Customizable Fault Tolerant Caches for Embedded Processors

Subramanian Ramaswamy and Sudhakar Yalamanchili
Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
ramaswamy@gatech.edu, sudha@ece.gatech.edu

Abstract— **The continuing divergence of processor and memory speeds has led to the increasing reliance on larger caches which have become major consumers of area and power in embedded processors. Concurrently, intra-die and inter-die process variation at future technology nodes will cause defect-free yield to drop sharply unless mitigated. This paper focuses on an architectural technique to configure cache designs to be resilient to memory cell failures brought on by the effects of process variation. Profile-driven re-mapping of memory lines to cache lines is proposed to tolerate failures while minimizing degradation in average memory access time (AMAT) and thereby significantly boosting performance-based die yield beyond that which can be achieved with current techniques. For example, with 50% of the number of cache lines faulty, the performance drop quantified by increase in AMAT using our technique is 12.5% compared to 60% increase in AMAT using existing techniques.**

I. Introduction

This paper addresses the confluence of two challenges facing the design of embedded processors as the industry enters the deep sub-micron region of semiconductor design. The first challenge is posed by process variation (both inter-die and intra-die) at future technology nodes that will cause defect-free yield to drop sharply. The causes of process variation continue to be the focus of many studies (for example see [1]–[3]) and significant effort is being devoted to innovations in manufacturing and circuit technologies(e.g., adaptive body biasing [4], [5]) to reduce its impact. The second challenge is the continuing divergence of processor and memory speeds. This has led to the increasing reliance on larger caches which have become dominant consumers of area and power [6]. This paper proposes architectural techniques for the design of robust cache architectures that can complement and utilize existing and anticipated circuit and manufacturing advances to combat process variation.

Intra-die process variations cause access time and read/write stability failures for SRAM memory cells [7]. Lowering supply voltages to reduce energy requirements exacerbates these failure modes [8]. Our goal is to make the cache resilient to cell failures by using the available fault-free area of the cache while minimizing additional investments in area and power. Further, the manner in which the available fault-free area is utilized attempts to minimize the performance degradation due to faulty memory cells. The challenges lay in the interactions between competing goals. The AMAT delivered by the cache is a function the number of non-faulty cache lines. Lower supply voltages to lower the power demands will reduce the number of non-faulty lines. Adopting larger caches to increase the average number of non-faulty lines increases both area and power requirements - particularly undesirable in embedded processors.

This paper proposes a micro-architecture level technique for improving the yield of cache memory designs in a manner that is transparent to application software. Yield refers to the number of die that can meet target performance goals - in this paper we define performance yield based on the AMAT. The approach builds on a key idea proposed in [9] and [10] where cache placement policy is modified to map main memory lines to non-faulty cache lines. Our approach extends and differs from these prior efforts in several ways. First, we treat the problem as an optimization problem - the assignment of main memory lines to cache lines/sets is driven by an application memory reference profile. Profiles are analyzed to extract temporal usage information for each cache line. This information is used to make global decisions about how memory addresses map to cache lines and produce a placement function that is customized to the memory reference profile and distribution of faulty cells. The result is a tighter control of AMAT as the number of faulty cells vary and an effective greater improvement in the yield. Second, the placement is under compiler control and can be varied across applications or phases within an application, or even regions within an application, e.g., loop nests or functions/procedures. In general, one can conceive of the use of powerful program analysis techniques based on the analysis of dependences and data-flows to produce customized placement information. This generalization is not treated in this paper and is the subject of ongoing work.

Section II provides an overview of related research to date, concluding with an assessment of the state of the practice motivating the proposed approach. Section III, provides a description of the fault tolerant cache (FTC) design and the associated methodology for configuring the cache placement based on application profiles. The remainder of the paper summarizes the results of a quantitative design space exploration to quantify the effectiveness of the design methodology and

1-4244-9707-X/06/$20.00 ©2006 IEEE

establish directions for future work. The evaluation reports in an exploration of a design space of AMAT, miss rates, failure rates, and cache sizes to quantify the improvement in yield and the corresponding degradation in performance, the latter being surprisingly low for even a modest number of non-functional cache lines. Section VI provides some concluding remarks and a discussion of ongoing and future work specifically pointing out the opportunities for using the proposed technique to impact cache power dissipation.

II. Related Work

Our approach extends the techniques proposed in [9], [10] where faulty cache lines/blocks are mapped to neighboring cache lines/blocks. This assignment is performed once and remains fixed. The goal of those efforts was fault tolerance while performance optimization was not addressed. The scope of their techniques is limited by the block size (not the same as cache line size). Traditional techniques of column/row redundancy are limited, for example in the number and distribution of faulty cells that can be tolerated, and as described in [10], are improved upon by approaches that utilize re-mapping.

Other approaches include avoiding faulty ways in a set associative cache [11] or placing code intelligently in faulty set associative caches to minimize misses [12]. An early proposal treated accesses to a faulty cache line as a miss [13], i.e. memory lines which mapped to faulty cache lines were never brought into the cache. This technique predictably causes rapid degradation of cache performance as the number of faults increase. Adding redundant blocks [14], [15] and using error correcting codes [16] also provide a certain degree of fault tolerance but remain limited in the number of faults that can be tolerated. As industry moves into the deep sub-micron region, an approach that accommodates graceful degradation across a wider range of process parameters is desirable. In particular, concurrent optimization of performance and fault tolerance is desired. Such an approach is described in this paper.

III. Fault Tolerant Cache (FTC) Architecture

The operation of the fault tolerant cache is based on a simple principle - main memory lines which are mapped to faulty cache lines are mapped to non-faulty cache lines. The determination of this mapping or *placement* is driven by application reference profiles and is under compiler control. This paper addresses the implementation of fault tolerant direct mapped caches. Their small hardware footprint and fast hit times make them attractive for embedded processors. However, it will be evident that the techniques have natural extensions to set associative caches.

A. Placement Model

We target embedded processors that typically have a single level cache and where servicing a miss from off-chip memory is typically two orders of magnitude slower, e.g., 300 cycles in the IXP 2800 [17]. In a traditional cache architecture (Figure 1) the placement policy assigns memory line L to cache line L mod S where there are S lines in the cache.

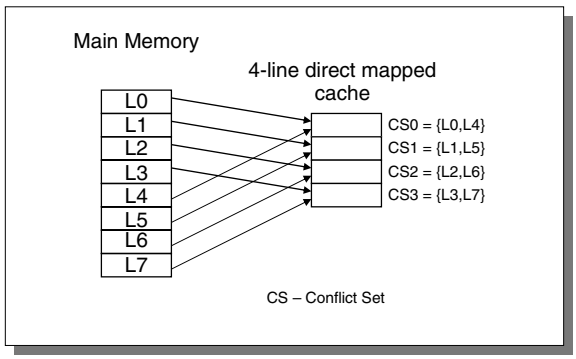

Fig. 1. Cache with Traditional Placement

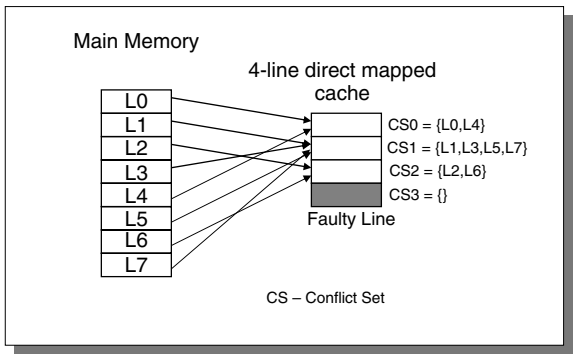

Fig. 2. Cache with Customized Placement

Based on this placement policy, main memory can be viewed as being partitioned into *conflict sets*. Each conflict set is the set of all memory lines that are mapped to the same line in the cache. In traditional caches, all conflict sets have the same cardinality.

Now consider a direct mapped cache, a specific operating supply voltage, and consequently the failure of some memory cells (as defined in [10]). Assuming a RAM Tag implementation, tags and lines with failed cells are marked as faulty lines. Now consider a direct mapped cache where there are f such faulty lines. If a tag is faulty, the corresponding line is treated as faulty. The optimization problem is to find a new assignment of the S conflict sets to the $S-f$ non-faulty cache lines. The assignment is computed with the goal of minimizing conflict misses. Conflict sets from the non-faulty cache that are mapped to the same cache line in the faulty cache are *merged*. Thus, we now have $S-f$ conflict sets in the fault tolerant cache. An example of profile-driven customized placement is illustrated in Figure 2. The challenges are i) the development of algorithms to determine the most effective placement policy, and ii) the efficient implementation of address translation mechanisms for the customized placement policy. The former is driven by the locality properties of the reference profile measured as described in the following section.

B. Capturing Reference Locality

Program execution has been observed to evolve through *phases* [18] with each phase being characterized by a working set of memory references. Within a phase, program reference behavior exhibits spatial and temporal reference locality around a set of memory locations. *Group temporal* locality has been defined as behavior wherein memory references in a phase that are not spatially local, are temporally local, i.e., they are clustered in time [19]. For example, when access to a data element allocated on the heap is strongly correlated (i.e., invariably followed by) an access to a local variable allocated on the stack. Studies have proposed various metrics [19]–[21] for capturing group temporal locality.

In the presence of faulty cache lines, multiple conflict sets are mapped to a non-faulty cache line. In doing so, we expect that the number of conflict misses to this cache line will increase. The optimization problem is to find an assignment of conflict sets to cache lines that will minimize the increase in conflict misses in the FTC. Conflict sets that exhibit group temporal locality are poor candidates for sharing a cache line. To identify good candidates for sharing a cache line, we define an affinity metric called *interference potential* as a measure of group temporal locality between conflict sets.

A memory reference trace is partitioned into contiguous segments of references called *windows* where each window represents a program execution phase. For window w the number of references to cache line i is defined as $r(w,i)$. In this study, the interference potential between conflict sets i and j is $min(r(w,i), r(w,j))$ - representative of the potential increase in the number of conflict misses in window w if conflict sets i and j are mapped to the same cache line in that window. The interference potential between two conflict sets for the application is the sum of the interference potential between the conflict sets across all windows.

C. Fault Tolerant Placement Policies

We evaluate two placement policies for fault tolerant direct mapped caches. The first is a profile-agnostic policy that is representative of existing approaches to handle faulty cache sets [9], [10]. The second is a proposed profile-driven placement policy customized from a reference stream.

1) Modulo Placement: Modulo placement is employed in existing fault-free caches and is representative of fixed alternatives realized by existing proposals for by-passing faulty cache sets or lines [9], [10]. Consider a cache with f faulty lines and $S-f$ non faulty lines that are addressed contiguously 0 through $(S-f)$. Main memory line L is mapped to $L \ mod \ (S-f)$. The practical difficulty is performing modulo $(S-f)$ arithmetic on every cache access. Since the fault pattern is not known a priori, address translation must be programmable. Thus, the decoder has to be programmed such that faulty sets in the cache are bypassed. Similar implementations are described in [9], [10]. For comparison purposes we use the implementation of address translation for custom placement to also implement modulo placement. This is shown in Figure 3, where a lookup table is used to perform the modulo function and bypass faulty lines.

Depending on the fault pattern, two lines with the same tag can now map to the same non-faulty cache line. Therefore to ensure unique tags across all memory lines that map to cache line, the new tags are comprised of the old tags concatenated with the index bits. For example, in Figure 2, if the line size was 16 bytes, memory addresses 0x00000010 and 0x00000070 map to the same line in the faulty cache (line 1), but the original higher order tag bits (0x000000 in both cases) are not sufficient to identify the memory location uniquely. If the index bits are concatenated with the old tag, the tags are now 0x0000001 and 0x0000007 respectively, which can be used to differentiate the memory lines.

2) Customized Placement: The goal of customized placement is to map conflict sets to non-faulty lines so as to minimize conflict misses. The algorithm for determining such a placement is outlined in Algorithm 1. The number of conflict sets in the original cache is S, which is equal to the number of cache lines in a direct-mapped cache and the total number of windows is W. The input to the algorithm consists of the reference count per cache line per window for all cache lines, $r[S][W]$, the interference potential matrix, $ip[S][S]$ and the number of faulty lines f. Algorithm lines 1-3 initialize the placement, by mapping each conflict set in the original fault-free cache to the corresponding cache line. This if followed by an iterative traversal of the interference potential matrix to i) select the conflict set pair with the minimum interference potential, ii) merge them to form a new conflict set (lines 5-6) and iii) update the interference potential matrix and reference counts (lines 7-8). The algorithm terminates after f merges, i.e. the number of new conflict sets have been made equal to the number of fault-free cache lines. At this point the S conflict sets in the fault-free cache have been assigned to the $S-f$ cache lines. Note that multiple conflict sets in the fault-free cache can be mapped to a single cache line. The algorithm returns a placement $map[S]$, which maps conflict sets in the original cache to fault-free cache line. *Note that all conflict sets are re-mapped - not just those sets that are mapped to faulty-cache lines!* The only requirement is that there are f merge operations. This enables a tighter control of AMAT as the number of faulty lines increases. Hence, a final update of the $map[S]$ array is required (line 9), where the mappings of the conflict sets are updated such that they point to fault free lines in the cache.

Address translation is achieved using a lookup table as shown in Figure 3. Main memory addresses are translated via a lookup table, which is indexed by the original cache index (i.e. the index for the fault-free case), the output of which contains the new cache line to which a memory address is mapped. The width of the tag array is increased by the size of the index to differentiate distinct memory lines as in the case of the modulo fault tolerance scheme. Faulty cache lines are never activated.

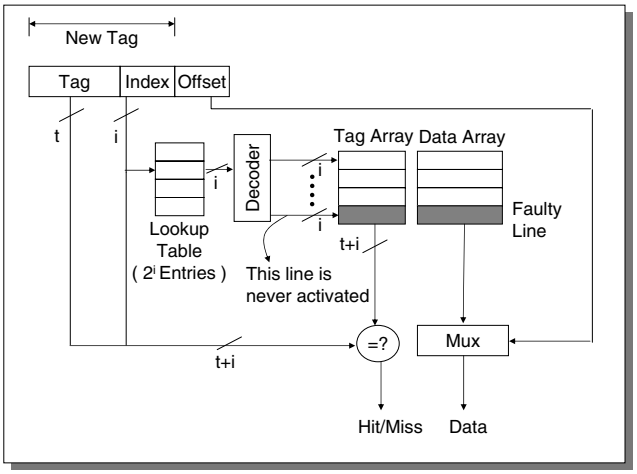

Fig. 3. Custom Placement Implementation

Algorithm 1 Placement Algorithm for FTC

Input: $r[S][W], f, ip[S][S]$
Output: $map[S]$

1: **for** $iter$ = 0 to $S-1$ **do**
2: $map[iter] = iter$ {Initialize}
3: **end for**
4: **for** $iter$ = 1 to f **do**
5: find i, j s.t. $ip[i][j] = min(ip[S][S])$ {Find conflict sets having minimum ip}
6: $map[j] = map[i]$ {Merge the two conflict sets}
7: $update(r[S][W])$ {Update reference counts}
8: $update(ip[S][S])$ {Update the ip matrix}
9: **end for**
10: $update(map[S])$ {Update to remove mapping to faulty lines}
11: **return** $map[S]$

IV. Evaluation Methodology

The probability of a cache line being faulty is assumed to be normally distributed with mean μ and standard deviation σ.

The following assumptions underly our model. We assume the fault detection model in [10], where the cache has BIST circuitry which identifies defects and errors post fabrication and marks faulty SRAM cells. Access time failures or read/write failures due to process variation as considered faulty behavior. If a single bit in a cache line or tag is faulty the entire line is marked faulty. An S-bit register records the result of the BIST operation. The contents of this register can be read by compiler/configuration software. The customized placement is loaded by the compiler into the lookup table.

The fault tolerant cache placement schemes were simulated using *valgrind* [22]. The area, latency and power estimates were derived using *cacti* [23]. The kernels that were analyzed belong to the *mibench* embedded benchmark suite [24] which covers a broad domain of embedded system kernels. With a target of embedded processors, we focus on smaller caches where the impact of parameter variations is expected to (relatively) higher and the benefits of the proposed approach the greatest. The miss penalty used in our analysis was as 100 cycles for a 32-byte cache line predicated on off-chip DRAM accesses of 100-300 cycles [17]. The miss penalties for 64-byte and 128-byte cache lines were 108 and 124 cycles respectively, with the memory bank model used in [25]. We used a fixed window size in our analysis, 100000 references, which was about 1% of the trace length for most of the *mibench* kernels.

V. Results and Analysis

A. Fault Tolerance

Figure 4 compares the performance of a fault tolerant cache using a modulo scheme and one using the customized placement scheme. It is observed that the performance degradation for the customized placement cache in terms of AMAT is less than 5% with 12.5% of the cache faulty, and the degradation is only 20% when 50% of the cache is faulty, compared to degradations of 10% and 60% for the modulo scheme respectively. As the percentage of cache area that is faulty nears 100%, the difference between the two schemes will be less noticeable, as both techniques will yield very high miss rates. The better performance of the customized placement cache is due to the more efficient sharing of cache resources among memory lines. Since, the conflict sets that are grouped together (equivalently $merged$) have a relatively lower interference potential, the impact on miss rates and hence AMAT and cache performance are reduced.

From Figure 6, it is observed that for larger caches using customized placement, the AMAT remains flat for a higher percentage of faulty cache area. This is because, the AMAT will not be affected as long as the number of fault-free lines in the cache is larger than the application footprint. Thus, the slope of the curves decrease as the cache size increases. The AMAT shown in Figures 4 and 6 represent AMAT averaged over the *mibench* kernels. Figure 5 captures the performance variation of the individual *mibench* kernels with faults. From Figure 5, it is seen that for certain benchmarks, the performance degradation is less than 5% even with 50% of the cache area being faulty using the customized placement FTC. Thus customized placement shares cache resources effectively, that even with the effective cache being halved, there is very little ($< 5\%$) performance degradation. The trends were found to be similar with varying cache line sizes.

B. Area, Latency and Power

The area and latency impact of the fault tolerant cache were measured using *cacti* [23]. The area impact for a 4K direct mapped cache with 32 byte lines is a lookup table consisting of 128 7-bit entries plus an additional $128*7$ bits of increased tag store and a 128-bit register where the fault status of the cache lines are stored which together constitute approximately 5% of the overall cache area. For a 16K cache with 128 byte lines, the increased storage is again 310 bytes, which is less than 2% of the overall cache area.

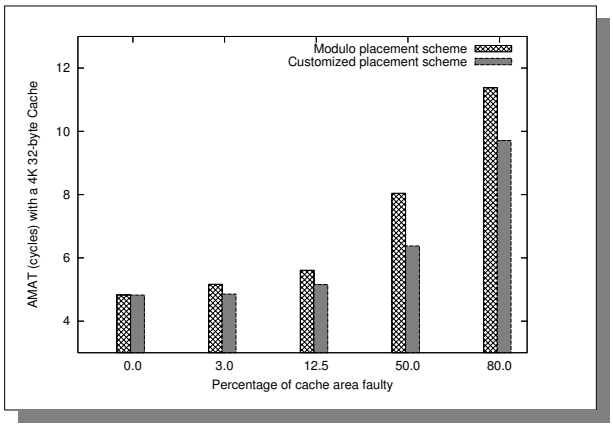

Fig. 4. Modulo Vs Custom Placement

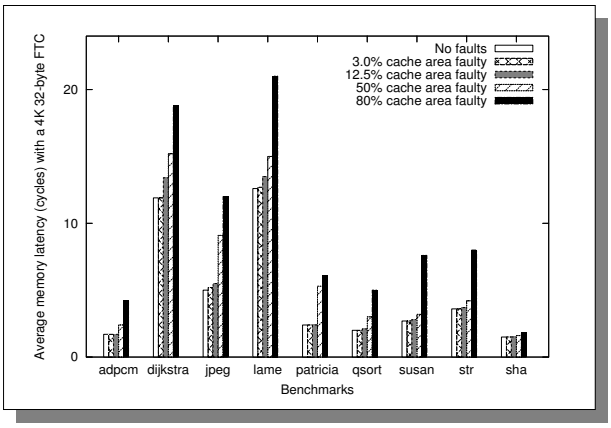

Fig. 5. AMAT Variation with Faults

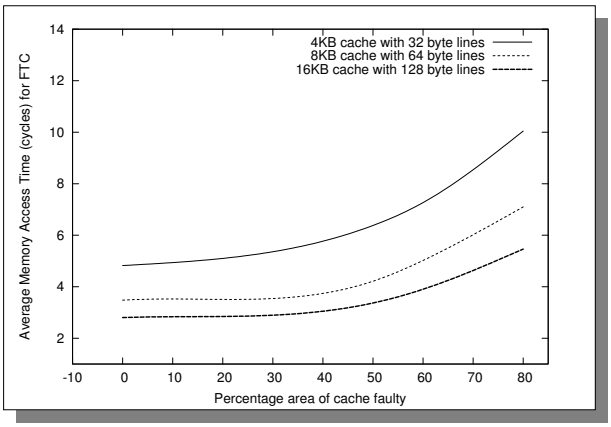

Fig. 6. AMAT Variation with Faults for Various Cache Configurations

The access time for an 4K direct mapped cache with 32 byte lines is 0.48 ns (0.58 ns for a 16KB, 128 byte line cache) using 90 nm technology, whereas the added latency for the lookup is 0.23 ns (comprising mainly of the decoding latency necessary for indexing the lookup table). Thus, the combined access time of the variable placement cache is around 0.7 ns. Though the added latency is significant, for an embedded processor running at 1.2 GHz, the access can be performed within one cycle. However, modern embedded processors typically operate at 250-500 MHz and therefore the separate lookup stage will not adversely affect the AMAT. Larger the cache size or higher the associativity, the impact of the additional latency will be reduced considerably (e.g. for a 16KB cache, the lookup table increases latency by 30%, as opposed to nearly 45% for a 4KB cache). Note that for the modulo fault tolerant scheme, the hardware implementation of the address translation is also on the critical path, although the impact is not significant.

Since the lookup table size is very small compared to the cache size, and additionally, there are no tags or muxes required, the energy consumed by the lookup table is again concentrated in the decoding circuitry. This adds about 12% power increase to a 4K cache, and the impact again diminishes with larger caches and caches with higher associativity (e.g. for a 4K 4-way cache, the increase is approximately 2.5%, and for a 16 KB cache the increase is 8%).

C. Performance Yield

Performance yield recognizes that at the micro-architecture level yield is measure of the ability to meet performance goals and it is not a measure of identification of defect-free substructures. Thus we measure yield as the percentage of implementations (for convenience we use the term die) that produce an AMAT that deviate no more than 5% from the non-faulty die. The simulations were carried out for 1000 dies and for various μ, σ values of the number of faulty cache lines. The measured AMAT was a averaged across all of the kernels. Those die whose AMAT was within 5% of the averaged AMAT for non-faulty designs were classified as usable die. The results are summarized in Table I.

From Table I, it is observed that the number of usable dies is substantially higher for the customized placement scheme over the modulo placement scheme. For a distribution with $\sigma = 8$ and $\mu = 8$, it is seen that die yield is over 85% for the customized placement scheme compared to 27% for the modulo placement scheme. The difference in the number of usable dies is noticed to be significant irrespective of the fault distribution. If the application footprint fit into the set of fault free lines, the difference in yield is not expected to be significant. However, in practice we expect this to be rarely the case. Further, the results suggest a strategy of using smaller caches to save area and power with minimal sacrifice is performance. This performance loss may well be recovered by putting the recovered silicon area to good use, e.g., larger register file.

Note the importance of the definition of yield - it includes a measure of performance. This is a more stringent requirement than simply requiring that the design tolerate faults and enable the die to be functional. Therefore, if a weaker definition of yield sufficed, the number of usable die would be higher. Finally, a better definition of yield would include binning by

μ, σ	Dies usable with Mod. Placement	Dies usable with Cust. Placement
1,2	921	998
1,8	380	958
2,2	820	990
4,4	604	984
4,8	355	925
8,8	279	859
16,16	130	612

TABLE I
PERFORMANCE YIELD COMPARISON

frequency and power (leakage). That is the subject of current efforts.

VI. FUTURE EXTENSIONS & CONCLUDING REMARKS

While this paper centered around direct mapped caches, the extensions to set-associative caches are straight-forward - merging conflict sets to share fault free sets and to share sets where some lines may be faulty. An unexplored dimension is to consider finer granularity of optimization by splitting and re-composing conflict sets. Thus a heavily used conflict set may be assigned multiple lines in the cache or multiple sets in a set associative cache. Further once the placement function is placed under compiler control, optimization may span program regions and become subject to run-time manipulation and improvement. It remains to be seen whether such fine grained management will be productive.

From Figure 6, it is noted that the average AMAT across the benchmark kernels degrades by 20% when 50% of the cache lines in a direct mapped cache are faulty and this envelope is pushed further for larger caches. The AMAT degradation is negligible when 15% of the cache area is faulty. For many individual kernels (e.g. sha, qsort, lame, susan etc.), it is observed that the degradation is very low even with 50% of the cache area being faulty. Lowering Vdd, produces power savings, but increases error rate and thus adversely affect performance (AMAT). Since the customized placement scheme mitigates the effect of errors on AMAT, we hypothesize that this can be used to further lower Vdd. A separate branch of our work is focussed on adaptive power management built on accurate models of devices that incorporate failure modes, leakage power, short circuit power etc. The goal is to adaptively trade-off bit error rate, power, and performance inspired by the work with microprocessor datapaths [26]. A convergence of these techniques with the datapath techniques can produce robust single chip embedded processor power management strategies for deep sub-micron implementations.

REFERENCES

[1] X. Tang, D. V.K., and M. J.D, "Intrinsic MOSFET parameter fluctuations due to random dopant placement," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 369–376, December 1997.

[2] J. Tschanz, K. A. Bowman, and V. De, "Variation-tolerant circuits: circuit solutions and techniques." in *DAC*, 2005, pp. 762–763.

[3] D. E. Hocevar, P. F. Cox, and P. Yang, "Parametric yield optimization for MOS circuit blocks," *IEEE Transactions on Computer Aided Design*, vol. 7, no. 6, pp. 645–658, 1988.

[4] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture." in *DAC*, 2003, pp. 338–342.

[5] S. Narendra, D. Antoniadis, and V. De, "Impact of using adaptive body bias to compensate die-to-die variation on within-die variation," in *ISLPED*, 1999, pp. 229–232.

[6] M. Zhang and K. Asanovi, "Fine-grain CAM-tag cache resizing using miss tags," in *ISLPED*, 2002, pp. 130–135.

[7] S. Mukhopadhyay, H. Mahmoodi-Meimand, and K. Roy, "Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1859–1880, 2005.

[8] P. Ndai, A. Agarwal, Q. Chen, and K. Roy, "A soft error monitor using switching current detection," in *ICCD*, 2005.

[9] P. P. Shirvani and E. J. McCluskey, "PADded cache: A new fault-tolerance technique for cache memories," *VTS*, vol. 00, p. 440, 1999.

[10] A. Agarwal, B. C. Paul, and K. Roy, "A novel fault tolerant cache to improve yield in nanometer technologies." in *IOLTS*, 2004, pp. 149–154.

[11] O. Y., M. Kashimura, H. Takeuchi, and E. Kawamura, "Fault-tolerant architecture in a cache memory control LSI," *IEEE Journal on Solid-State Circuits*, vol. 27, no. 4, pp. 507–514, April 1992.

[12] H. R. Zarandi, S. G. Miremadi, and H. Sarbazi-Azad, "Fault detection enhancement in cache memories using a high performance placement algorithm," *IOLTS*, vol. 00, p. 101, 2004.

[13] D. A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. V. Dyke, "Architecture of a VLSI instruction cache for a RISC," in *ISCA*. IEEE Computer Society Press, 1983, pp. 108–116.

[14] P. R. Turgeon, A. R. Steel, and M. R. Charlebois, "Two approaches to array fault tolerance in the IBM enterprise system/9000 type 9121 processor," *IBM J. Res. Dev.*, vol. 35, no. 3, pp. 382–389, 1991.

[15] D. Nokolos, "Performance recovery in direct-mapped faulty caches via the use of a very small fully associative spare cache," in *IPDS*. IEEE Computer Society, 1995, p. 326.

[16] H. L. Kalter, C. H. Stapper, J. E. B. Jr., J. DiLorenzo, C. E. Drake, J. A. Fifield, G. A. K. Jr., S. C. Lewis, W. B. van der Hoeven, and J. A. Yankosky, "A 50-ns 16-mb DRAM with a 10-ns data rate and on-chip ECC," *IEEE Journal on Solid-State Circuits*, vol. 25, no. 5, pp. 1118–1128, October 1990.

[17] *IXP2800 Network Processor Hardware Reference Manual*, Intel Corporation, November 2002.

[18] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, 2003.

[19] P. Petrov and A. Orailoglu, "Towards effective embedded processors in codesigns: customizable partitioned caches." in *CODES*, 2001, pp. 79–84.

[20] R. M. Rabbah and K. V. Palem, "Data remapping for design space optimization of embedded memory systems." *ACM Transactions in Embedded Computing Systems*, vol. 2, no. 2, pp. 186–218, 2003.

[21] K. Hazelwood, M. C. Toburen, and T. M. Conte, "A case for exploiting memory-access persistence," in *Workshop on Memory Performance Issues*, June 2001.

[22] "Valgrind tool suite version - 2.1.2." [Online]. Available: http://www.valgrind.org

[23] "CACTI, HP-Compaq Western Research Lab." [Online]. Available: http://research.compaq.com/wrl/people/jouppi/CACTI.html

[24] D. E. Matthew R Guthaus, Jeffrey S Ringenberg, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 1–12.

[25] Hannessy and Patterson, *Computer Architecture - A Quantitative Approach, 3rd Edition*. Morgan Kauffman, May 2002.

[26] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *IEEE Micro*, December 2003. [Online]. Available: http://www.gigascale.org/pubs/426.html