Multi-Layer Memory Resiliency

Invited Paper in Special Session "Embedded Resiliency: Approaches for the Next Decade"

Nikil Dutt[1], Puneet Gupta[2], Alex Nicolau[1],
Abbas BanaiyanMofrad[1], Mark Gottscho[2], Majid Shoushtari[1]

Department of Computer Science[1]
University of California, Irvine
Irvine, CA 92697
{dutt,nicolau,abanaiya,anamakis}@uci.edu

Department of Electrical Engineering[2]
University of California, Los Angeles
Los Angeles, CA 90095
puneet@ee.ucla.edu, mgottscho@ucla.edu

ABSTRACT

With memories continuing to dominate the area, power, cost and performance of a design, there is a critical need to provision reliable, high-performance memory bandwidth for emerging applications. Memories are susceptible to degradation and failures from a wide range of manufacturing, operational and environmental effects, requiring a multi-layer hardware/software approach that can tolerate, adapt and even opportunistically exploit such effects. The overall memory hierarchy is also highly vulnerable to the adverse effects of variability and operational stress. After reviewing the major memory degradation and failure modes, this paper describes the challenges for dependability across the memory hierarchy, and outlines research efforts to achieve multi-layer memory resilience using a hardware/software approach. Two specific exemplars are used to illustrate multi-layer memory resilience: first we describe static and dynamic policies to achieve energy savings in caches using aggressive voltage scaling combined with disabling faulty blocks; and second we show how software characteristics can be exposed to the architecture in order to mitigate the aging of large register files in GPGPUs. These approaches can further benefit from semantic retention of application intent to enhance memory dependability across multiple abstraction levels, including applications, compilers, run-time systems, and hardware platforms.

1. INTRODUCTION

The advent of many-core computing platforms exacerbates the classical processor-memory performance bottleneck. Traditionally, memory hierarchies have attempted to address this performance bottleneck by keeping frequently accessed data close to where they are consumed (e.g., by caching). However, contemporary design processes also need to guarantee other non-functional constraints such as power, energy and thermal bounds. Furthermore, since memories occupy a significant percentage of a chip's area, the memory subsystem has become vulnerable to a host of manufacturing, environmental, and operational failure/degradation mechanisms that affect the overall resiliency of the system. This paper outlines memory resilience challenges and opportunities across and between multiple levels of abstraction in a typical hardware/software design flow for computing systems (see Figure 1). The overall discussion is focused on systems-on-chip (SoCs), although similar analyses can be made for large-scale distributed systems as well. Section 2 describes memory abstractions across the design hierarchy shown in Figure 1, the typical causes of memory errors, and error manifestations at each abstraction level. Sections 3 and 4 use memory voltage scaling and wearout, respectively, as exemplars for multi-layer memory resiliency approaches. Section 5 outlines challenges for managing manufacturing variability and describes memory-related efforts within the NSF Variability Expedition project that aims to opportunistically exploit and manage hardware variability through software mechanisms. Section 6 closes with the outlook for multi-level memory resilience.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2596684

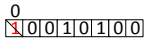
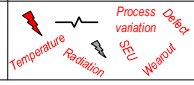
System Abstraction	Memory Abstraction	Error Manifestation	Opportunity
Application	*Program, Data Structures, Files, Libraries*	 *Incorrect Output, Infinite Loop, Crash*	*Trade Performance or Accuracy for Energy Saving*
Operating System	*Main Memory, File System Address Space, Heap, Stack*	 *Wrong Pointer, Erroneous System Call, Trap*	*Exploit Memory Mapping for Reliable vs Unreliable Pages*
ISA/RTL/ Arch	*Buffers, Register File, L1$, L2$, SPM*	 *Faulty Word, Cache Block, Way*	*Approximate Data Storage*
Logic	*Memory Cells, Bit Arrays*	 *Bit flip, Stuck @ 0/1*	*Operate at Lower Precision*
Circuit/ Device	*Voltage, Current, Transistor, Cell*	 *Low Noise Margin, Unstable Cell, V_{th} Variation*	*Relax Hardware Guardbands*

Figure 1. Memory Abstractions, Errors, and Opportunities.

This paper is part of the DAC special session on "Embedded Resiliency: Approaches for the Next Decade". Other papers in this session are: "Monitoring Reliability in Embedded Processors – A Multi-layer View" [68], "Multi-Layer Dependability: From Microarchitecture to Application Level" [69], and "Workload- and Instruction-Aware Timing Analysis – The missing Link between Technology and System-level Resilience" [70].

2. MEMORIES AND ERRORS

Figure 1 shows the typical hardware/software abstraction layers for computing systems. Each row of Figure 1 describes the system abstraction layer, the memory abstraction at that level, and typical manifestations of memory errors that can compromise system resiliency. The last column of Figure 1 describes opportunities for relaxed and approximate computing in the face of memory error manifestations at that level of abstraction. Memory errors manifest themselves in different ways across abstraction stack. For instance, an unstable memory cell at the circuit/device level can cause a bit failure at the memory logic level, which in turn might propagate up the abstraction stack as a faulty memory access at the architecture level, a wrong function

call or system halt at OS-level, and finally an output error or an exception at application layer.

Figure 1 represents a symbolic abstraction of memory errors over the entire hardware/software system stack. Traditionally, memory resilience has been addressed via disparate techniques at each level of design abstraction, while newer efforts attempt to couple strategies across layers with the goal of improving system efficiency for energy, heat dissipation, lifetime, cost, etc. Furthermore, efforts in relaxed and approximate computing attempt to create designs that can trade off application quality for these system efficiency goals.

To understand memory faults, we can classify them by their temporal behaviors (persistence) as well as their causes. With respect to persistence, a memory fault can be *permanent* or *transient*. Permanent faults persist indefinitely in the system after occurrence, while transient faults manifest for a relatively short period of time after occurrence. Furthermore, causes of memory faults can be *hard* or *soft*. Hard faults are static and caused by device failure or wear-out failure. In contrast, soft faults are dynamic and are typically caused by the operating environment.

Memories suffer from different sources of unreliability that can be classified into three main groups:

- *Manufacturing.* Worsening manufacturing imperfections in nanoscale technologies result in increasing variability of device and circuit-level parameters. This process variation particularly affects transistor threshold voltages through random dopant fluctuation (RDF), increasing the likelihood of memory cells failing permanently due to insufficient noise margins at a given supply voltage.
- *Environmental.* Alpha particle radiation coming from the operating environment can cause single event upsets (SEU). Combined with weakened noise margins from manufacturing effects, memory cells are also becoming more susceptible to SEU, impacting their soft error resilience [1]. Noise stemming from variations in the supply voltage and thermal effects can also cause memory faults exhibiting dynamic and random behavior.
- *Aging and Wearout.* Depending on the type of technology used, memory cells can age, reducing their performance, data retention capability, and power consumption. Aging can lead to memory wearout, resulting in permanent faults.

Different memory technologies suffer from various sources of unreliability. Volatile memories such as SRAM and DRAM mostly suffer from manufacturing defects and environmental issues that lead to hard and soft errors, respectively. Endurance is not an issue in SRAM and DRAM. In contrast, different non-volatile memories (NVMs) have their own sources of unreliability. For flash and phase change memory (PCM), wearout is the primary source of unreliability due to limited write endurance. PCMs also suffer from hard and soft errors [2]. Other emerging NVMs such as MRAM and its newer cousin STT-RAM also suffer from hard and soft errors. However, for these devices, wearout is not as great of a reliability threat, because they have large write endurances similar to that of SRAM.

The design of reliable computer systems has a rich history spanning several decades: variants of spatial, temporal, and information redundancy have been exploited to improve reliability. Memory systems also deploy these forms of redundancy to achieve resilience across various layers of system abstraction. Additionally, memory designers have leveraged a variety of other memory-specific techniques.

Here, we provide a sampling of common techniques used for reliable memory design at the architectural level. A significant body of research exists on the design of a reliable memory hierarchy comprising multiple levels of caches and main memory. Fault-tolerant memory designs have often used simple techniques such as adding redundant rows/columns to the memory array [18] or applying memory down-sizing techniques by disabling a faulty row or cache line (block) [20].

Information redundancy via error coding is also commonly used to improve the reliability of memory components. Wide ranges of error detection and correction codes (EDC and ECC, respectively) have been used [7]. Typically, EDCs are simple parity codes, while the most common ECCs use Hamming [8] or Hsiao [9] codes. ECC is proven as an effective mechanism for handling soft errors. For NVMs that have limited write endurance, various wear-leveling approaches have been proposed to mitigate aging and extend memory lifetime.

For many embedded applications, hardware controlled caches do not provide predictable performance and can also be energy inefficient. Consequently, caches are increasingly replaced by or augmented with software-controlled scratchpad memories (SPMs). The design of reliable SPMs has also received great attention recently, including efforts that address the reliability of SPMs for chip-multiprocessors (E-RoC [15] and SPMVisor [16]), or for hybrid memories (FTSPM [17]).

Surprisingly, very little work has attempted to leverage higher-level semantic retention [67] to assist at all levels of unreliability. Indeed, by having a "big-picture" understanding of what data structures/parts-thereof are accessed, how frequently, and in what way during a program phase, and relating these to the fault profiles of the underlying memory subsystems, one could improve the efficiency of (or even eliminate the need for) recovery mechanisms in both hardware and software.

An exhaustive survey of memory resilience is beyond the scope of this paper. However, in the next two sections we present two recent research topics – *resilient caches* and *memory aging* – as vehicles to illustrate opportunities for multi and cross-layer memory resilience. For each case, we briefly explain ongoing efforts and highlight an exemplar study that leverages a multi-layer approach toward improving memory resilience.

3. RESILIENT CACHES

We can categorize resilient SRAM cache design efforts into three main groups. Many of these have the common property of "fault-tolerant voltage-scalable" (FTVS) design, because low voltage operation – while critical for achieving power and energy savings – is the primary driver behind unreliable memories. In general, regardless of whether the fault-tolerant design is done at the cell, circuit, coding, or architecture level, there is a tradeoff in terms of memory capacity and area. This may be due to larger memory cells, spare or redundant cells, error correction logic, or a reduced amount of reliable memory available for use by the application.

3.1 Cell and Circuit-Level Techniques

The root of most SRAM reliability problems is the cell noise margin. At low supply voltages, noise margins are reduced, increasing susceptibility to data corruption caused by environmental factors described earlier. Furthermore, variability in cell noise margins requires a statistical approach to designing a reliable memory array and choice of minimum supply voltage, which must be increased to maintain yield under large variations.

Engineers have designed larger memory cells using more transistors and/or larger transistors to increase mean noise margins and/or reduce margin variability, but these come at the cost of reduced area efficiency and sometimes power. Several of these circuit-level techniques include 8T [3][4], 10T [5], and Schmidt Trigger (ST) [6] SRAM cells.

3.2 Error Coding Techniques

Single error correction double error detection (SECDED) is a widely used coding technique for protecting memory structures against soft errors. When greater error detection is necessary, more complex multi-bit error correction schemes have also been proposed. Double error correction triple error detection (DECDED), two-dimensional ECC (2D-ECC) [10], multiple-bit segmented ECC (MS-ECC) [11], Hi-ECC [12], variable-strength ECC (VS-ECC) [13], and Memory Mapped ECC [14] are some of the more notable schemes. Besides common codes such as Hamming [8] and Hsiao [9], other strong codes such as BCH [12], OLSC [11], and Reed Solomon [7] have also been used to gain strong error detection. However, ECC techniques generally come at high cost due to significant memory storage and logic overheads. Despite this, ECC remains a popular method for memory resilience due to its effectiveness against soft errors, and the lack of involvement from other layers of abstraction.

3.3 Architecture-Level Techniques

Many architecture-level schemes deploy redundancy or capacity downsizing techniques to improve the reliability of cache memories. Earlier works on fault-tolerant cache design use simple techniques by adding redundant rows/columns to the cache [18] or disabling faulty cache block, sets, and/or ways [20]. Similarly, Wilkerson et al. [21] proposed multiple techniques using part of a cache line as redundancy for defective bits for the rest of cache lines in the same set. PADed cache [19] and Agarwal's design [1] program column multiplexer and address decoders to select non-faulty blocks, respectively.

Other efforts, such as In-Cache Replication (ICR) [23] and Multi-Copy Cache (MC^2) [22], use data replication to improve reliability. Schemes such as Replication Cache [24] and ZerehCache [25] use external spare caches. Similarly, variants of fault-grouping and fault remapping have been used to tolerate faulty cache blocks without adding any spare elements, but by using other parts of the cache, such as GRP2 [26], RDC-Cache [27], Abella [28], Archipelago [29], and FFT-Cache [36]. Wilkerson's scheme [21] also could fall under this category.

In all the above schemes, algorithmic and compiler semantic retention could help enhance the efficiency of the proposed mechanisms, by facilitating more accurate remapping, accurate (more limited) replication, and/or more efficient relocation approaches. Some hybrid schemes combine multiple techniques mentioned earlier to minimize the costs of memory protection. Zhou [30] minimizes area overhead through joint optimization of cell size, redundancy, and ECC; and Ndai [31] performs circuit-architecture codesign for memory yield improvement.

More recent architectural schemes for cache resilience address newer challenges for multi and many-core platforms, such as scalability [32][59], variation in fault behaviors [11], non-uniform memory access latency [59], limited shared redundancy [33], low-overhead multi-VDD support [37], and high costs of uniform design [34][35].

3.4 Power/Capacity Scaling

We now turn to our most recent work [37] as an exemplar for cross-layer resilient cache design. Many works in resilient SRAM caches target power reduction by enabling low voltage operation. As described earlier in Section 2, low voltage operation results in higher probability of faulty memory cells, thus requiring some form of fault tolerance. Thus, there is a tradeoff between power (as it depends on supply voltage) and fault tolerance overheads (in terms of area, performance, and power). Despite this, most fault-tolerant voltage-scalable (FTVS) SRAM cache designs emphasize the metric of minimum achievable VDD at fixed yield. This can be misleading when judging the efficacy of such an approach.

Thus, we proposed in [37] a better metric for evaluating FTVS SRAM caches: power versus effective capacity. For example, one can consider an ECC-based cache as either having a power overhead for a given amount of bit storage, or for a given amount of power, fewer bits that are usable to store data. These sorts of tradeoffs are captured appropriately by this metric, and enable more effective cross-layer design.

We realized that employing sophisticated ECC, block-level redundancy or address remapping can achieve very low supply voltages, but not the best design tradeoff in power vs. capacity. When voltage scaling an SRAM array, there is a critical point where the memory becomes virtually useless due to very high bit error rates. Fault tolerance mechanisms allow incrementally lower voltages, but at ever-increasing costs in area, power, performance, and complexity. Thus, it appears that tolerating many errors for low voltage operation can quickly become a fool's errand.

In [37] this realization led us to come up with a simple FTVS SRAM cache architecture for energy savings. The idea is to achieve a better power/capacity tradeoff for a cache by using ultra-lightweight fault tolerance that gracefully degrades cache utility as voltage is lowered. Essentially, an offline or built-in-self-test (BIST) routine identifies blocks that have any faulty bits at each pre-determined VDD level. Using the so-called fault inclusion property [37], we keep a very small fault map (1-2 bits per block) in the tag array, which is not voltage scaled. At any given runtime voltage, the fault map directly controls power gate transistors which disable blocks that are unreliable for further power savings. Meanwhile, the cache controller prohibits valid data from being placed in a faulty block. From the software's perspective, the cache capacity is reduced at low voltage, causing more misses, but otherwise the cache operates correctly with good power savings. However, the yield could be affected since each set requires at least one non-faulty block at all runtime voltages.

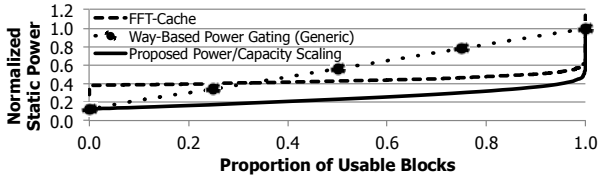


Figure 2. Static power vs. effective capacity for three different SRAM cache resizing approaches [37].

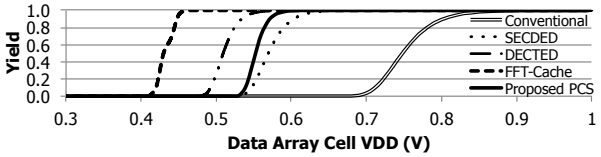


Figure 3. Yield vs. VDD for several different fault-tolerant voltage-scalable (FTVS) SRAM cache approaches [37].

Figure 2 illustrates the benefit of our power/capacity scaling approach compared with power gating and FFT-Cache [36] (one of our recent FTVS works), for trading off power and capacity.

This is despite the inability of the proposed power/capacity scaling method to achieve the lowest voltage at any yield target (Figure 3), motivating further studies in this direction.

We proposed in [37] two policy variants of power/capacity scaling: static (SPCS) and dynamic (DPCS). SPCS allows the system software or cache controller to choose the optimal cache voltage at boot time, based on knowledge of faulty blocks gained through BIST, to achieve a minimum of 99% fault-free blocks. While SPCS is simple and can greatly reduce the voltage guardband, it ignores the opportunity for even better energy savings through cross-layer hardware/software optimization.

DPCS allows the system to adapt the cache VDD at runtime in response to varying workload behaviors. In [37] we had the cache controller adapt the voltage in response to changing miss rates and an estimate of the miss penalty. When many misses were encountered at low voltage, the controller raises VDD to make more blocks available for use and thereby reduces capacity and conflict misses. When few misses are encountered, the controller reduces VDD to opportunistically save power.

Higher level semantics can mitigate the effect of the reduced cache size on performance (e.g., by simply increasing power – and hardware reliability - in phases of execution where the cache is fully utilized) or more interestingly, by using the higher-level information to adapt the organization/utilization of the data so as to minimize misses given the faulty-cache configuration. More sophisticated cross-layer policies are part of our ongoing work. With knowledge of the power/capacity scaling mechanism and particular cache operating points, software could be optimized at compile-time or runtime to improve energy efficiency with minimal performance degradation.

4. MEMORY AGING AND WEAROUT

We now review sample efforts that cope with wearout in memories and their limited lifetime at different levels of abstraction. As with resilient caches, higher-level semantic retention can help, by using information about how different program and algorithm-level structures are utilized (frequency of access, of reads of writes, their mappings at bank or cache level, etc. in different phases of program execution) to both increase efficiency of execution in the presence of faults, and to alleviate the expense of recovery mechanisms in software or hardware. We also illustrate how program characteristics can be exposed to the hardware in order to mitigate wearout effects, using the example of large GPGPU register files.

4.1 Wearout Mechanisms and Their Effects

Wearout mechanisms are different depending on the type of the memory family. While electron tunneling degrades the oxide layer in flash memory cells, SRAM is threatened by negative-bias temperature instability (NBTI) which weakens the drive current of PMOS devices. Furthermore, wearout effects are also different for each memory type. Wearout in NVMs limits the number of reliable writes. In SRAM, it decreases the stability of cells, especially for the read operation. Although wearout in NVMs is typically irreversible, SRAM wearout is partially recoverable.

4.2 Improving NVM Write Endurance

Traditional memory management techniques are write-variation oblivious and therefore cause parts of the memory to reach its maximum write count much earlier than the rest. Thus, most approaches for enhancing write endurance of NVMs are based on two ideas: (1) uniformly distributing writes over the whole memory space, and (2) reducing the number of write operations.

4.2.1 Flash as Main Memory

Approaches for wear-leveling in flash memories fall into two categories. First, dynamic wear leveling (DWL) techniques look at all of the available blocks that are free and select the one with the lowest erase count for next write. However, they do not move cold data afterwards [38]. Second, static wear leveling (SWL) techniques try to prevent cold data from staying at any block for a long period of time. If the difference between two blocks' erase counts is too large, SWL starts erasing young blocks by moving cold data away from them [39].

4.2.2 PCM as Main Memory

Architectural Level Solutions: Flip-N-Write [40] is a micro-architectural technique that performs a read-before-write to decide whether to write the original data or its flipped version depending on which causes fewer bit flips. This is transparent to the rest of the system and the memory device takes care of inverting data whenever required. The authors in [41] consider manufacturing variation, which causes the programming current to be adjusted, based on the most difficult-to-reset cell. Instead of sacrificing lifetime of other cells, they use a lower programming current through Fine-Grained Current Regulation, allowing difficult-to-reset cells to be recovered by error correcting pointers (ECP).

OS Level Solutions: Dhiman et al. propose PDRAM [42] for hybrid PCM and DRAM memories. The operating system's page manager uses the page-level access frequency of PCM pages, tracked by hardware, in order to perform wear leveling. The OS also tries to swap hot pages from PRAM to DRAM. By changing the memory controller, the TLBs, and the operating system, the authors of [43] dynamically form clean pages out of pages with faulty bits. This enables continued operation through graceful degradation when cells fail.

Application Level Solutions: A recent work by Sampson et al. [44] offers a new perspective for improving PCM lifetime. Through annotations, the application developer can identify some program variables as candidate for approximate storage. Hardware exploits this by reducing number of programming pulses for that part of physical memory that holds this data. In addition, even failed cells are used for storing approximate data.

4.2.3 PCM as On-Chip SPM

HaVOC [66] uses a combination of programmer annotations and a data volatility metric to simultaneously save energy while increasing the lifetime of NVMs. The volatility metric measures write frequency of a piece of data over its accumulated lifetime. Variable annotations are used to pass this metric to the run-time system, allowing the SPM manager to prioritize mapping of data with higher write frequency to be put in on-chip SPM. Thus by reducing the write operations to NVM, not only is the energy consumption of SPM reduced, but also its life-time is increased.

4.2.4 ReRAM as On-Chip Last-Level Cache

[45] proposes inter/intra-set write variation-aware cache policy (i^2WAP) for ReRAM caches. Using address remapping, it uniformly distributes cache writes between all of the cache sets. This solves the problem of inter-set variation but within a set, hot cache lines are accessed more frequently because of locality. To solve this, i^2WAP slightly modifies the Least Recently Used (LRU) replacement policy by intelligently writing back hot data at some timestamp and invalidating the corresponding line. The invalidated line would be a candidate for the next replacement, possibly for cold data.

4.3 Mitigating SRAM Aging

4.3.1 Architectural Level Solutions for SRAM Caches

[46] proposes Dynamic Indexing for SRAM caches. The authors observe that in a partitioned cache architecture, some of the partitions are idle during most of the application execution time, while some others are accessed more. They exploit this behavior by putting idle partitions in drowsy mode (i.e., drooped VDD). This slows down the wearout of SRAM cells in those partitions. Also the cache indexing function is changed over time in order to uniformly distribute the idleness over all of the partitions.

4.3.2 Software Level Solution for SRAM SPMs

[47] presents a library of C-functions for wearout-aware data allocation on physically-banked SPMs. For data allocation, *SPM_malloc* calls the SPM controller which is aware of the current wearout status of each bank. This controller distributes allocation requests over the SPM banks in such a way that all banks could spend the same amount of time in drowsy mode.

4.4 Register File Aging Case Study: ARGO

Extreme multithreading with fast thread switching in GPGPUs is supported by large register files (RFs) that are much larger than on-chip caches holding the execution state of each thread. To protect these register files against NBTI, ARGO [48] exposes program characteristics to the hardware in order to design a low-overhead stress distributer. In ARGO's flow (Figure 4), the OpenCL compiler embeds some metadata in binary code, including number of required registers for that kernel and its maximum amount of required memory. Based on this information, the host CPU at runtime decides on how many threads to assign to each workgroup. Depending on the kernel requirements and resource limitations not all of the available register file space can be used. On average, 46% underutilization is observed for execution of 15 common general purpose kernels. In such a flow, the compiler helps the underlying hardware by letting it know how much of the register space is required by the software. The RF allocator then power-gates unused parts of the register file, thereby not only saving leakage power, but more importantly ameliorating aging by putting that part in NBTI recovery mode. Furthermore the RF allocator employs a virtual sensing approach to estimate the aging profile of different RF banks in a relative manner. Based on that, and without any need of having on-chip NBTI sensors, it circulates the allocated space in the entire physical space of RF over time to enhance the RF lifetime.

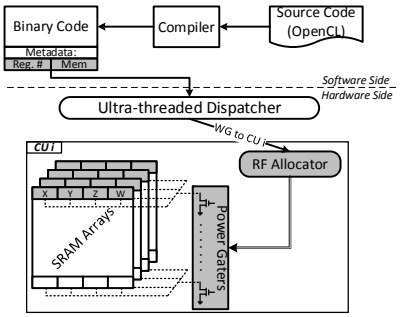


Figure 4. ARGO Overview.

5. VARIABILITY EXPEDITION

Variability in computer systems can stem from semiconductor manufacturing, ambient operating conditions, wearout over time, and differing vendors. The NSF Variability Expedition (Figure 5) [49] seeks to build opportunistic computing systems where hardware variations are monitored and exposed to software layers (instead of being hidden behind pessimistic margins) enabling adaptations. The work has spanned circuit-level monitoring and test (e.g., [50], [51], [60]), variability emulation ([52], [53]), runtime for embedded systems (e.g., [54], [55]), GPUs (e.g., [56], [48]), processors (e.g., [56], [58]), memories (e.g., [55], [64], [59]) and storage (e.g., [61], [62]). In the following, we briefly describe some of the research on memory variability done under the Variability Expedition.

DRAMs were observed to have over 20% read/write power variation [63] which was leveraged in [64] by dynamically adapting virtual to physical address mapping in the Linux operating system. The approach preferentially allocates frequently accessed data on to lower power memory DIMMs.

SRAM arrays are known to have large variations which limit their minimum operating voltage and hence power. [15] achieves reliability through redundancy by optimizing RAID-like policies tuned for on-chip distributed scratchpad memories at lower power cost than ECC with voltage overscaling. Extending this, [55] allows programmers to partition their application's address space (through annotations) into virtual address regions and create mapping policies for each region depending on their requirements (fault tolerance, power, etc). In the cache context, FFT-Cache [36] uses sophisticated fault tolerance schemes in cache organization to achieve a lower operating voltage, while [37] described earlier does this using simple fault tolerance mechanisms for lower overheads.

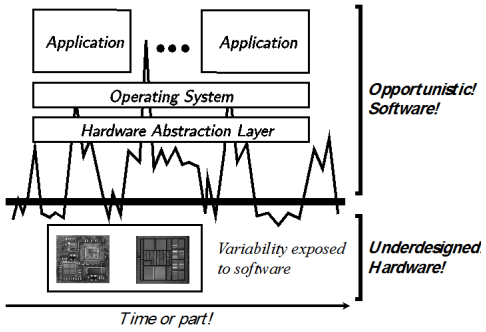


Figure 5. The Underdesigned and Opportunistic Computing vision of the NSF Variability Expedition [49].

Measurements show systematic variation in program latency within and across multi-level flash devices [65]. [62] extends conventional flash translation layers to schedule flash program operations on pages based on operations performance requirements and specific pages' performance characteristics. Based on the observation that, for multi-level cell flash, whenever a cell error occurs, with high probability only one bit in the cell has error, [61] proposed an error correcting code based on generalized tensor products.

The increasing fraction of memory real estate coupled with emerging memory technologies with varying variability mechanisms make architecture and software-level handling of memory variations an integral part of the Variability Expedition.

6. SUMMARY AND CONCLUSIONS

In this paper, we highlighted efforts and opportunities for achieving memory resiliency both within and across multiple layers of the abstraction stack. To enable cross-layer memory resilience, it is important to understand the abstractions of memories, manifestations of memory errors and memory vulnerability at multiple levels. Our paper gave a sampling of these memory issues within the context of complex SoC designs. We also used two exemplars (resilient caches and memory aging) to illustrate multi-layer strategies for enhancing resilience.

While traditional memory resilience efforts have focused primarily on hardware, it is increasingly important to develop software-enabled mechanisms for managing memory resilience. Moving forward, we should see efforts that synergistically combine hardware and software approaches to overcome adverse effects of memory failures, and also which opportunistically exploit application semantics for achieving more efficient designs, particularly in the context of applications that tolerate some level of quality degradation (e.g., approximate computing). System designers will need abstractions, tools, and methods to enable effective exploration of the memory resiliency design space.

7. ACKNOWLEDGMENTS

This work was partially supported by NSF Variability Expedition Grant Numbers CCF-1029783 and CCF-1029030.

8. REFERENCES

[1] A. Agarwal et al. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE TVLSI*, 2005.

[2] D. H. Yoon et al. FREE-p: Protecting non-volatile memory against both hard and soft errors. *Proc. HPCA*, 2011.

[3] Y. Morita et al. An area-conscious low voltage-oriented 8T-SRAM design under DVS environment. *Proc. Symp. on VLSI Circuits*, 2007.

[4] N. Verma and A. Chandrakasan. A 256 Kb 65 nm 8T subthreshold SRAM employing sense-amplifier redundancy. *IEEE JSSC,* 2008.

[5] B. Calhoun and A. Chandrakasan. A 256 Kb sub-threshold SRAM in 65nm CMOS. *Proc. ISSCC*, 2006.

[6] J. P. Kulkarni et al. A 160 mV, fully differential, robust schmitt trigger based sub-threshold SRAM. *Proc. ISLPED*, 2007.

[7] S. Lin and D. J. Costello. Error control coding, second edition. *Prentice-Hall, Inc.*, 2004.

[8] R. W. Hamming. Error correcting and error detecting codes. *Bell System Tech. Jour.*, 1950.

[9] M. Y. Hsiao. A class of optimal minimum odd-weight-column SECDED codes. *IBM J. Research and Devel.*, 1970.

[10] J. Kim et al. Multi-bit error tolerant caches using two-dimensional error coding. *Proc. MICRO*, 2007.

[11] Z. Chishti et al. Improving cache lifetime reliability at ultra-low voltages. *Proc. MICRO*, 2009.

[12] C. Wilkerson et al. Reducing cache power with low-cost, multi-bit error-correcting codes. *Proc. ISCA*, 2010.

[13] A. Alameldeen et al. Energy-efficient cache design using variable-strength error correcting codes. *Proc. ISCA*, 2011.

[14] D. H. Yoon and M. Erez. Memory mapped ECC: low-cost error protection for last level caches. *Proc. ISCA*, 2009.

[15] L. Bathen and N. Dutt. E-RoC: embedded raids-on-chip for low power distributed dynamically managed reliable memories. *Proc. DATE*, 2011.

[16] L. Bathen et al. SPMVisor: dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. *Proc. CODES+ISSS*, 2011.

[17] A.M.H. Monazzah et al. FTSPM: a fault-tolerant scratchpad memory. *Proc. DSN*, 2013.

[18] S. Schuster. Multiple word/bit line redundancy for semiconductor memories. *IEEE JSCC*, 1978.

[19] P. Shirvani and E. McCluskey. PADded cache: a new fault tolerance technique for cache memories. *Proc. VTS*, 1999.

[20] S. Ozdemir et al. Yield-aware cache architectures. *Proc. MICRO*, 2006.

[21] C. Wilkerson et al. Trading off cache capacity for reliability to enable low voltage operation. *Proc. ISCA*, 2008.

[22] A. Chakraborty et al. E < MC^2: less energy through multi-copy cache. *Proc. CASES*, 2010.

[23] W. Zhang et al. ICR: in-cache replication for enhancing data cache reliability. *Proc. DSN*, 2003.

[24] W. Zhang. Replication cache: a small fully associative cache to improve data cache reliability. *IEEE TC*, 2005.

[25] A. Ansari et al. ZerehCache: armoring cache architectures in high defect density technologies. *Proc. MICRO*, 2009.

[26] D. Roberts et al. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. *Proc. DSD*, 2007.

[27] A. Sasan et al. A fault tolerant cache architecture for sub 500mV operation: resizable data composer cache (RDC-cache). *Proc. CASES*, 2009.

[28] J. Abella et al. Low VCC-min fault-tolerant cache with highly predictable performance. *Proc. MICRO,* 2010.

[29] A. Ansari et al. Archipelago: a polymorphic cache design for enabling robust near-threshold operation. *Proc. HPCA*, 2011.

[30] S.T. Zhou et al. Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC. *Proc. ICCD*, 2010.

[31] P. Ndai et al. A scalable circuit-architecture co-design to improve memory yield for high-performance processors. *IEEE TVLSI*, 2010.

[32] A. BanaiyanMofrad et al. A novel NoC-based design for fault-tolerance of last-level caches in CMPs. *Proc. CODES+ISSS*, 2012.

[33] A. BanaiyanMofrad et al. Modeling and analysis of fault-tolerant distributed memories for networks-on-chip. *Proc. DATE*, 2013.

[34] S. Paul et al. Reliability-driven ECC allocation for multiple bit error resilience in processor cache. *IEEE TC*, 2011.

[35] P. Ampadu et al. Breaking the energy barrier in fault-tolerant caches for multicore systems. *Proc. DATE*, 2013.

[36] A. BanaiyanMofrad et al. FFT-Cache: a flexible fault-tolerant cache architecture for ultra low voltage operation. *Proc. CASES*, 2011.

[37] M. Gottscho et al. Power/capacity scaling: energy savings with simple fault-tolerant caches. *Proc. DAC*, 2014.

[38] L. P. Chang. On efficient wear-leveling for large-scale flash-memory storage systems. *Proc. SAC*, 2007.

[39] Y. H. Chang et al. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. *Proc. DAC*, 2007.

[40] S. Cho and H. Lee. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy, and endurance. *Proc. MICRO*, 2009.

[41] L. Jiang et al. Enhancing phase change memory lifetime through fine-grained current regulation and voltage upscaling. *Proc. ISLPED*, 2011.

[42] G. Dhiman et al. PDRAM: a hybrid PRAM and DRAM main memory system. *Proc. DAC*, 2009.

[43] E. Ipek et al. Dynamically replicated memory: building resilient systems from unreliable nanoscale memories. *ASPLOS*, 2010.

[44] A. Sampson et al. Approximate storage in solid-state memories. *Proc. MICRO*, 2013.

[45] J. Wang et al. i^2WAP: improving non-volatile cache lifetime by reducing inter- and intra-set write variations. *Proc. HPCA*, 2013.

[46] A. Calimera et al. Dynamic indexing: leakage/aging co-optimization for caches. *IEEE TCAD*, 2014.

[47] D. Papagiannopoulou et al. Flexible data allocation for scratch-pad memories to reduce NBTI effects. *Proc. ISQED*, 2013.

[48] M. Namaki-Shoushtari et al. ARGO: aging-aware GPGPU register file allocation. *Proc. CODES+ISSS*, 2013.

[49] P. Gupta et al. Underdesigned and opportunistic computing in presence of hardware variability. *IEEE TCAD*, 2012.

[50] L. Lai and P. Gupta. Accurate and inexpensive performance monitoring for variability-aware systems. *Proc. ASP-DAC,* 2014.

[51] P. Singh et al. Dynamic NBTI management using a 45nm multi-degradation sensor. *Proc. CICC*, 2010.

[52] H. Cho et al. Quantitative evaluation of soft error injection techniques for robust system design. *Proc. DAC,* 2013.

[53] L. Wanner et al. VarEMU: an emulation testbed for variability-aware software. *Proc. CODES+ISSS*, 2013.

[54] L. Wanner et al. Hardware variability-aware duty cycling for embedded sensors. *IEEE TVLSI*, 2012.

[55] L. Bathen et al. VaMV: variability-aware memory virtualization. *Proc. DATE*, 2012.

[56] A. Rahimi et al. Aging-aware compiler-directed VLIW assignment for GPGPU architectures. *Proc. DAC,* 2013.

[57] M. Fojtik et al. Bubble razor: an architecture-independent approach to timing-error detection and correction. *Proc. ISSCC*, 2012.

[58] J. Sartori et al. Stochastic computing: embracing errors in architecture and design of processors and applications. *Proc. CASES*, 2011.

[59] A. BanaiyanMofrad et al. REMEDIATE: a scalable fault-tolerant architecture for low-power NUCA cache in tiled CMPs. *Proc. IGCC*, 2013.

[60] M. Sauer et al. Early-life failure detection using SAT-based ATPG. *Proc. ITC*, 2013.

[61] R. Gabrys et al. Tackling intracell variability in TLC dlash through tensor product codes. *Proc. ISIT*, 2012.

[62] L. M. Grupp et al. The harey tortoise: managing heterogeneous write performance in SSDs. *Proc. USENIX Ann. Tech. Conf.*, 2013.

[63] M. Gottscho et al. Power variability in contemporary DRAMs. *IEEE ESL*, 2012.

[64] L. Bathen et al. ViPZonE: OS-level memory variability-aware physical address zoning for energy savings. *Proc. CODES+ISSS*, 2012.

[65] L. Grupp et al. Characterizing flash memory: anomalies, observations, and applications. *Proc. MICRO*, 2009.

[66] L. Bathen and N. Dutt. HaVOC: a hybrid memory-aware virtualization layer for on-chip distributed scratchpad and nonvolatile memories. *Proc. DAC*, 2012.

[67] S. Novack et al. A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation. *Proc. LCPC*, 1995

[68] V. Chandra. Monitoring reliability in embedded processors – A multi-layer view. *Proc. DAC*, 2014.

[69] J. Henkel et al. Multi-layer dependability: from microarchitecture to application level. *Proc. DAC*, 2014.

[70] V. B. Kleeberger et al. workload- and instruction-aware timing analysis – the missing link between technology and system-level resilience. *Proc. DAC*, 2014