VaMV: Variability-aware Memory Virtualization

Luis Angel D. Bathen*, Nikil D. Dutt*, Alex Nicolau*, and Puneet Gupta †
*School of Information and Computer Science, University of California, Irvine, {lbathen,dutt,nicolau}@ics.uci.edu
†Department of Electrical Engineering, University of California, Los Angeles, puneet@ee.ucla.edu

***Abstract*—Power consumption variability of both on-chip SRAMs and off-chip DRAMs is expected to continue to increase over the next decades. We opportunistically exploit this variability through a *novel* Variability-aware Memory Virtualization (*VaMV*) layer that allows programmers to partition their application's address space (through annotations) into virtual address regions and create mapping *policies* for each region. Each *policy* has different requirements (e.g., power, fault-tolerance) and is exploited by our dynamic memory management module (*VaMVisor*), which adapts to the underlying hardware, prioritizes the memory resources according to their characteristics (e.g., power consumption), and selectively maps data to the best-fitting memory resource (e.g., high-utilization data to low-power memory space). Our experimental results on embedded benchmarks show that *VaMV* is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 34% by exploiting: 1) SRAM voltage scaling, 2) DRAM power variability, and 3) Efficient dynamic *policy*-driven variability-aware memory allocation.**

I. Introduction

Hardware variability (e.g., chip-to-chip power consumption variation) is quickly becoming a major concern for the design community. ITRS predicts that over the next decade performance variability will increase from 48% to 66% and total power consumption variability will increase by up to 100% [1]–[3]. There are many factors that influence the variation within and across devices (e.g., temperature, voltage, wear-out, etc.). The memory hierarchy is also affected by variability [4]. Moreover, variability plays a major role not only in system performance and power consumption but also in production costs, since high degrees of variability might cause a device to be discarded [5]. In order to cope with this expected increase in variability, designers must build adaptable and tunable software/hardware. This paper presents *VaMV*: a Variability-aware Memory Virtualization approach that allows programmers to exploit such on- and off-chip memory variability to reduce power consumption through memory virtualization, while abstracting the underlying hardware variability from the programmers. *VaMV* allows programmers to partition their application's virtual address space into regions and create mapping *policies* for each region. Each *policy* can be designed to meet different requirements (e.g., power, performance, fault-tolerance). These user-defined and programmer-driven policies are then exploited by our dynamic memory management module (*VaMVisor*), which adapts to the underlying hardware, prioritizes the memory resources according to their characteristics (e.g., power consumption), and selectively maps program data to the best-fitting memory resource. (e.g., highly-utilized data to low-power memory space). The **novel contributions** of our work are that we:

- Exploit and co-optimize both on-chip and off-chip memory variability
- Introduce the notion of variability-aware address space partitioning (through programmer-driven source-code annotations)
- Present a dynamic variability-aware memory virtualization (*VaMV*) layer that transparently maps program data to physical memories while exploiting variability through the use of annotations and dynamic memory management (*VaMVisor*)

Our experimental results on a set of embedded benchmarks show that VaMV is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 34% by exploiting: 1) Selective voltage scaling to reduce SRAM power consumption, 2) DRAM power variability, and 3) Efficient dynamic *policy*-driven variability-aware memory allocation.

978-3-9810801-8-6/DATE12/©2012 EDAA

II. Related Work

Various efforts have shown that exploiting variability in off-the-shelf hardware may lead to promising results. Hanson et al. [6] measured the power consumption across *identical* Intel M processors and found between 3% and 10% variation and up to 2x active power variation across various DRAMs. Wanner et al. [3] found over 5x sleep power variation across various Cortex M3 processors. Sartori et al. [2] looked at frequency variation across processing cores. Pant et al. [7] proposed hardware signatures to adapt the software stack to deal with hardware. Gottscho et al. [4] observed up to 17.73% power variation across *different* vendor 1GB DRAMs and up to 16.40% power variation across *same* vendor DRAMs.

Aggressively voltage scaling on-chip SRAMs has been shown to reduce power consumption at the cost of increasing process variations. Chakraborty et al. [8] propose the idea of exploiting error maps to correct data-cache faulty cells. Mutyam et al. [9] proposed the concept of block rearrangement to minimize data-cache performance loss. Liang et al. [10] proposed replacing 6T SRAM with 3T1D DRAM for caches to address physical device variation. Meng et al. [11] proposed way prioritization to minimizes cache leakage to address within-die leakage variation. Li [12] proposed repairing only important bits to address process variations. Kurdahi et al. [13] proposed an application level solution to handle process variations in the memory subsystem.

To the best of our knowledge, we are the first to propose the idea of transparently exploiting on-chip and off-chip memory variability at the programmer level through annotations to reduce power consumption. In particular, *VaMV* differs from [2], [3], [6] because we focus on memory variability, however, *VaMV* could be complemented by other variability-aware approaches (e.g., CPU power consumption). *VaMV* is different from [8], [13], [14] in that we selectively voltage scale on-chip memories and take advantage of the variation (power, performance, error rates) opportunistically at the system level.

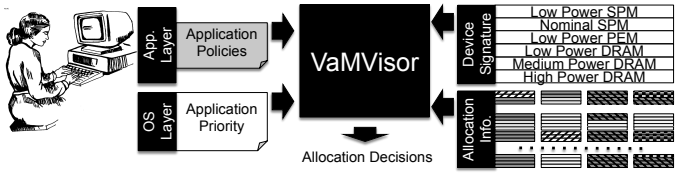

Fig. 1. VaMV: Variability-aware Memory Virtualization.

III. Variability-Aware Memory Virtualization

A. VaMV Overview

The goal of our Variability-aware Memory Virtualization (*VaMV*) layer is to allow programmers to opportunistically exploit variability across various levels of the memory hierarchy through annotations

in order to reduce power consumption. Figure 1 shows a high-level view of *VaMV*, which has five main components: 1) The application's programmer-specified source-code annotated data mapping *policies* (Sec. III-D), 2) The application's *priority*, which is used to prioritize use of the memory space among the various applications, 3) The device's signature (Sec. III-E), which is based on the memory subsystem's characteristics (e.g., power consumption). 4) Current memory allocation information used to derive available memory resources and memory re-mapping opportunities (Sec. III-F). 5) The *VaMVisor*, which enforces the mapping *policies* at run-time while using the application's priority, the device's signature, and the allocation information to efficiently allocate the memory space. For more details please refer to our technical report [15].

B. Target Platform and Assumptions

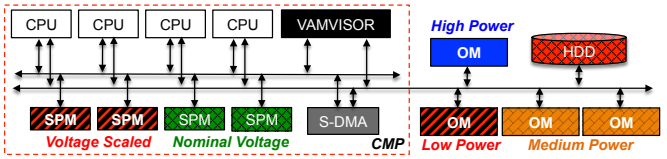

Fig. 2. Chip-Multiprocessor (CMP) with Distributed Memories.

Figure 2 shows our target CMP platform, which consists of a number of: 1) processing cores, 2) on-chip *distributed* ScratchPad Memories (SPMs) ([16], [17]), 3) off-chip DRAM memories (OMs), 4) a secure DMA (S-DMA) engine to protect the memory space, 5) a hard drive (HDD), and 6) the *VaMVisor*.

We make the following assumptions: 1) We have access to hardware signatures representing power consumption variability for on- and off-chip memories [4], 2) We can selectively voltage scale our on-chip memories, and consequently a subset of on-chip memories may have lower power consumption, higher access latencies and higher error rates than others [18], and 3) Programmers are able to partition their application's address space into regions with different requirements (power, performance, etc.) through source code annotations.

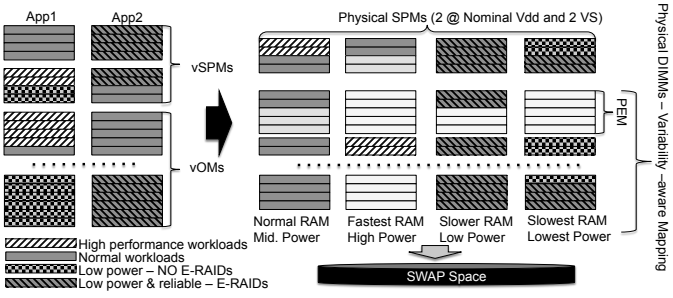

Fig. 3. Virtualization Layer Overview.

C. Virtual SPM and Virtual Off-chip Memory

To virtualize the memory space we define the notions of virtual SPMs (*vSPMs* [19]) and virtual Off-chip Memory (*vOM*) that are mapped to physical on-chip SRAMs and off-chip DRAMs with varying power and performance characteristics. vSPMs are realized by locking part of off-chip memory space, defined as Protected Evict Memory (*PEM*), in order to *extend* the available on-chip memory space. We take advantage of DRAM power variability by mapping the *PEM* space to the DRAM with lowest power consumption. vOMs follow the same notion as virtual memory, however, unlike traditional memory virtualization schemes, our memory virtualization layer allows programmers to partition their virtual memory into regions (within vSPMs/vOMs) and define *policies* for each region requiring different guarantees (low power, performance, fault-tolerance). Figure 3 shows an overview of the memory virtualization layer consisting of two applications (App1, App2) where the programmer has annotated and partitioned the virtual memory to meet varying power/performance needs through vSPMs/vOMs as follows: 1) High-performance/low-latency address space (back-slashed blocks), 2) Normal Vdd address space (gray blocks), 3) Low-power address space (checkered blocks), and 4) Low-power & fault-tolerant memory space (forward-slashed blocks).

D. Programmer-Driven Policy Generation

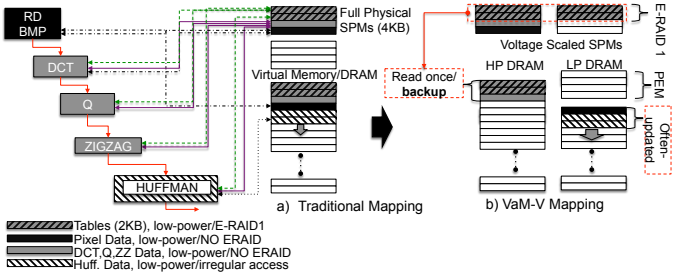

Fig. 4. Partitioning the Application's Memory Space

In order to efficiently exploit the available memory variability, programmers need to annotate their applications with memory allocation hints in the form of *policies*. The application's address space is partitioned into into virtual regions, which are then associated with a mapping *policy* that dictates how to map the data into physical address space and the type of guarantee needed (power, performance, fault-tolerance). Figure 4 shows a *sample* address space partitioning for JPEG [20], where the programmer has identified: 1) Read-only and highly utilized data, i.e., look up tables (back-slashed blocks), 2) A temporary buffer for inter-task communication (gray block), 3) Read-only pixel data (black blocks), and 4) Irregularly access data (forward-slashed blocks).

Figure 4 (a) shows a traditional mapping of these data blocks, where variability is *not* taken into account. Figure 4 (b) shows the result of our *VaMV* virtualization layer mapping that exploits: 1) Data mapping *policies* customized by the programmer and used to make dynamic memory allocation decisions. 2) On-chip memory voltage scaling (using E-RAIDs [16] to deal with process variation), and 3) DRAM variability. For the sake of illustration, *VaMV* maps commonly used read-only data to voltage scaled SRAM protected by an E-RAID 1 level, pixel data to voltage scaled SRAM (NO ERAID), and irregular commonly used data to low power DRAM. A programmer can annotate the application with expected mapping *policies* with low-power (*LP*) memory space in mind. *VaMV* then takes these policies and tries to opportunistically enforce them (best effort), regardless of how the *LP* memory space is implemented by the hardware layer. For instance, If there is no noticeable DRAM power variability, then *VaMV* will not prioritize DRAMs and follow a more traditional memory management scheme (e.g., malloc).

E. Hardware Device Signature

The hardware variability signature of the device (e.g., DRAMs and on-chip SRAMs) allows the *VaMVisor* to opportunistically exploit the available memory power variability. Traditional memory allocation schemes treat on-chip SRAM as precious memory space and off-chip DRAM as second-tier memory space, while making no other

TABLE I
MEMORY RANKING

Rank	Description
R1	Voltage Scaled SRAM: process variations, low power, & increased access latency
R2	Nominal Vdd On-Chip Memories: higher power consumption
R3	Low-power DRAM: characterized at manufacture-time or at run-time
R4	Mid-power DRAM
R5	High-power DRAM

difference. A sample allocation policy would then map commonly used data to SRAM (what it can fit), and all other data to DRAM. In order to exploit the device's signature, *VaMV* further breaks down the memory space and ranks it based on the characteristics of the memories. Table I shows a sample ranking of the various memories, from most precious resource ($R1$) to least precious ($R5$). First, we voltage scale some SRAM memories, as a result we can have two types of on-chip memories: 1) Normal SRAM and 2) Low-power SRAM with their side-effects (e.g., process variations, higher access latency, etc.). Similarly, DRAMs can be further ranked based on power consumption, exploiting the inherent chip-to-chip variability [4]. The *VaMVisor* then uses this ranking information (device signature) to prioritize the data/virtual space to physical space mapping in order to save power.

F. VaMVisor: Policy-Driven Dynamic Allocation

The *policy* associated with each block will determine how the block is mapped (e.g., low-power memory space). The application's priority is used by the run-time environment to decide how to efficiently use the memory space (e.g., give higher priority to applications with real-time requirements). Finally, the current memory mapping information is used to determine whether it makes sense to re-map data blocks.

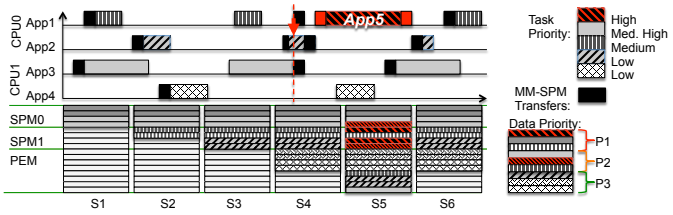

Fig. 5. Multi-tasking Policy-driven Variability-aware Allocation.

The *VaMVisor* can be implemented as a *software* module at the OS/hypervisor level (flexible, no extra hardware needed) or a *hardware* module that can be embedded in today's CMPs as an augmented arbiter/MMU [16], [21] (lower performance overheads and reduced power consumption). In this paper we focus on the HW-assisted *VaMVisor* with a fixed-size block-based allocation unit, which takes as input the triplet (*AppPriority*, *BlkPriority*, *MallocPolicy*), to determine where to allocate a given block. Figure 5 shows the execution of a set of applications (*App1-4*) on two CPUs (*CPU0-1*) with 2x4KB SPMs and the status of the memories as vSPMs are created and blocks are allocated (States *S1-S6*). On arrival of the first application (*App3*), the vSPM is created and the *VaMVisor* maps *App3*'s blocks to *SPM0*, and the process continues up until *S3*. When *App4* arrives, the *VaMVisor* evaluates *App4*'s block priorities (*BlkPriority*) and decides to map them to *dedicated* off-chip memory space (*PEM*). When *App5* (forward-slashed block) executes, rather than evicting all of *App1 and App2's* contents from SPM (as in traditional approaches [22]), the *VaMVisor* looks at the *priorities* of the various blocks, evicts *some* of the lower priority blocks from SPM space (*App1-3*), and allocates the space to *App5*'s blocks (*S5*). The *VaMVisor* partitions the off-chip memory (OM) space and ranks it to exploit the variability present in DRAM. An example is to exploit a utilization-based priority policy, where two applications request virtual off-chip memory (vOM) space with the same priority (but utilization is higher for one than the other); the *VaMVisor* would then try to map the vOM to the physical OM with the lowest power consumption. If it cannot serve both requests, then the application with the highest utilization would be given priority. Of course, the *VaMVisor* would try to map the other application's data to the next low-power OM. The goal is to exploit the application's priority *and* their data blocks' priorities to efficiently manage the memory space.

TABLE II
SAMPLE USER ANNOTATIONS AND POLICIES

Data Type	Description
T1	Look-up tables (e.g., quantization variables, ziz-zag indices)
T2	Commonly used data (e.g., inter-task communication buffers, variables)
T3	Non-Critical Data (e.g., pixels)
T4	All other
Policy	**Desctiption**
P	*physical SPM (pSPM) ← {T1,T2,T3}, random-malloc OM ← T4*
VE1	*virtual SPM (vSPM)/E-RAID1/0.5V ← {T1,T2,T3}, High Power (HP) OM ← T4*
VE2	*vSPM/E-RAID1/0.5V ← {T1,T2,T3}, Low Power (LP) vOM ← T4*
VE3	*vSPM/E-RAID1/0.5V ← {T1}, vSPM/NO-ERAID/0.5V ← {T2,T3}, HP OM ← T4*
VE4	*vSPM/E-RAID1/0.5V ← {T1}, vSPM/NO-ERAID/0.5V ← {T2,T3}, LP vOM ← T4*
M1	*8×vSPMs/LP PEM ← {T1,T2,T3}, HP OM ← T4*
M2	*8×vSPMs: E-RAID1 ← {T1}, 3×NO-ERAID & 4×LP PEM ← {T2,T3}; HP OM ← T4*
M3	*8×vSPMs: E-RAID1 ← {T1}, 3×NO-ERAID & 4×LP PEM ← {T2,T3}; LP vOM ← T4*

IV. EXPERIMENTAL EVALUATION

A. Experimental Evaluation Goals and Setup

Our goal is twofold; First, we want to show the benefits of exploiting DRAM power variability and voltage scaling *combined* to reduce power consumption. Second, we want to show how a programmer can exploit hardware-variability through custom *policies*. Our simulation environment is implemented in SystemC with SimpleScalar [23] and CACTI [24] support. We can simulate a bus-based Chip-Multiprocessor with *distributed* on-chip SPMs, 4x1GB DRAMs from the same vendor & specs [4] (referred to as OMs), and our *VaMVisor* as shown in Figure 2. We assume 65nm process technology for our SPMs. We cross-compiled a set of applications from Mediabench II [20] and analyzed them to obtain SPM mappable data sets. We simulate a light-weight hypervisor, where each CPU can run 1-4 OSes and 1-8 applications. Table II shows a set of sample user annotations (mapping policies). The base-line policy (e.g., memory space allocation across random DRAM) starts with *P*, the (vSPMs/E-RAID) policies start with VE, and the policies that support multi-tasking start with M. The various data sets are represented by T. We now show how to exploit DRAM variability, combine it with SRAM voltage scaling, and finally show the efficacy of VaMV in exploiting variability across on-chip and off-chip memories.

B. Exploiting DRAM Variability

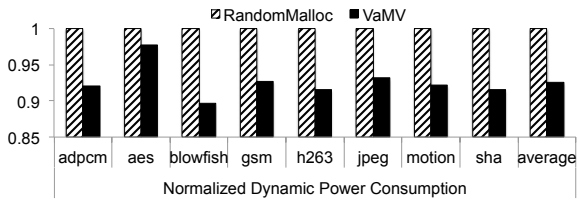

Fig. 6. Exploiting DRAM Variability.

To show the benefits of exploiting DRAM variability, we simulated a single application running on the system with 4x1GB DRAMs exhibiting variability from the same vendor/specs [4] and no data cache/SPMs. All data was directly accessed from DRAM. We then compared our variability-aware memory allocation approach (*VaMV*)

with a traditional memory allocation scheme that randomly allocated data to any of the four 1GB DRAMs (*RandomMalloc*). Figure 6 shows that our approach can save an average 7.4% dynamic power consumption by selectively allocating data blocks to the DRAM with the lowest power consumption.

C. Exploiting SRAM Voltage Scaling and DRAM Variability

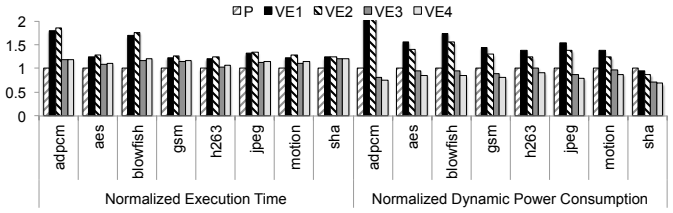

Fig. 7. Exploiting SRAM Voltage Scaling and DRAM Variability.

Next, we combine on-chip SRAM voltage scaling with DRAM variability by progressively augmenting the complexity of our policies (*VE1 to VE4*), and compare their power consumption and performance overheads to the baseline policy (*P*) for a *single* application. The first two custom policies (*VE1/VE2*) incur higher overheads in both power and performance primarily because we utilized the entire on-chip memory space for the E-RAID 1 level (reducing usable on-chip space by half). The next two policies (*VE3/VE4*) utilize the on-chip space much better by partitioning the data into finer E-RAID/NO-ERAID granularities, as a result we observe an average 16% power consumption reduction with 13% performance overheads (with respect to *P*). Memory intensive applications such as H.263 benefit the most from vOM-based policies (e.g., *VE4*) as we observe up to 14% power consumption reduction for H.263 with minimal performance overheads (0.06%). These experiments show that carefully crafting variability-aware *policies* to meet an application's needs leads to reduced dynamic power consumption.

D. Dynamic Policy-driven Variability-aware Allocation

App.	2x2x1	4x2x2	8x2x4
adpcm		Y	Y
aes		Y	Y
blowfish			Y
gsm			Y
h263	Y	Y	Y
jpeg	Y	Y	Y
motion			Y
sha			Y

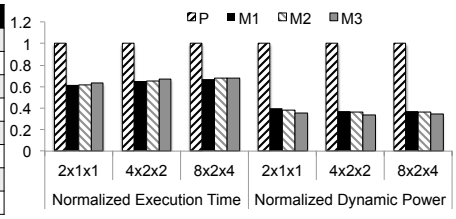

Fig. 8. VaMV: Dynamic Policy-driven Variability-aware Allocation.

In the final set of experiments, we demonstrate VaMV's benefits by simulating a series of virtualized environments running various applications (with support of page tables - 1KB mini-pages) concurrently on a CMP. Figure 8 shows various configurations (x-axis): $\{\#Apps\}x\{\#OSes\}x\{\#CPUs\}$ with 4x8KB physical SPMs and the set of applications ran for each configuration (marked by a **Y** in their respective row/column). The base-line policy (*P*) utilized the entire physical space with context-switching (*CX*) enabled [22] (e.g., swap SPM data on *CX*).We observe that user-defined policies (*M1-M3*) managed to reduce dynamic power consumption by 63% on average while reducing total execution time by an average of 34% because: 1) We have up to $\{8Apps\}x\{4OSes\}x\{4CPUs\}$ competing for memory resources, and traditional malloc (*P*) is unable to cope with the demand, and 2) The *VaMVisor* efficiently manages the memory space by exploiting the idea of variability-aware dynamic *policy*-driven memory allocation.

V. Conclusion

This paper proposed a *novel* Variability-aware Memory Virtualization (*VaMV*) layer that allows programmers to partition their application's address space into virtual regions with different power, performance, and fault-tolerance guarantees through software annotations. *VaMV* adapts to the underlying hardware and virtualizes the memory hierarchy, while opportunistically exploiting techniques such as voltage scaling to reduce on-chip power consumption and power consumption variability present in off-the-shelf off-chip memories. Since this is the first piece of work exploring variability in *distributed* SRAM and DRAM memories, we believe that there are many directions for future work: 1) Studying variability across the entire memory hierarchy (e.g., caches), 2) Exploiting other types of variability (e.g., processor frequency), and 3) Compiler-assisted variability-aware *policy* generation (e.g., efficient address-space partitioning, level of protection).

Acknowledgment

This work was partially supported by NSF Variability Expedition Grant Number CCF-1029783.

References

[1] ITRS, "Process int., dev. and structs." *http://www.itrs.net/*, 2007.
[2] J. Sartori et al., "Variation-aware speed binning of multi-core processors," in *ISQED '10*, 2010.
[3] L. Wanner et al., "A case for opportunistic embedded sensing in presence of hardware power variability," in *HotPower'10*, 2010.
[4] M. Gottscho et al., "Analyzing Power Variability of DDR3 Dual Inline Memory Modules for Applications," *TR UCLA-EE*, 2011.
[5] S. Borkar et al., "Parameter variations and impact on circuits and microarchitecture," in *DAC '03*, 2003.
[6] H. Hanson et al., "Benchmarking for Power and Performance," *2007 SPEC Workshop*, 2007.
[7] A. Pant et al., "Software adaptation in quality sensitive applications to deal with hardware variability," in *GLSVLSI '10*, 2010.
[8] A. Chakraborty et al., "$E<MC^2$: less energy through multi-copy cache," in *CASES '10*, 2010, pp. 237–246.
[9] M. Mutyam et al., "Working with process variation aware caches," in *DATE '07*, 2007.
[10] X. Liang et al., "Process Variation Tolerant 3T1D-Based Cache Architectures," in *MICRO '07*, 2007.
[11] K. Meng et al., "Process variation aware cache leakage management," in *ISLPED '06*, 2006.
[12] X. Li, "Rethinking memory redundancy: optimal bit cell repair for maximum-information storage," in *DAC '11*, 2011.
[13] F. Kurdahi et al., "Low-power multimedia system design by aggressive voltage scaling," *TVLSI*, vol. 18, no. 5, may 2010.
[14] M. Makhzan et al., "Limits on voltage scaling for caches utilizing fault tolerant techniques," in *ICCD '07*, oct. 2007.
[15] L. Bathen et al., "A Case for an Adaptive and Opportunistic Variability-aware Memory Virtualization Layer," in *UCI CECS TR #11-09*, 2011.
[16] ——, "E-RoC: Embedded RAIDs-on-Chip for low power distributed dynamically managed reliable memories." in *DATE '11*, 2011.
[17] IBM, "The cell project," *IBM, http://www.research.ibm.com/cell/*, 2005.
[18] B. Calhoun et al., "A 256-kb 65-nm Sub-threshold SRAM Design for Ultra-Low-Voltage Operation," in *JSSC '07*, 2007.
[19] L. Bathen et al., "SPMVisor: Dynamic ScratchPad Memory Virtualization for Secure, Low Power and High Performance, Distributed On-Chip Memories," in *CODES+ISSS '11, 2011*, october 2011.
[20] C. Lee et al., "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO '97*, 1997.
[21] P. Francesco et al., "An integrated hardware/software approach for run-time scratchpad management," in *DAC '04*, 2004.
[22] H. Takase et al., "Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems," in *DATE '10*, 2010.
[23] T. Austin et al., "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, Feb. 2002.
[24] S. Thoziyoor et al., "Hp labs cacti v5.3," *CACTI 5.1, TR, http://www.hpl.hp.com/ techreports/2008/ HPL-2008-20.html*, 2004.