three lines are faulty, given the access counts from Figure 4(a). The highlighted lines are the faulty ones. Similarly, MAX(n) shows the maximum additional misses with n faults.

To find MIN(n), the algorithm simply accumulates the small n numbers from the given access counts. For example, to find MIN(4) , the next smallest line access count (15) is added to MIN(3). On the other hand, determining MAX(n) is more complex. For MAX(n), the algorithm sums the access counts for each set and picks the set with the highest summation. For each set, the summation starts from the LRU block to the MRU block. For example, MAX(3) is the summation of LRU 3, 2 and 1. As shown in Figure 4(b), set 3 has the maximum number of summation from LRU 3 to LRU 1. However, MAX(4), as illustrated in Figure 4(c), comes from set 2 because LRU 0 block of set 2 has far greater number than that of other sets.

Currently, DEFCAM generates cache access profile data using SimpleScalar with an additional cache profiler module. The impact of the delete schemes for each benchmark is produced with the greedy algorithm in Section 4.4. Table III summarizes the experimental configurations. We selected these configurations to represent two design points—one for embedded systems and the other for high-performance systems. The embedded processor is modeled after an ARM-based design and the high-performance processor after an aggressive out-of-order superscalar design. We used different benchmark suites for each processor to represent the kinds of applications that would run on these designs. The embedded design uses MiBench [Guthaus et al. 2001] and the

Table IV. Benchmark Simulation Configuration

Benchmarks	Inputs	Fastforward	Warmup	Simulation period
MiBench	Small input sets	No	No	Whole execution
SPEC2000	All inputs, Averaged	5B instructions	500M instructions	1B instructions

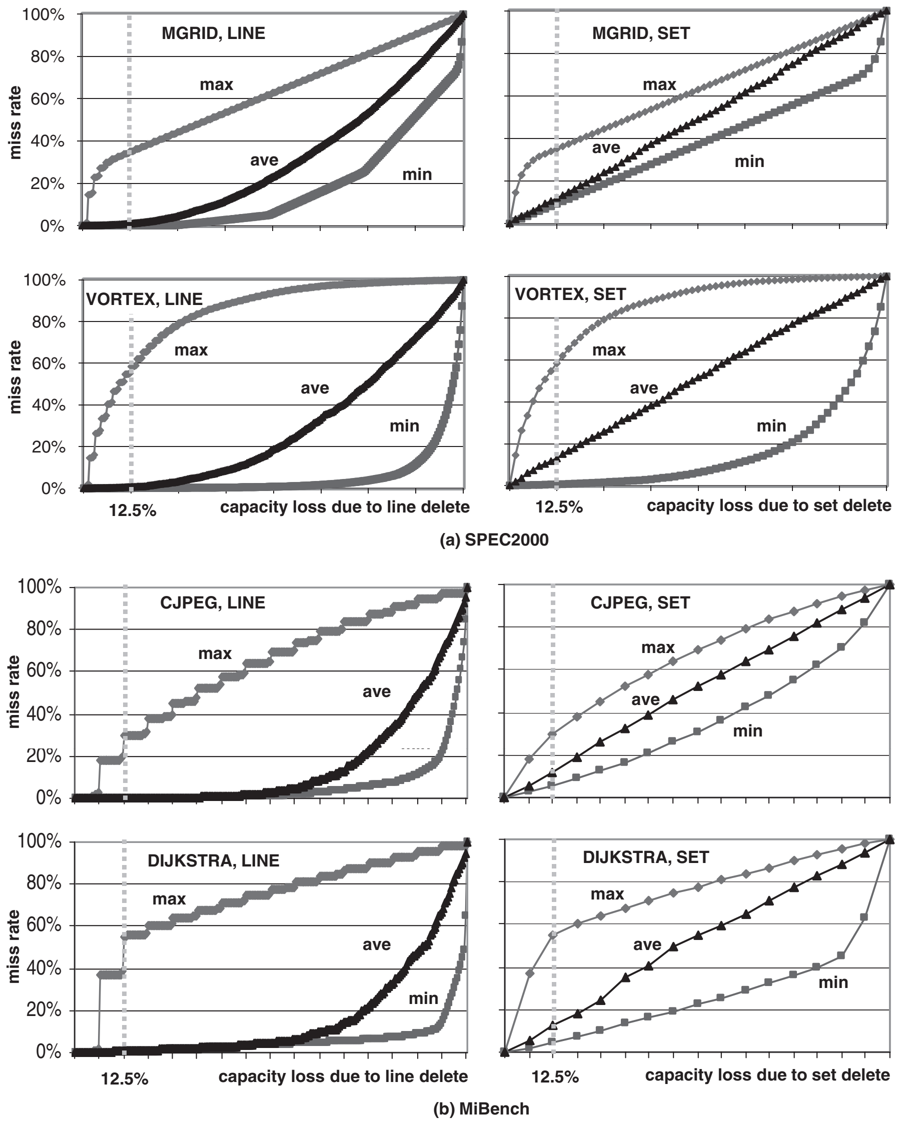


Fig. 5. Max./avg./min. impact of deleting lines (left) and sets (right) on miss rate. for selected programs.

high-performance design uses SPEC2000. Simulation benchmark configuration is summarized in Table IV.

Figure 5 illustrates the relationship between miss rate (y-axis) and capacity loss (x-axis) due to two delete schemes: line delete and set delete. Left four figures are for line delete and, similarly, right four are for set delete. It shows only four benchmarks, for brevity; two are from MiBench [Guthaus et al. 2001] and two are from SPEC2000.

Table V. Miss Rate with 12.5% Capacity Loss for Selected Programs

	MGRID	VORTEX	CJPEG	DIJKSTRA
MAX:line	0.3506	0.5842	0.2949	0.5535
MAX:set	0.3506	0.5842	0.2949	0.5535
AVE:line	0.0090	0.0064	0.0006	0.0065
AVE:set	0.1216	0.1296	0.1140	0.1267
MIN:line	0.0000	0.0006	0.0004	0.0044
MIN:set	0.0927	0.0053	0.0528	0.0471

These benchmarks have two extreme cache access trends. One case has uniformly distributed accesses to all lines/sets and the other case has a bias toward accessing only certain lines/sets. The balanced cases are *mgrid* and *cjpeg*. The biased cases are *vortex* and *dijkstra*.

Each graph has three curves showing the maximum, average, and minimum performance impact. As shown, *vortex* in Figure 5(a) and *dijkstra* in Figure 5(b) have a large gap between the maximum and minimum curves. These "big eyes" suggest that cache set usage is heavily unbalanced and clustered in these programs and that only a few lines are actively used in the sets. Therefore, it becomes more difficult to predict performance accurately given the number of faulty cache lines or sets for these benchmarks. On the other hand, *mgrid* in Figure 5(a) and *cjpeg* in Figure 5(b) have a narrower gap between the curves, especially in the set graph. *cjpeg* again has the most balanced usage of cache lines within each set, and the maximum impact curve for line deletion becomes in essence a 16-segment (16-way cache) piecewise linear curve.

Now, we turn our attention to the criticality of the lines (sets) deleted. Table V summarizes the maximum, average, and minimum miss rate with 12.5% capacity loss as an example for the four programs in Figure 5. Interestingly, the maximum miss rate from the line losses and the set losses are the same for all four programs regardless of the program's characteristics. We can interpret that those faulty lines compose faulty sets which have the same amount of capacity loss. Consequently, even with 12.5% disabled lines, the miss rate can be more than 50% for programs with "big eye" from Figure 5. "Small eye" programs also have significant miss rates (i.e., more than 29%) when critical lines become unusable.

For average and minimum miss rates, faulty sets generate a moderate miss rate, while faulty lines have a small miss rate. On the average case, faulty sets' miss rate is almost corresponding to the amount of the capacity loss. Minimum miss rate row in Table V illustrates that set faults produce a substantial miss rate even if they make the least number of misses with the given capacity loss. However, the minimum impact of line faults is very limited. For example, *mgrid* shows 9.27% miss rates for minumum set faults while miss rates becomes negligible for minumum line faults. In fact, *mgrid* has significant miss rates from only one faulty set (see Figure 5), which leads to sizable performance penalty. From above observations, we conclude that set fault is the most important to mitigate performance penalty, whereas line fault is much less significant. Line faults only become more meaningful when they compose set faults.

4.4.1. Experimental Results. Figure 6 shows the misses from one set fault using a delete scheme for all programs in MiBench and SPEC2000. The graph shows the additional misses caused by deletion per 1,000 instructions. Many programs have a large number of additional misses only from one faulty set. For example, *mcf* in SPEC2000 generates 97 additional misses per 1k instructions with only one faulty set. Similarly, *tiff2rgba* and *rijndael* in MiBench add more than 70 misses per 1k instructions with one faulty set. Although delete schemes are used in many processors, they cannot guarantee the performance when an entire set is deleted.

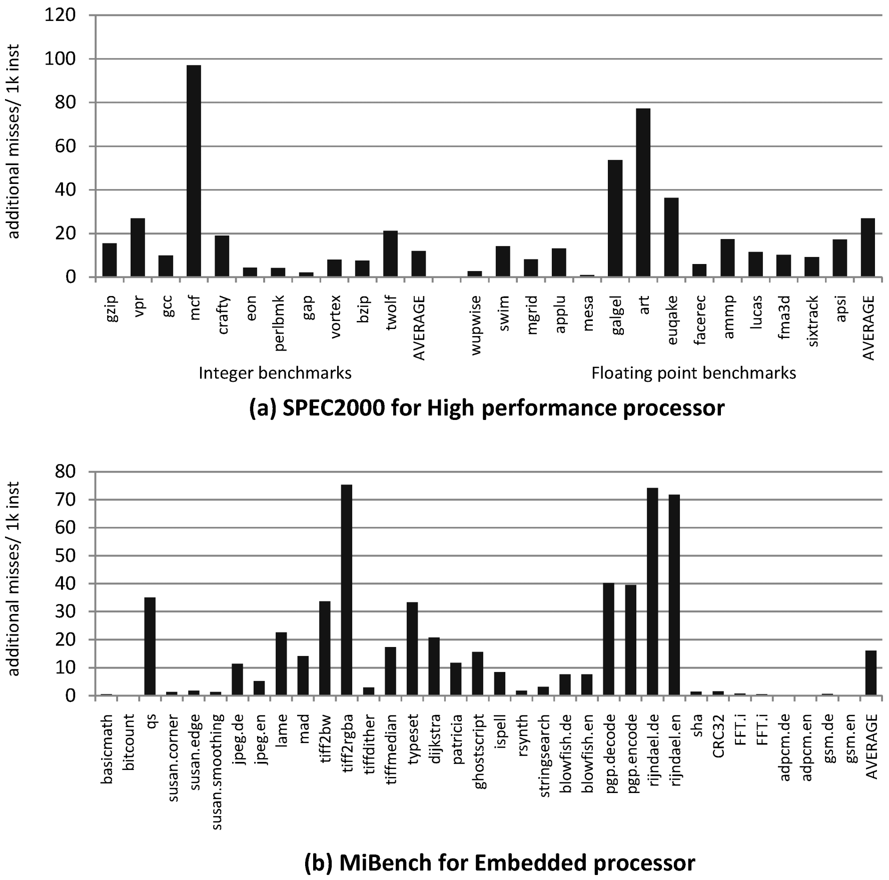


Fig. 6. Additional misses per 1,000 instructions with one faulty set defect.

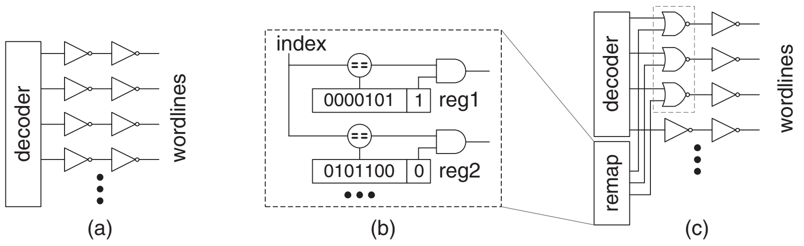


Fig. 7. (a) Conventional decoder. (b) Remap unit. (c) Remap-enabled decoder.

5. FAULT MASKING STRATEGIES: SET REMAPPING SCHEMES

The investigation with line and set delete reveals two important findings. First, the number of additional misses from faulty deleted lines is limited unless the faulty lines make up most of a set (i.e., the set is faulty). For the MiBench programs, on average, 50% of the faulty lines contribute only 3% additional misses. Second, however, when an entire set is faulty (or mostly lost due to line faults), a portion of the address space is noncacheable, which leads to a significant performance penalty.

Based on these observations, we propose a new mechanism, called *set remapping*, that permits the caching of addresses that map to faulty sets. In essence, accesses to faulty sets can be directed to other sound sets. This strategy is very effective in minimizing the performance loss due to a few faulty cache sets (see Figures 8 and 9). Figure 7 shows how a conventional row decoder can be changed to accommodate this strategy. The proposed set remapping scheme calls for a change in the decoder driver (typically a series of inverters) as shown in Figure 7(a) and (c). In addition, a set of programmable "remap match" registers are needed to record the faulty cache sets to remap (Figure 7(b)). When there is a cache access to a "remapped" set, one of the remap registers has a match, and consequently, drives the wordline to the sound (replacement) target set.

The performance impact of set remapping is dependent on the target set chosen for a faulty set. Well-chosen target sets can reduce the cache miss count dramatically. How can the best target set be picked for a faulty cache set? We expect that a heavily used set may not be an appropriate target set. For example, the set which receives the most accesses is not a good candidate target set. However, if most accesses on that target set use only a few lines among all lines in the set, even a heavily used set can be an appropriate target set. Furthermore, it is possible that the target set's usage is limited to the specific program phase when the faulty set is rarely accessed. In these cases, using access count numbers for each set to find a target set does not guarantee its optimality. Therefore, we investigate several heuristic approaches to finding the best target set. We consider three set remapping schemes: static, profile-based, and dynamic remapping.

5.1. Static Remapping

This scheme selects a sound target set for a faulty set at design time (i.e., it is hard-wired at the manufacturing time). The target sets that can be used for a faulty set are predefined at design time. An access to a faulty set is remapped to a specific target set. Thus, a target set handles accesses which would have been to two different sets (i.e., the remapped faulty set and the target set). To select a target set for a faulty one, we randomly pick a replacement from the candidate targets in a round-robin fashion, starting from the first target set. This selection avoids the need for information about program behavior. If there are more target sets than faulty sets, one target set index shares at most two set indices.

To implement this scheme, we need two hardware changes. First, an additional "fault bit" must be added in the tag memory to distinguish the original set index and the remapped set index. If the design only supports the case where the number of faulty sets is smaller than the number of target sets, only one fault bit is required per set. When the processor accesses a faulty set that is remapped to a target set, the fault bit is "1." Otherwise, an access for a non-remapped set has a "0" in the fault bit. If there are more possible faulty sets than target sets, more faulty bits are needed, which is not assumed in this study. The second hardware change is to include a remap-enabled decoder (shown in Figure 7). The access index number is compared to the remap register value which has the faulty set index number. If it matches, the appropriate target set is accessed instead of the faulty set. The mapping between faulty and sound target sets is done at manufacturing time and the remap decoder is permanently programmed at this time.

The addition of the remap capability is not expected to have a performance penalty in terms of access latency. The necessary additional logic has two components: comparators in the remap unit and NOR gates on wordlines as shown in Figure 7(b) and (c). Each comparator has a data storage and a XOR logic. The data storage has a fixed value that is set with laser fusing during manufacture. XOR logic has four transistors among which only one transistor causes switching delay. Therefore, the comparison latency of the remap unit in Figure 7(b) consists of NMOS transistor delay and AND gate delay. A conventional address decoders' critical path has multiple NAND and NOR gates to reduce the latency of the fan-in delay of large fan-in gates. Previously, Zhang showed that the delay of a CAM-based comparator circuit can be tuned to match that of a regular SRAM decoder circuit [Zhang 2006]. Given that our proposed circuit design is simpler (static from laser fusing vs. dynamic) than the design of Zhang et al., address matching in the comparators is faster than normal address decoding and the comparison latency can be effectively hidden. NOR gates are used to select an actual target set on a match and they replace inverters on the potential target sets. Again, with proper circuit optimization (e.g., cell sizing), the use of NOR gates rather than inverters leads to no performance penalty.

5.2. Profile-Based Remapping

Although static remapping covers the whole address space, it may not select the best target set because it does not use a profile about the expected workload to pick target sets. Some applications may have lots of memory accesses in a single set while other applications have evenly distributed access patterns. Utilizing profile data for each application gives a good clue to decide the most appropriate target for each faulty set. The profile includes access count and miss count. Two phases compose this scheme: the profile collecting phase and the actual execution phase. First, a processor is simulated assuming all sets are good to collect access patterns before the actual execution of the application. Counters for each cache line are used to get access patterns. Miss counts or access counts can be used for the whole application execution. This phase collects profile data for all possible faulty sets. Next, the best target set can be decided for the specific faulty set. The best target sets are the ones which produce the least additional misses counts from remapping. Once target sets are established for the application, this information can be used at run time directly. Therefore, if an application is executed N times, profile-based remapping requires $N+1$ times executions (i.e., one for the profile collecting phase and N for the actual execution phase).

We suggest three strategies to choose the best target sets. First, the best target set is chosen with the minimum hit count. When a faulty set is remapped to a target set, maximum additional miss count is limited by the original hit count in target set. Therefore, choosing the set with the smallest hit count will minimize additional misses. Second, the minimal access count set is picked for the target set. Assuming the miss count is the proportion to the access count, the set with the least access count can be a good candidate target. Lastly, the last half of the LRU stack access count could be used as the candidate. When one physical set shares two sets (the target set and the faulty set), higher LRU stack accesses for the two sets may not contribute additional misses. For example, in four way set associative cache, two higher LRU stack accesses (i.e., four LRU stack accesses in total) rather than two lower LRU stack accesses for each of the two sets will have more chance to be safe from being evicted. Consequently, lower LRU stack accesses will have more chance to sum up more additional misses. Based on the assumption that LRU stack access counts are balanced between the target set and the faulty set, the lower half of LRU stack access is counted.

5.3. Dynamic Remapping

Although profile-based remapping uses actual access information to determine target sets, once the targets are chosen, it can not adapt the targets based on the actual workload. It also requires the workload be executed once for profiling. Therefore, a question can be raised: What if applications have significantly different access patterns for different execution phases? Benefits will be limited even if we use access patterns to determine appropriate target sets.

We suggest to gather online profile data during program execution. Applications have many different phases during execution and each phase may have significantly different memory access patterns. Therefore, up-to-date information from a program's current phase can lead to more accurate set selection. Utilizing current data requires dynamically programmable address decoder in the cache. *Time window* is defined as the period of execution in which data is accumulated. Target sets are reassigned from the current profile data in the beginning of the new *time window*. After determining target sets, all information is reset to "0" to be ready for accumulating the next phase's information. Whenever a new time window begins, there are additional cache misses for two indexes, one for the old target set and the other for the new one. Therefore, *dynamic remapping* may have worse performance than static or profile-based remapping. In this case, instead of dynamic remapping, profile-based or static remapping is preferred. If the application has a similar access pattern for its whole execution, the target sets determined by dynamic remapping will be similar to static or profile-based remapping.

5.4. Experimental Results

Using DEFCAM, we examined the three remapping schemes for the same processor models and benchmarks as the delete scheme study. Figure 8 compares the schemes for all MiBench benchmarks on an ARM-based embedded processor, and Figure 9 does the same for all SPEC2000 benchmarks on the superscalar processor. Figure 8(a), (c), 9(a) and (c) show the misses for static remapping normalized to the delete scheme. "Fault free" is the number of misses without any fault. Figure 8(b), 8(d), 9(b), and 9(d) introduce the additional miss reductions for profile-based and dynamic remapping per 1,000 instructions. "Full" can choose any set as a target set whereas "4" has only 4 candidate target sets. Oracle chooses the best target set for each time window and shows the minimum misses for dynamic remapping. Because dynamic remapping must flush data for a dirty line that is remapped, a smaller window size is not always good. After sample simulations, we determined 1M cycle window size is good. We simulated many possible combinations between faulty sets and target sets and averaged them to get these numbers. As we see in Figure 8(a), 8(c), 9(a), and 9(c), static remapping can reduce most of the additional misses arising from the set delete scheme. Although static remapping for *stringsearch* incurs significant additional misses relative to the set delete scheme in Figure 8(c), *stringsearch* has a very small misses for the delete scheme in Figure 6(b). Therefore, the role of dynamic or profile-based remapping for *stringsearch* is very limited. In a similar fashion, static remapping reduces most of the additional misses of the delete scheme for SPEC2000 benchmarks as shown in Figure 9(a) and (c). Although *wupwise* and *sixtrack* in Figure 9(c) have a noticeable gap which cannot be eliminated by static remapping, Figure 9(d) shows that dynamic or profile-based remapping scheme reduces small additional misses (i.e., less than 0.11 per 1k instructions). Therefore, the benefit of using dynamic or profile-based remapping for these applications is not significant. From these experimental results, we conclude that static remapping is the best choice. It reduces the misses over the set delete scheme but does not incur the overhead of profile-based or dynamic remapping.

6. YAP EVALUATION: A CASE STUDY

To investigate the utility of DEFCAM, we did a case study to select a yield-effective cache design (with YAP) from a set of candidate designs. This study illustrates how

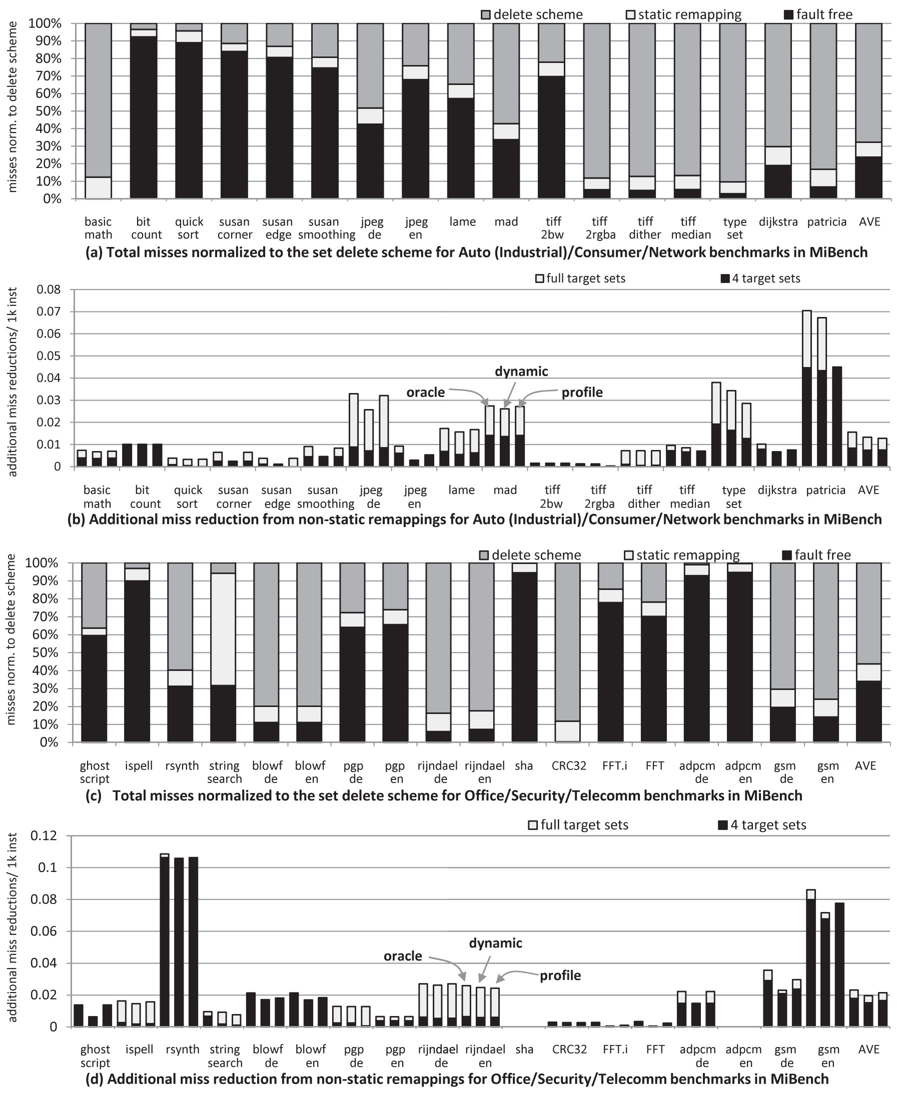


Fig. 8. Total misses normalized to the set delete scheme and additional miss reductions from three "non-static" remapping for MiBench; (a) and (b) are for Auto (Industrial)/Consumer/Network benchmarks, (c) and (d) are Office/Security/Telecomm benchmarks; oracle in (b) and (d) shows the theoretical limit of dynamic remapping (i.e., it knows the best target sets whenever new target sets are assigned). Dynamic remapping predicts the best target and assigns it to the next execution window (e.g., 100M instruction interval) from the information of the current execution window. Profile collects "profile" information in the "phase-collecting phase" and assigns best target sets for whole execution time (i.e., it becomes static remapping after assigning target sets).

a processor architect might use DEFCAM to evaluate several process/defect tolerant cache designs to select one. The first part in this section explains how we specify the design space for the case study, and the second part describes the evaluation of candidate designs.

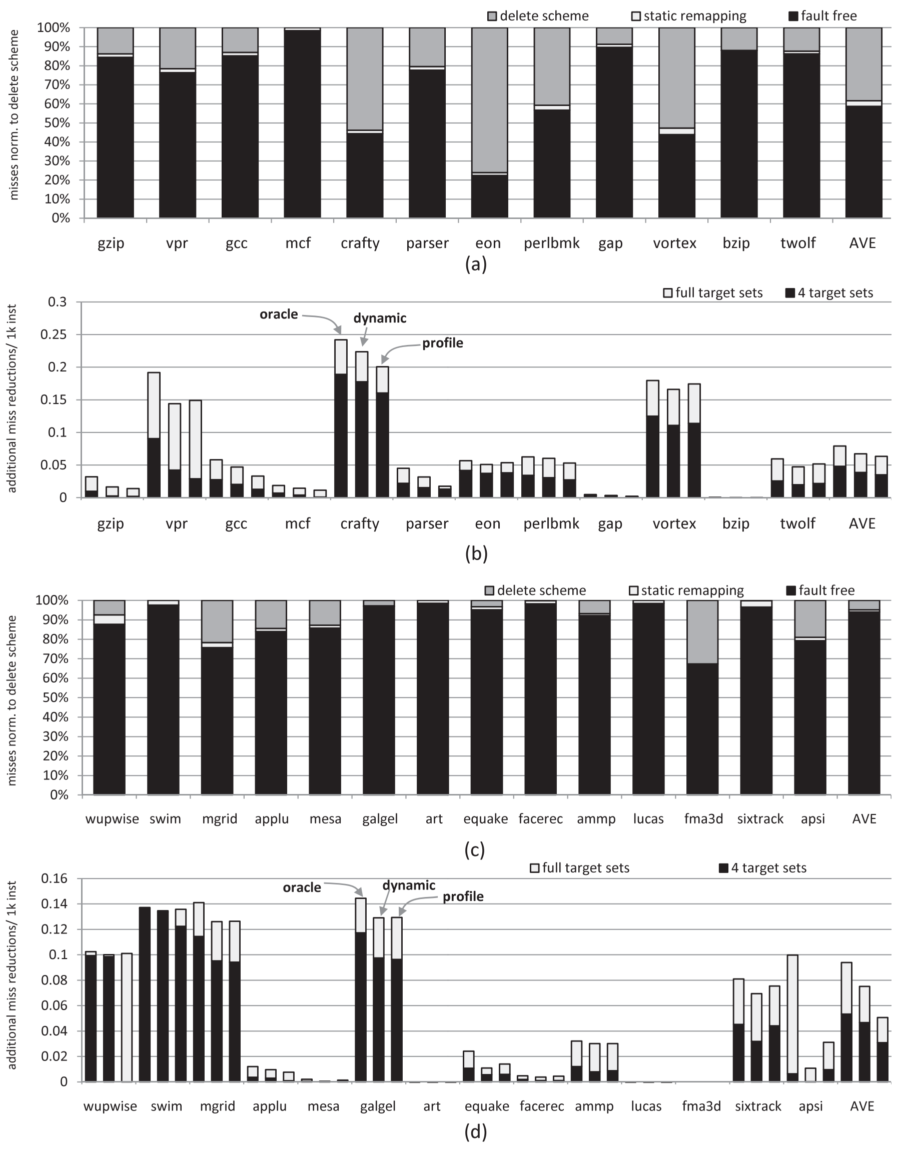


Fig. 9. (a) and (b) are for SPEC2000 INT; (c) and (d) are for SPEC2000 FP. the meaning of oracle, dynamic and profile follows the notation in Figure 8.

6.1. Specification

The case study is a system-on-a-chip (SoC) with an ARM processor in Table III. To simplify the study, we assume that defects occur only in the caches. The internal cache organization is derived with CACTI 4.1 [Tarjan et al. 2006].

The case study considers four yield management schemes for the cache: no redundancy (baseline), 12.5% and 25% row redundancy, line delete, and static set remapping (from Section 5.4). Row redundancy uses spare memory rows to cover defective ones; spares can replace any defective row. The number of spares is the percentage 12.5% or

25% of total cache lines. Line delete uses graceful degradation to permanently disable defective lines. A spare line is not used in place of a defective one, and as a result, program addresses that map to defective lines are not cached. Because error correction codes (ECC) can mitigate the effect of process variations [Agarwal et al. 2005], we consider the designs with and without ECC. For set remapping, as suggested in Section 5, we use the static scheme. The pool of target sets has four target set candidates. Therefore, at most four faulty sets can be remapped. Any faulty sets beyond this limit are deleted.

Our defect model uses a uniform distribution to select defect locations and a Gaussian distribution for defect size. Process variation has a single parameter: $\sigma_{V_{th}}$. The model is configured with a tuple (*defect density*, $\sigma_{V_{th}}$), which includes physical defect and process variation informations in one tuple. The study uses the configurations (1, 30mV), (10, 30mV), (100, 30mV), (100, 20mV), and (100, 40mV).[2] Defect density 1 and 10 conform ITRS projection data [ITRS 2005]. We aggressively generate the heavy defect density to 100. $\sigma_{V_{th}}$ 30 comes from 45nm technology's process variation [Agarwal et al. 2005]. $\sigma_{V_{th}}$ 20 and 40 are used to evaluate the impact of different parameter variations. Similarly, we evaluated the yield management schemes for three cache organizations with same cache capacity. We used three configurations with the 3-tuple, $(N_{dwl}, N_{dbl}, N_{spd}) = (1, 4, 0.5), (4, 4, 0.5), (1, 4, 0.125)$. The study uses 1,000 total die samples.

To measure program performance, the SimpleScalar tool set [Austin et al. 2002] is used in DEFCAM to simulate an ARM-like in-order pipelined processor. The workload is MiBench [Guthaus et al. 2001] with the large data sets.

Using DEFCAM, we evaluated the yield management schemes. The results are in Tables VI. In the tables, yield is the ratio of operational caches to total caches. Area is the cache area relative to the baseline. Performance is an average of the benchmarks. In this case study, we use four YAP metrics, labeled M1–M4, denoting $Y \cdot A^{-1} \cdot P$, $Y \cdot A^{-1} \cdot P^2$, $Y^2 \cdot A^{-1} \cdot P$, and $Y \cdot A^{-2} \cdot P$, respectively. As we describe in Section 3.7, power terms of YAP can be defined by the user depending on design goals. There are twelve designs with four cases for line delete and two cases for set remapping. 95% line delete and set remapping discards all caches that have more than a 5% performance degradation. Similarly, 99% line delete and set remapping discards caches above a 1% degradation.

6.2. Evaluation with Different Defects and Process Variation Density

For different defects and process variation density, most interestingly, set remapping has a very good yield for a small area cost. For example, in the (10, 30mV) configuration from Table VI(a), 95% set remapping has 100% yield, while 25% redundancy has 94% yield. For one and ten defects per cache from Table VI(a), set remapping achieves 100% yield (the numbers in the table are rounded up from yields as high as 99.9% for 95% set remapping). Yield is higher for line delete because it selects caches that meet the degradation threshold without attempting to cover the faults. Although 12.5% and 25% redundancy with ECC also attains 100% yield, its larger area cost makes it worse than set remapping in terms of YAP. This trend continues for all defect and process variation densities, and organizations.

YAP captures the trade-off between better yield and increased area/decreased performance. The tables show that no redundancy has a reduced YAP as defect density and process variation increase. In some cases, the yield gain is offset by the area cost. Although ECC and redundancy generally improve YAP, it sometimes reduces the YAP metrics with the line delete scheme. This shows the cases where area overhead is

[2]We scaled the defect densities for intuitive presentation and comparison. The defect density 1 corresponds to 0.5/mm^2.

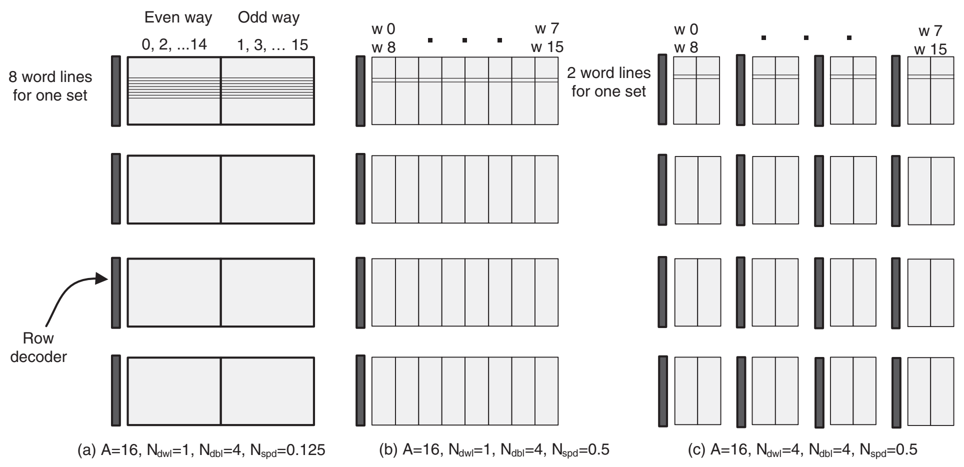

Even way
Odd way
0, 2, ...14
1, 3, ... 15
8 word lines for one set
Row decoder
w 0
w 8
w 7
w 15
2 word lines for one set
(a) A=16, N_{dwl}=1, N_{dbl}=4, N_{spd}=0.125
(b) A=16, N_{dwl}=1, N_{dbl}=4, N_{spd}=0.5
(c) A=16, N_{dwl}=4, N_{dbl}=4, N_{spd}=0.5

Fig. 10. Different organizational parameters.

greater than the benefit achieved as shown in (1, 30mV) and (10, 30mV) of Table VI(a). Another example is (100, 20mV) in Table VI(b). 25% redundancy with ECC has a better yield than 12.5% redundancy with ECC, but its YAP metrics are lower. Set remapping has the overall highest YAP because yield is improved for a modest performance and area cost. For example, in Table VI(a), 99% line delete in the (100, 30mV) case has a 66 YAP(M3). In this configuration, set remapping does much better than redundancy and delete, even with ECC.

6.3. Evaluation with Different Organizational Parameters

Figure 10 compares three different cache organizations. For the organization in Figure 10(a), one word-line error affects two lines. However, Figure 10(b) has eight unavailable lines with one word-line error. Interestingly, one word-line error in Figure 10(b) affects four times as many lines as in Figure 10(a). One word-line error in Figure 10(c) is four times less than in Figure 10(b). Their bit-line error impacts are the same. Table VI(c) illustrates yield management scheme evaluations for three organizations with the same cache size. Different organizational parameters have different area overheads because of the number of row decoders. For example, the area numbers in the (1, 4, 0.5) configuration from Table VI(c) are different from (4, 4, 0.5) or (1, 4, 0.125). As we can see in the Table VI(c), different organizational parameters have less difference than the fault masking schemes. For each design, the yield differs between 1% to 4% among the configurations. Area difference between each design is not significant. This is because the area occupied by decoders as very small as compared with the area of cell array.

6.4. Discussion

DEFCAM simplifies the process to evaluate and select different cache design for a processor architect. With user-defined YAP metrics, computer architects can evaluate many designs for their own purpose. From our evaluations, we make the following observations. First, adding ECC lines to cover hard fault works very well. For different designs including redundant rows and delete schemes, ECC increases yield up to 45%. However, using ECC for hard fault correction makes it unusable to cover soft error. In that case, ECC should be extended to cover two bit error correction with extra area cost. Next, set remapping achieves the best yield and YAP in most cases, even without

Table VI. Yield, Area, Performance, and four YAP metrics (M1–M4) in % for different: (a) defect densities, (b) process variations, and (c) organizations. The YAP metrics M1–M4 are Y·A^{-1}·P, Y·A^{-1}·P^{2}, Y^{2}·A^{-1}·P, and Y·A^{-2}·P, respectively. Numbers in bold face represent the best YAP values

(a) (defect, $\sigma_{V_{th}}$)	(1, 30mV)							(10, 30mV)							(100, 30mV)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	83	100	100	83	83	69	83	80	100	100	80	80	64	80	51	100	100	51	51	26	51
No redun. ECC	100	104	100	96	96	96	92	98	104	100	94	94	92	90	68	104	100	66	66	45	63
12.5% redundancy	91	108	100	84	84	77	78	90	108	100	84	84	75	77	69	108	100	64	64	44	59
25% redundancy	94	116	100	81	81	76	70	94	116	100	83	83	77	73	82	116	100	72	72	59	64
12.5% redun. ECC	100	113	100	88	88	88	78	100	113	100	86	86	86	74	95	113	100	82	82	78	71
25% redun. ECC	100	121	100	83	83	83	68	100	121	100	82	82	82	68	95	121	100	79	79	75	65
95% delete	100	102	100	**98**	**98**	**98**	**96**	99	102	100	**97**	**97**	96	**95**	86	102	98	83	81	71	81
99% delete	99	102	100	97	97	96	95	98	102	100	96	96	94	94	82	102	100	80	80	66	79
95% set remap	100	103	100	97	97	97	94	100	103	100	**97**	**97**	**97**	94	98	103	100	**95**	**95**	**93**	**92**
99% set remap	100	103	100	97	97	97	94	100	103	100	**97**	**97**	**97**	94	93	103	100	90	90	84	88

(b) (defect, $\sigma_{V_{th}}$)	(100, 20mV)							(100, 30mV)							(100, 40mV)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	62	100	100	62	62	39	62	51	100	100	51	51	26	51	28	100	100	28	28	8	28
No redun. ECC	71	104	100	68	68	48	66	68	104	100	66	66	45	63	68	104	100	65	65	44	63
12.5% redundancy	78	108	100	72	72	56	66	69	108	100	64	64	44	59	47	108	100	43	43	20	40
25% redundancy	91	116	100	78	78	71	68	82	116	100	72	72	59	64	53	116	100	47	47	25	42
12.5% redun. ECC	95	113	100	84	84	80	74	95	113	100	82	82	78	71	92	113	100	79	79	72	68
25% redun. ECC	97	121	100	80	80	77	66	95	121	100	79	79	75	65	95	121	100	78	78	74	65
95% delete	90	102	97	88	88	79	87	84	102	98	82	82	69	81	83	102	97	80	78	66	78
99% delete	88	102	100	86	86	76	85	82	102	100	80	80	66	79	81	102	100	79	79	64	78
95% set remap	99	106	100	**95**	**94**	**94**	**92**	98	106	100	**95**	**95**	**93**	**92**	98	106	100	**94**	**93**	**92**	**91**
99% set remap	94	106	100	91	91	86	89	93	106	100	90	90	84	88	93	106	100	90	90	84	88

(c) (N_{dwl}, N_{dbl}, N_{spd})	(1, 4, 0.5)							(4, 4, 0.5)							(1, 4, 0.125)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	55	99	100	55	55	30	55	51	100	100	51	51	26	51	51	100	100	51	51	26	51
No redun. ECC	71	103	100	68	68	48	65	69	104	100	66	66	46	64	68	104	100	66	66	45	63
12.5% redundancy	69	107	100	64	64	44	59	68	108	100	63	63	42	58	69	108	100	64	64	44	59
25% redundancy	81	115	100	72	72	58	64	81	116	100	70	70	57	60	82	116	100	72	72	59	64
12.5% redun. ECC	96	112	100	82	82	79	71	94	113	100	83	83	78	74	95	113	100	82	82	78	71
25% redun. ECC	95	120	100	78	78	74	65	95	121	100	78	78	74	65	95	121	100	79	79	75	65
95% delete	87	101	97	84	82	73	82	87	102	98	85	85	74	84	86	102	97	84	84	73	83
99% delete	82	101	100	80	80	66	79	81	102	100	79	79	64	78	82	102	100	80	80	66	79
95% set remap	99	103	100	**95**	**94**	**94**	**92**	98	103	100	**95**	**95**	**93**	**92**	98	103	100	**95**	**95**	**93**	**92**
99% set remap	93	103	100	90	90	84	88	94	103	100	91	91	86	89	93	103	100	90	90	84	88

ECC. This result shows the benefit of set remapping. Lastly, different organizational parameters lead to different yields and areas. YAP can capture these differences. Nevertheless, in our experiments, the difference is very small; the YAP values differ from 1% to 4% for each metric.

To judge the computational demands of our tools, we measured the speed of DEFCAM by continuously running simulations for 20 hours. With 10 cache designs, 1,000 samples from 10 wafers, and 7 defect-process-organization variation configurations, the design space has 70,000 caches. Using four 3.4GHz Intel Xeon-based Linux boxes, DEFCAM evaluated all caches with no redundancy and redundancy schemes. For evaluating caches with delete and set remapping, DEFCAM could simulate roughly 10% of all caches, that is, 100 out of 1,000. This sampling results in a maximum error of as low as 0.2% at the 95% confidence interval. This rate is fast enough to evaluate a large design space overnight. The case study demonstrates how our flow and tools can be used to evaluate yield, area and performance of many cache designs. It also highlights the importance that set remapping schemes will play in yield management for future cache designs and technologies.

7. RELATED WORK AND FUTURE PLAN

There is a large body of previous work on defect modeling and yield analysis [Milor 1999; Segal et al. 1999; Berglund 1996; Stapper and Rosner 1995; Maly 1985]. Physical defects have been a major factor undermining yield. Maly and Deszczka [1983] developed a random defect model to compute defect distributions. Their model was later extended to accommodate a submicron domain [Pleskacz and Maly 1997]. Dekker et al. [1988] developed an efficient SRAM fault model and a set of linear march-based test algorithms. Wang et al. [2005] categorized various defect models and established a defect injection procedure. Stapper and Rosner [1995] analyzed yield for different defect types which are manifested in various logic and circuit elements (e.g., DRAM, SRAM, CMOS logic, ASIC and CMOS and biCMOS microprocessors). However, they considered only cell-level faults and did not analyze defects affecting wires such as a bitline or wordline.

Process variations have become a major reliability threat in modern deep submicron (DSM) technologies, which are further classified into inter- and intra-die components [Borkar 2003; SEMATECH 2003]. With aggressive technology scaling, the random and correlated components of intradie variations exceed those of interdie variations [Duvall 2000]. Their work, however, considered only process variations (and not physical defects). Agarwal et al. [2005] analyzed the impact of process variations on cache yield and developed a direct-mapped cache structure to mask faulty memory cells by substituting an adjacent sound line for a faulty line. However, their technique is only for a direct-mapped cache and does not consider different fault manifestations (i.e., word-line, bit-line, or cell).

There are several previous works studying the impact of redundancy techniques on yield. Thomas and Anthony [1999] analyzed a set of redundancy schemes in terms of yield and performance. They, however, assumed a simple defect model with only defect density and probability of a faulty row. Shivakumar et al. [2003] developed a metric called *performance averaged yield* which accounts for both fully functional chips and those that exhibit some performance degradation. Unfortunately, their model only considers performance degradation due to defects in logic blocks performing computations (i.e., not in memory). Our work in this article studied the impact of memory array's defects in cache memory.

The interdependency between circuit performance and yield at the circuit level has been studied extensively. Datta et al. [2005] studied yield prediction methodology from the process delay variation and showed that yield can be improved by an unbalanced pipeline design. Kim et al. [2005] analyzed process variation effects on microarchitecture design and suggested optimal logic depth for a 1-wide processor due to delay variations. Tschanz et al. [2002] suggested bidirectional adaptive body bias (ABB) can be used to compensate for die-to-die parameter variations by applying an optimum body bias voltage to maximize the die frequency. Forward body bias (FBB) for critical path logic blocks has been studied to increase the operating frequency of a design [Narendra et al. 2002; Vangal et al. 2002; Oowaki et al. 1998]. However, none of these works deal with application-level performance.

A relatively small amount of work has been done for the system-level performance estimation from low-level defects and variations. Recently, Liang and Brooks [2006] presented a microarchitectural parameter model and a methodology for designing logic blocks to reduce the frequency impact due to variations for distinct structures, while minimizing IPC (instructions-per-cycle) loss. Marculescu and Talpes [2005] present GALS (globally asynchronous, locally synchronous) microarchitecture-level models for Within-Die (WID) process variability. Though they touch on system-level performance, none of these works have studied a fully integrated fault-tolerant design methodology that incorporates all the key parameters and metrics ranging from physical-level defects and process variations to application performance.

There are a few studies that examined the performance of a fault-degradable cache approach. Sohi [1989] looked at the performance impact of line deletion in a unified cache using a trace-driven simulation method. 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 [1993] extended Sohi's work by introducing a more systematic way to evaluate the impact of defects. However, these earlier studies simulate a rather simple unified cache configuration using ATUM traces and use miss rate as the only metric. Shirvani and McCluskey [1999] introduced "padded cache" which has a programmable row decoder to avoid accessing faulty blocks. Their design has non-negligible timing penalty for the programmable decoder and significant area overhead (11%). More recently, Lee et al. [2007] analyzed the performance impact of defects more rigorously using a generic fault degradable cache model methodology with an execution-driven simulation method. Our work in this article focuses on studying the interplay between cache yield, cache area, and program performance in an integrated framework and builds on these previous performance studies.

In addition to the YAP metric, clock frequency could be added in our future work. With the frequency metric, DEFCAM framework can be a more accurate tool for fault-tolerant cache architecture study. Furthermore, lifetime reliability has been an interesting topic in the fault-tolerant community. We also remain developing integrated framework including yield and lifetime reliability as our future work.

8. CONCLUSION

Traditional cache optimizations have focused on the performance, area, and power aspect of the resulting design. Guaranteeing fault tolerance and enhancing chip yield have been largely a separate effort made by low-level circuit quality engineers, layout designers, and process engineers. As chips built with a future deep submicron technology are more susceptible to manufacturing defects, process variations, and aging phenomena, it becomes imperative to consider reliability and yield together at an early design time by the processor architect/cache designer.

This article presented DEFCAM, a new cache design flow that integrates cache defect, yield, and performance models, and lays a solid foundation for evaluating a cache memory designed on nanometer-scale technology in terms of its yield, area, and performance. We also introduced a metric called YAP (yield-area-performance) which enables a designer to quickly evaluate different cache designs with a single number. Using DEFCAM and the YAP metric, a cache designer can directly compare cache designs having different architectural, organizational, and defect-related parameters at an early design stage.

With DEFCAM, we evaluated the performance impact from three different architecturally visible fault classes (line, set, and way) and discovered that masking set faults is crucial to reducing cache misses, and hence, minimizing performance impact. To tackle set faults, we proposed set remapping, which redirects memory accesses to faulty sets to available sound cache sets. We also showed that set remapping can be done statically or dynamically, using off-line or on-line profile information. Among these remapping policies, the static remapping policy is the simplest in terms of design complexity and it achieves much of the potential of an oracle based approach.

To illustrate how DEFCAM can be used in practice, we performed a case study to compare a number of cache yield management strategies, including: redundant row, ECC, line delete, and set remapping. When the redundant row and ECC schemes are used in isolation, they fall short of other degradation-based schemes quickly as the effect of defect and process variations is increased. Degradation-based schemes such as line delete and set remapping were shown to offer much higher yield with limited area overheads. Among the cache yield management strategies, we find that set remapping consistently offers the highest YAP metrics under various defect and process variation scenarios.

REFERENCES

Agarwal, A., Paul, B. C., Mahmoodi-Meimand, H., Datta, A., and Roy, K. 2005. A processtolerant cache architecture for improved yield in nanoscale technologies. *IEEE Trans. VLSI Syst. 13*, 1.

Albonesi, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 248–259.

Austin, T., Larson, E., and Ernst, D. 2002. Simplescalar: An infrastructure for computer system modeling. *IEEE Comput. 35*, 2, 59–67.

Berglund, C. N. 1996. A unified yield model incorporating both defect and parametric effects. *IEEE Trans. Semicond. Manuf. 9*, 3, 447–4 4.

Borkar, S. 2004. Microarchitecture and design challenges for gigascale integration. In *Proceedings of the speech at International Symposium on Microarchitecture (MICRO)*.

Borkar, S. E. A. June 2003. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the Design Automation Conference*. 338–342.

Bossen, D. C., Tendler, J. M., and Reick, K. 2002. Power4 system design for high reliability. *IEEE Micro 22*, 1, 16–24.

Datta, A., Mukhopadhyay, S., Bhunia, S., and Roy, K. 2005. Yield prediction of high performance pipelined circuit with respect to delay failures in sub-100nm technology. In *Proceedings of the IEEE International On-Line Testing Symposium*. 275–280.

Dekker, R., Beenker, F., and Thijssen, L. 1988. Fault modeling and test algorithm development for static random access memories. In *Proceedings of the International Test Conference (lTC)*. 343–352.

Duvall, S. G. Aug. 2000. Statistical circuit modeling and optimization. In *Proceedings of the 5th International Workshop on Statistical Metrology*. 56–63.

Friedman, D. J., Hansen, M. H., Nair, V. N., and James, D. A. 1997. Model-free estimation of defect clustering in integrated circuit fabrication. *IEEE Trans. Semicond. Manuf. 10*, 3, 334–359.

Guthaus, M., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workshop Workload Characterization (WWC).*

ITRS. 2005. International technology roadmap for semiconductors. http://public.itrs.net.

Kim, N. S., Kgil, T., Bowman, K., De, V., and Mudge, T. 2005. Total power-optimal pipelining and parallel processing under process variations in nanometer technology. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (lCCAD)*. 535–540.

Kumar, S., Kim, C., and Sapatnekar, S. 2006. Impact of NBTI on SRAM read stability and design for reliability. In *Proceedings of the International Symposium on Quality Electronics Design (ISQED)*.

LAMET, D. AND FRENZEL, J. F. 1993. Defect-tolerant cache memory design. In *Proceedings of the IEEE VLSI Test Symposium*. 159–163.

LEE, H., CHO, S., AND CHILDERS, B. 2007. Performance of graceful degradation for cache faults. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (lSVLSI)*. 409–415.

LIANG, X. AND BROOKS, D. 2006. Microarchitecture parameter selection to optimize system performance under process variation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. 429–436.

MALY, W. 1985. Modeling of lithography-related yield losses for cad of VLSI circuits. *IEEE Trans. Comput.-Aid. Des.* 3.

MALY, W. AND DESZCZKA, J. 1983. Yield estimation model for VLSI artwork evaluation. *Electron. Lett.* 226–227.

MARCULESCU, D. AND TALPES, E. 2005. Variability and energy awareness: A microarchitecture-Ievel perspective. In *Proceedings of the 42nd Conference on Design Automation*. 11–16.

MILOR, L. S. 1999. Yield modeling based on in-line scanner defect sizing and a circuit's critical area. *IEEE Trans. Semicond. Manuf.* 1, 26–35.

NARENDRA, S., HAYCOCK, M., GOVINDARAJULU, V., ERRAGUNTLA, V., WILSON, H., VANGAL, S., PANGAL, A., SELIGMAN, E., NAIR, R., KESHAVARZI, A., BLOECHEL, B., DERMER, G., MOONEY, R., BORKAR, N., BORKAR, S., AND DE, V. 2002. 1.1 v 1 GHz communications router with on-chip body bias in 150nm CMOS. *IEEE ISSCC Dig. Tech. Papers*, 270–271.

OOI, Y., KASHIMURA, M., TAKEUCHI, H., AND KAWAMURA, E. 1992. Fault-tolerant architecture in a cache memory controllsi. *IEEE J. Solid-State Circ. 4*, 507–514.

OOWAKI, Y., NOGUCHI, M., AND TAKAGI, S. 1998. A sub-0.1 um circuit design with substrate-over-biasing. *IEEE ISSCC Dig. Tech. Papers*, 88–89.

PATTERSON, D. A., GARRISON, P., HILL, M., LIOUPIS, D., NYBERG, C., SIPPEL, T., AND VAN DYKE, K. 1983. Architecture of a VLSI instruction cache for a rise. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 108–115.

PLESKACZ, W. A. AND MALY, W. 1997. Improved yield model for submicron domain. In *Proceedings of the Workshop on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2–10.

POUR, A. F. AND HILL, M. D. 1993. Performance implications of tolerating cache faults. *IEEE Trans. Comput. 3*, 257–267.

RUSU, S., TAM, S., MULIJONO, H., STINSON, J., AYERS, D., CHANG, J., VARADA, R., RATTA, M., AND KOTTAPALLI, S. 2009. A 45nm 8-core enterprise xeon processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

SEGAL, J., GORDON, A., SAJOTO, D., DUFFY, B., AND KUMAR, M. 1999. A framework for extracting defect density information for yield modeling from in-line defect inspection for real-time prediction of random defect limited yields. In *Proceedings of the IEEE International Symposium on Semiconductor Manufacturing (ISSM)*.

SEMATECH. Mar. 2003. Critical reliability challenges for itrs. Tech. transfer #03024377A-TR.

SHIRVANI, P. P. AND MCCLUSKEY, E. J. 1999. Padded cache: A new fault-tolerance technique for cache memories. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 440–445.

SRIVAKUMAR, P., KECKLER, S. W., MOORE, C. R., AND BURGER, D. 2003. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the International Conference on Computer Design (ICCD)*. 481–488.

SORI, G. S. 1989. Cache memory organization to enhance the yield of high-performance vlsi processors. *IEEE Trans. Comput*.

SRINIVASAN, J., ADVE, S. V., BOSE, P., AND RIVERS, J. A. 2004. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*.

STAPPER, C. H. AND ROSNER, R. J. 1995. Integrated circuit yield management and yield analysis: Development and implementation. *IEEE Trans. Semicond. Manuf.* 2.

TANG, T., DE, V., AND MEINDL, J. D. 1997. Intrinsic mosfet parameter fluctuations due to random dopant placement. *IEEE Trans. VLSI Syst.* 369–376.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. Cacti 4.0. Tech. rep. HPL-2006-86, AP Labs.

THOMAS, T. AND ANTHONY, B. 1999. Area, performance, and yield implications of redundancy in onchip caches. In *Proceedings of the International Conference on Computer Design (ICCD)*. 291–292.

TSCHANZ, J. W., KAO, J. T., AND NARENDRA, S. G. 2002. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE J. Solid-State Circ.* 422–478.

VANGAL, S., BORKAR, N., SELIGMAN, E., AND GOVINDARAJULU, V. 2002. A 2.5 GHz 32b integer execution core in 130 nm dual-vt cmos. *IEEE ISSCC Deg. Tech. Papers*, 412–413.

WADA, T., RAJAN, S., AND PRZYBYLSKI, S. A. 1992. An analytical access time model for on-chip cache memories. *IEEE J. Solid-State Circ. 8*, 1147–1156.

WANG, X., OTTAVI, M., MEYER, F. J., AND LOBBARDI, F. 2005. Estimating the manufacturing yield of compiler-based embedded srams. *IEEE Trans. Semicond. Manuf. 3*, 412–421.

YE, W., VIJAYKRISHNAN, N., KANDEMIR, M., AND IRWIN, M. J. 2000. The design and use of simple power: A cycle-accurate energy estimation tool. In *Proceedings of the Design Automation Conference (DAC)*.

ZHANG, C. 2006. Balanced cache: Reducing conflict misses of direct-mapped caches through programmable decoders. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 155–166.

Received January 2009; revised February 2010, January 2011; accepted July 2011