Variability-Aware Memory Management for Nanoscale Computing

Nikil Dutt[1], Puneet Gupta[2], Alex Nicolau[1], Luis Angel D. Bathen[1], Mark Gottscho[2]

[1]Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3435
e-mail: {dutt, nicolau, lbathen}@ics.uci.edu

[2]Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594
e-mail: puneet@ee.ucla.edu, mgottscho@ucla.edu

Abstract— As the semiconductor industry continues to push the limits of sub-micron technology, the ITRS expects hardware (e.g., die-to-die, wafer-to-wafer, and chip-to-chip) variations to continue increasing over the next few decades. As a result, it is imperative for designers to build variation-aware software stacks that may adapt and opportunistically exploit said variations to increase system performance/responsiveness as well as minimize power consumption. The memory subsystem is one of the largest components in today's computing system, a main contributor to the overall power consumption of the system, and therefore one of the most vulnerable components to the effects of variations (e.g., power). This paper discusses the concept of variability-aware memory management for nanoscale computing systems. We show how to opportunistically exploit the hardware variations in on-chip and off-chip memory at the system level through the deployment of variation-aware software stacks.

I. Introduction

The International Technology Roadmap for Semiconductors (ITRS) predicts that over the next decade, both performance and power consumption variation will increase by up to 66%, and 100%, respectively [28]. Variations can stem from semiconductor manufacturing processes, ambient conditions, device wear out, and in case of multi-sourced systems, vendors:

- *Manufacturing.* The ITRS highlights power/performance variability and reliability management in the next decade as a red brick (i.e., a problem with no known solutions) for design of computing hardware.
- *Environment.* The ITRS projects Vdd variation to be 10% while the operating temperature can vary from -30^oC to 175^oC (e.g., in the automotive context) resulting in over an order of magnitude sleep power variation and several tens of percent performance change.
- *Aging/Wear-out.* Wires and transistors in integrated circuits suffer substantial wear-out leading to power and performance changes over time of usage. Physical mechanisms leading to circuit aging include bias temperature instability, hot carrier injection, and electromigration.
- *Vendor.* Parts with almost identical specifications can have substantially different power, performance or reliability characteristics. This variability is a concern as single vendor sourcing is difficult for large-volume systems.

Despite considerable hardware variability, the software stack has traditionally assumed homogeneity in both frequency and power dissipation for a given hardware specification. Device manufacturers have partially masked the presence of variability by guardbanding systems, leading to over-design with less than optimal power and performance.

An example of guardbanding is the common practice of processor binning based on operating frequencies to reduce the impact of inter-die variation. Despite the use of guardbanding, binning, and dynamic voltage and frequency scaling, variations are inherently present in any set of chips with identical specifications. Furthermore, with the emergence of multi-core technology, intra-die variation has also become an issue [26]. Recent efforts have shown that exploiting the inherent variation in devices [22, 58, 16, 51] yields significant improvements in both the energy-delay product and overall system performance over traditional guardbanding techniques.

Variations also manifest themselves in the memory subsystems in the form of power and timing variations. Given that off-chip DRAM memory may consume as much power as the processor in a server-class system [37, 63, 27, 64], the problem worsens as we move towards emerging many-core platforms (e.g., Tilera's TILEPro64 [57], Intel's Single Chip Cloud Computer (SCC) [24]). A recent study observed up to approximately 20% power variation in an off-the-shelf set of nineteen 1 GB DDR3 DIMMs [19]. On-chip memory designers have tried to create process variation-aware memory subsystems [42, 39, 52, 4] to address this issue, and multiple efforts have been made to minimize off-chip memory accesses via caching [59, 31, 50, 21], OS-level [65, 17, 27], and DRAM-level power management [14, 40, 27]. However, these designs required changes to existing memory configurations. As a result, we should adapt existing DRAM power management schemes in software to account for these variations in power consumption. Moreover, this layer should be flexible enough to deal with a predicted increase in power variation for current [28] and emerging [66, 1] memory technologies (e.g., phase-change memory).

The rest of this paper is organized as follows: Section II gives an overview of the NSF Variability Expedition Project, that attempts to address hardware variations through opportunistic software at the system level. Section III briefly discusses how memory variations may be addressed through efficient memory management. Section IV describes strategies for addressing on-chip and off-chip memory variations by exploiting hardware-assisted memory virtualization. Section V outlines an OS-level approach that exploits DRAM memory power variation to save energy. Finally, Section VI presents the concluding remarks.

II. NSF Variability Expedition Vision: Underdesigned and Opportunistic Computing

We envision a new paradigm for computer systems, one where nominally designed (and hence underdesigned) hardware parts work within a software stack that opportunistically adapts to variations. We call this paradigm *Underdesigned and Opportunistic Computing* (UnO).

UnO machines can be classified along the following two axes:

- Type of Underdesign. Use parametrically under-provisioned circuits (e.g., voltage over-scaling as in [2, 10]) or be implemented with explicitly altered functional description (e.g., [32, 13, 53]);
- Type of Operation. Erroneous operation may rely upon applications' level of tolerance to limited errors (as in [8, 11, 29] to ensure continued operation. By contrast, error-free UnO machines correct all errors (e.g., [2]) or operate hardware within correct-operation limits (e.g., [45, 60]).

The IC design flow will use software adaptability and error resilience for relaxed implementation and manufacturing constraints. UnO machines make a paradigm shift away from a traditional "crash-and-recover" approach to errors, towards an approach that makes proactive measurements, and predicts parametric and functional deviations to ensure continued system operation and availability. This will preempt impact on software applications, rather than just reacting to failures (as is the case in fault-tolerant computing) or under-delivering along power/performance/reliability axes. Fig. 1 shows the UnO adaptation scheme. Underdesigned hardware may be acceptable for appropriate software applications (i.e., the ones that degrade gracefully in presence of errors or are made robust). In other cases, software adaptation may be aided by hardware signatures (i.e., measured hardware characteristics). There are several ways to implement such adaptation ranging from software-guided hardware power management to just-in-time recompilation strategies.

A. Sensing Hardware Signatures

A hardware signature provides a way of capturing one or more hardware characteristics of interest and transmitting these characteristics to the software at a given point in time. Examples of hardware signatures include performance, power, temperature, error rate, delay fluctuations and working memory size.

Hardware signatures may be collected at various levels of spatial and temporal granularities depending on hardware and software infrastructure available to collect them and their targeted use in the UnO context. Part of the NSF Variability Expedition goal is to develop efficient monitoring methods: replica-based (e.g., [9]), in-situ (e.g., [7]), online self-test (e.g., [38]) or software-based inference (e.g., [49]). Since each method of runtime sensing inevitably makes a fundamental trade-off between cost, accuracy and applicability across various stages of system lifetime, it is necessary to combine these approaches to meet the area/power/test time/accuracy constraints.

B. Variability-Aware Software

Hardware variability may be visible to the software in several ways: changes in the availability of modules in the platform (e.g., a processor core not being functional); changes in module speed or energy performance (e.g., the maximum feasible frequency for a processor core becoming lower or its energy efficiency degraded); and changes in error rate of modules (e.g., an ALU computing wrong results for a higher fraction of inputs). The range of possible responses that the software can make is rich: alter the computational load (e.g., throttle the data rate to match what the platform can sustain); use a different set of hardware resources (e.g., use instructions that avoid a faulty module or minimize use of a power hungry module); change the algorithm (e.g., switch to an algorithm that is more resilient to computational errors); and change hardware's operational setting (e.g., tune software-controllable control knobs such as voltage/frequency).

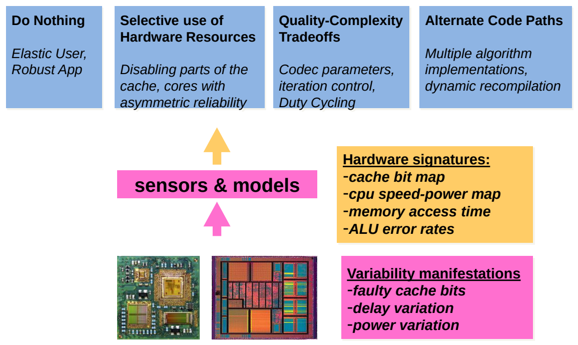

Do Nothing
Elastic User, Robust App
Selective use of Hardware Resources
Disabling parts of the cache, cores with asymmetric reliability
Quality-Complexity Tradeoffs
Codec parameters, iteration control, Duty Cycling
Alternate Code Paths
Multiple algorithm implementations, dynamic recompilation
sensors & models
Hardware signatures:
-cache bit map
-cpu speed-power map
-memory access time
-ALU error rates
Variability manifestations
-faulty cache bits
-delay variation
-power variation

Fig. 1. Examples of UnO adaptation aided by embedded hardware monitoring.

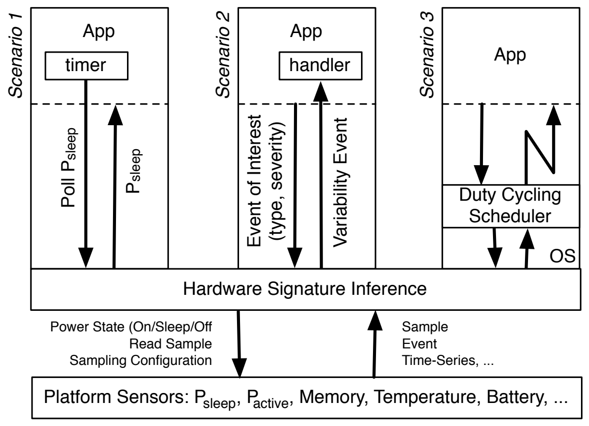


Fig. 2. Designing a Software Stack for Variability-Aware Duty Cycling

Recently, there have been some efforts to handle variability at higher layers of abstraction. For instance, software schemes have been used to address voltage [47] or temperature variability [12]. Hardware "signatures" are used to guide adaptation in quality-sensitive multimedia applications in [45]. Using architecture similar to that of Scenario 3 in Fig. 2, [60] explored a *variability-aware duty cycle scheduler* where application modules specify a range of acceptable duty cycling ratios, and the scheduler selects the actual duty cycle based on run-time monitoring of operational parameters, and a power-temperature model that is learned off-line for the specific processor instance. In context of erroneous UnO machines, several alternatives exist ranging from detecting and then correcting faults within the application as they arise (e.g., [23, 25]) to designing applications to be inherently error-tolerant (e.g., application robustification [55]).

C. Building Underdesigned Hardware

Unfortunately, current design methodologies are not always suitable for such underdesign. For example, conventional processors are optimized such that all the timing paths are critical or near-critical ("timing slack wall"). This means that any time an attempt is made to reduce power by trading off reliability (by reducing voltage, for example), a catastrophically large number of timing errors is seen [29]. The slack distribution can be manipulated (for example, to make it look gradual, instead of looking like a wall, to reduce the number of errors when power is reduced [30]).

Though most existing work [44, 34, 18] introduces errors by intelligently over-scaling voltage supplies, there has been some work in the direction of introducing errors into a system via manipulation of its logic-function. Let us consider an example of a functionally underdesigned integer multiplier circuit [32] with potentially erroneous operation. We use a modified 2x2 multiplier as a building block which computes 3×3 as 7 instead of 9. By representing the output using three bits (111) instead of the usual four (1001), we are able to significantly reduce the complexity of the circuit. These inaccurate multipliers achieve an average power saving of 31.78% - 45.4% over corresponding accurate multiplier designs, for an average percentage error of 1.39% - 3.32%. The design can be enhanced (albeit at power cost) to allow for correct operation of the multiplier using a correction unit, for non error-resilient applications which share the hardware resource. Of course the benefits are strongly design-dependent. For instance, in the multiplier case, the savings translate to 18% saving for a FIR filter and only 1.5% for a small embedded microprocessor. Functional underdesign will be a useful power reduction and/or performance improvement knob especially for robust and gracefully degrading applications.

At its core, handling variability of hardware specification at run-time amounts to detecting the variability using hardware or software based sensing mechanisms, followed by selecting a different execution strategy in the UnO machine's hardware or software or both. Our early results illustrating UnO hardware-software stack are very promising. For instance, our implementation of variability-aware dutycycling in TinyOS gave over 6X average improvement in active time for embedded sensing. With shrinking dimensions approaching physical limits of semiconductor processing, this variability and resulting benefits from UnO computing are likely to increase in future.

III. Variations in the Memory Subsystem

Like processors, where power consumption variability across various dies has been reported despite following the same design specs [22, 61, 51, 46], similar phenomena have been observed in the memory subsystem [22, 19]. [19] tested 19 DDR3 DIMMs, and found that power usage in DRAMs is dependent both on operation type (write, read, and idle) as well as data type. Idle power variations were up to 12.29% for the same model (same vendor) and 16.40% for different models from the same vendor. More importantly, dynamic power was also affected as there was up to 21.84% variation in write power across the different 1 GB DIMMs tested.

These variations can be observed in Fig. 3, where power (write, read, and idle) variability is shown across various categories (e.g., across all nineteen 1 GB DIMMs tested, across same exact parts, etc.) tested at 30 C. As a result, just like [62, 51, 46] exploited variability in processor power consumption, our goal is to adapt our memory management layer and opportunistically take advantage of these variations to reduce power consumption.

Furthermore, the move towards deep sub-micron (DSM) technologies makes integrated circuits increasingly vulnerable to both transient errors, as well as process-variation induced errors [15, 43, 6, 41, 48, 20, 54]. The problem is further exacerbated as systems are aggressively voltage scaled to save power, resulting in even higher error rates for embedded memories.

One major concern for memories is the increasing incidence of soft errors, i.e., transient faults, or single-event upsets (SEU), that are caused primarily by external radiations in microelectronic circuits. As alpha or cosmic particles come into contact with a silicon device, the probability of a single event failure increases with decreasing charge within a memory cell [36]. Aggressive voltage scaling greatly reduces the capacitance that keeps the charge in a single cell, therefore increasing its vulnerability to low energy alpha particles, or cosmic rays [36].

Memories are believed to be the most vulnerable components to soft-errors since they may occupy up to 90% of the on-chip real-estate [33, 28]. Indeed, this problem worsens for

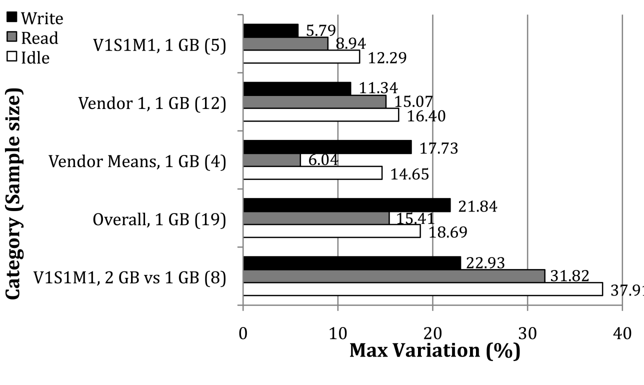


Fig. 3. Power Variation in Off-The-Shelf DIMMs [19]

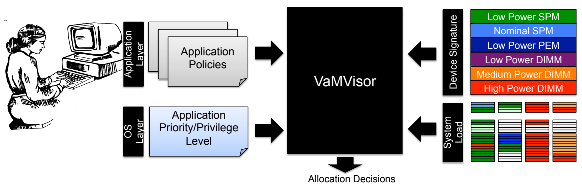


Fig. 4. VaMV: Variability-aware Memory Virtualization

many-core platforms, as a result, software must account for the increased probabilities of failure in the memory subsystem.

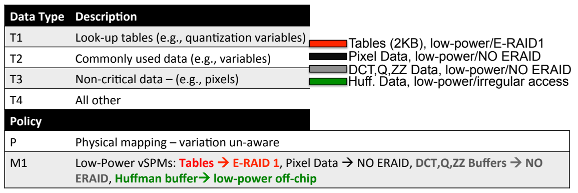
The next two sections describe two approaches that leverage software and hardware to opportunistically exploit the variations introduced in this section.

IV. Hardware-assisted Variation-aware Memory Virtualization

To address and exploit these variations, [5] proposed the concept of *VaMV*: a power variation-aware memory virtualization layer for scratchpad memory-based chip-multiprocessors. The goal of the 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. Fig. 4 shows a high-level view of VaMV, which has five main components: 1) the application's programmer-specified source-code annotated data mapping *policies*, 2) the application's *priority*, which is used to prioritize use of the memory space among the various applications, 3) the device's signature, 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, and 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.

A. Programmer-driven Address Space Partitioning

In order to efficiently exploit the available memory variability (on- and off-chip [5]), programmers can provide VaMV with data mapping hints in the form of policies. A programmer will consider the application's requirements and partition its virtual address space into 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).



Data Type	Description
T1	Look-up tables (e.g., quantization variables)
T2	Commonly used data (e.g., variables)
T3	Non-critical data – (e.g., pixels)
T4	All other
Policy	
P	Physical mapping – variation un-aware
M1	Low-Power vSPMs: **Tables → E-RAID 1**, Pixel Data → NO ERAID, **DCT,Q,ZZ Buffers → NO ERAID**, **Huffman buffer→ low-power off-chip**

🟥 Tables (2KB), low-power/E-RAID1
⬛ Pixel Data, low-power/NO ERAID
▨ DCT,Q,ZZ Data, low-power/NO ERAID
🟩 Huff. Data, low-power/irregular access

Fig. 5. Sample User Annotations and Policies

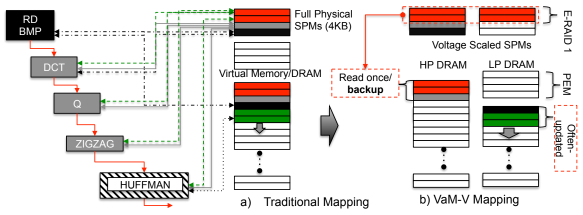
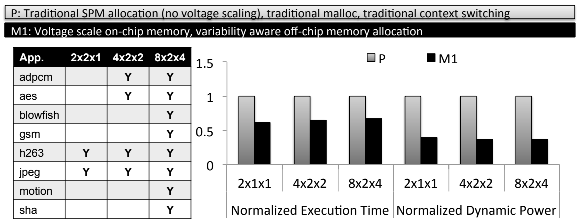


Fig. 6. Partitioning the Application's Memory Space

Fig. 5 shows a *sample* address space partitioning for JPEG [35], where the programmer has identified: 1) read-only and highly utilized data, e.g., look up tables (red blocks), 2) a temporary buffer for inter-task communication (gray block), 3) read-only pixel data (black blocks), and 4) irregularly accessed data (green blocks).

Fig. 6 (a) shows a traditional mapping of these data blocks, where variability is *not* taken into account. Fig. 6 (b) shows the result of VaMV's 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 [4] 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 with knowledge of the application's requirements can create custom data mapping policies with low-power (*LP*) memory space in mind. VaMV then takes these policies and tries to opportunistically enforce them, regardless of how the LP memory space is implemented by the hardware layer. For instance: 1) 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*), or 2) if voltage scaling on-chip memories is not possible, VaMV will proceed to treat all on-chip memories the same.

Consider the case where there is power and latency variation in both on-chip (due to voltage scaling the SPMs) and off-chip memories (due to the inherent hardware-variability in DRAM memories). The programmer can then partition each application's virtual address space as shown in Fig. 5, and define allocation policies for each virtual address space as shown in Fig. 6 (pictorially shown by adding a "color" to each virtual address space according its requirements). These annotations are used by VaMV's run-time system, which opportunistically exploits the variations in the memory subsystem. Each application will



P: Traditional SPM allocation (no voltage scaling), traditional malloc, traditional context switching
M1: Voltage scale on-chip memory, variability aware off-chip memory 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. 7. Dynamic Policy-driven Variability-Aware Allocation

then have different requirements (e.g., fault-tolerant memory space, secure memory space, etc.), some being more critical than others (e.g., *h263*'s higher memory footprint requirements vs. *sha*'s need for full memory isolation).

B. Applying Policies to Runtime Allocations

Fig. 7 shows various configurations of a chip multi-processor (CMP) system that has SPMs for each processor, with multiple applications managed by multiple operating systems (OSes) execution on this CMP; the triplet on the x-axis represents {*#Apps*}x{*#OSes*}x{*#CPUs*} with 4x8KB physical SPMs and the set of applications run 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 [56] (e.g., swap SPM data on *CX*). Policy *M1* uses vSPMs and allows VaMV to dynamically map each application's data. Because we are running various applications concurrently, VaMV needs to prioritize and judiciously map different data to on-chip and off-chip memory. The data sets (*T1-4*) in Fig. 7 represent a high-level abstraction of the application's workload. In this experiment, despite mapping all *T1-3* data to vSPM for a given application, it is not guaranteed that the data will go into physical SPM space, as the resources are limited (only 4x8KB). So VaMV prioritizes among all data blocks of each category (*T1-4*), and based on their priorities (*BlkPriority=block utilization*), decides where to map the data. For example, *h263* has much higher on-chip/off-chip memory requirements, as a result, higher priority is given to *h263*'s *T1-4* blocks than the other applications (on-chip SRAM and low-power DRAM). User-defined policies (represented by *M1*) managed to reduce dynamic power consumption by 63% on average while reducing total execution time by an average of 34% because: 1) there are up to {*8Apps*}x{*4OSes*}x{*4CPUs*} competing for memory resources, and traditional malloc (*P*) is unable to efficiently cope with the demand, and 2) VaMV efficiently manages the memory space by exploiting the idea of variability-aware dynamic policy-driven memory allocation.

V. OS-Based Variability-Aware Memory Allocation

For more conventional systems without scratchpad memories, a different approach from VaMV is needed. Here, we introduce ViPZonE [3], an OS-based variability-aware memory management solution that targets server-class systems with many DDR3 memory modules (DIMMs). Essentially, ViPZonE allows the application programmer to hint to the OS the expected usage patterns for dynamic memory allocations (e.g., write or read-dominated, high or low utilization). With sufficient knowledge about the power consumption characteristics of the underlying DRAM system, the OS can physically allocate pages for a process using these hints to reduce memory power consumption. Although our approach targets the DIMM modular level (motivated by power variability data from [19]), it could be generalized to work at arbitrary granularities of memory, if power data are available.

ViPZonE is currently implemented for a Linux-based, high-end x86-64 desktop system. Its software stack is composed

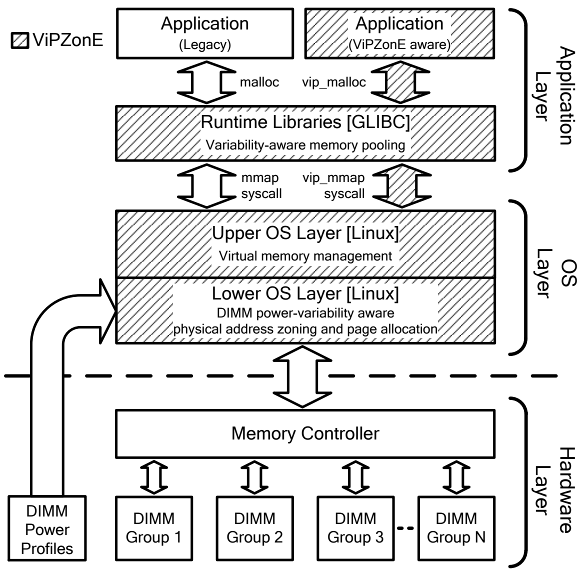


Fig. 8. ViPZonE Software Architecture

of a modified GLIBC layer (with a special version of *malloc*), and a modified kernel with power variability-aware allocation features and a new system call. Fig. 8 depicts the software architecture of ViPZonE. Note that it has no need for special hardware support beyond the availability of DIMM power data (e.g., characterization is done at boot time through embedded power sensors, which is sufficient for instance-to-instance static variability that changes slowly over time). We discuss the implementation of ViPZonE in a top-down manner.

A. Front-End Implementation

One of the features of the ViPZonE software stack is a minimal impact on application development. ViPZonE-enabled applications simply use the API *vip_malloc* for dynamic allocations. Implemented in the GLIBC library, the API only requires a single additional parameter which is a bitwise-OR of two flags. *vip_malloc* relies on a new syscall, *vip_mmap*, which embeds the bitwise ViPZonE flags into the *prot* bits (as there are no free bits in *flags*, and Linux does not support a seventh syscall parameter). Available flags for the user include *VIP_READ* or *VIP_WRITE*, combined with *VIP_HI_UTIL* or *VIP_LO_UTIL*. Table I summarizes the available API.

TABLE I
ViPZonE User-space API

Function	Parameter	Type	Description
*void * vip_malloc* (GLIBC function)	size	size_t	Request size in bytes
	vip_flags	int	Bitwise flags used by ViPZonE back-end page allocator
*void * vip_mmap* (kernel syscall)	addr	void *	Address to be mapped. (best effort mapping)
	length	size_t	Size to be allocated
	prot	int	PROT_READ |PROT_WRITE |vip_flags
	flags	int	MAP_PRIVATE |MAP_ANONYMOUS
	fd	int	File descriptor
	offset	off_t	Page offset

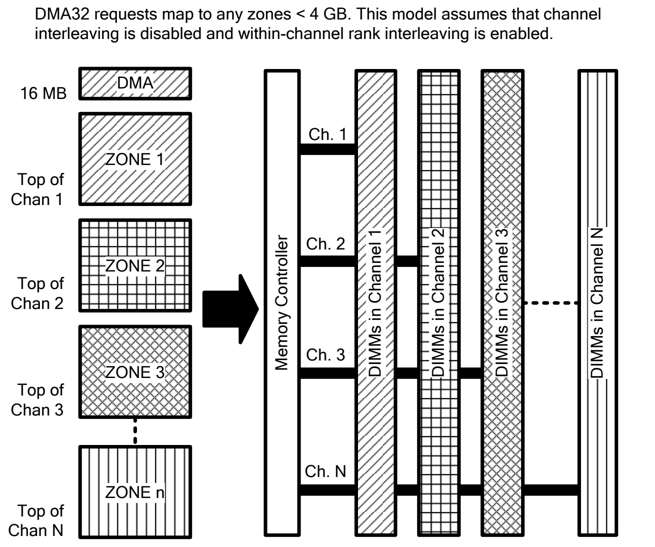


Fig. 9. ViPZonE Physical Address Space Partitioning (Zoning) in the Linux Kernel for x86-64

An application could use the ViPZonE API to reduce memory power consumption by intelligently using the flags. For example, if a piece of code will use a dynamically-allocated array for heavy read utilization (e.g., an input for matrix multiplication), then it can request memory as follows:

```
retval = vip_malloc(arraySize*elementSize, VIP_READ
    | VIP_HI_UTIL);
```

Alternatively, the application could use the syscall directly:

```
retval = vip_mmap(NULL, arraySize*elementSize,
    PROT_READ | PROT_WRITE | VIP_READ | VIP_HI_UTIL
    , MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

For each *vip_mmap* call, the kernel tries to either expand an existing VM area that will suit the set of flags, or create a new area. When the kernel handles a request for physical pages, it checks the associated VM area, if applicable, and can use the ViPZonE flags passed from user-space to make an informed allocation.

B. Back-End Implementation

At some point during process execution, physical pages must be allocated, usually on demand from the page fault mechanism. For physical page allocations that are directly related to a virtual memory region, the ViPZonE page allocator checks the associated ViPZonE flags. By default, the flags are a combination of *VIP_READ* with *VIP_LO_UTIL* for user-space, and *VIP_READ* with *VIP_HI_UTIL* for kernel-space (this assumes that the kernel's memory will be referenced heavily over time and should always be given preferred low power space). Any applications that used the *vip_mmap* syscall will have their chosen flags handled appropriately by the physical page allocator.

At boot time, the kernel constructs a set of zones (contiguous sets of physical addresses), to align with the different physical memories in the system. The current implementation assumes that channel interleaving is disabled, while within-channel rank interleaving is enabled. Peak bandwidth could potentially be reduced for some workloads due to the lack of channel interleaving, but each channel is still independently accessible.

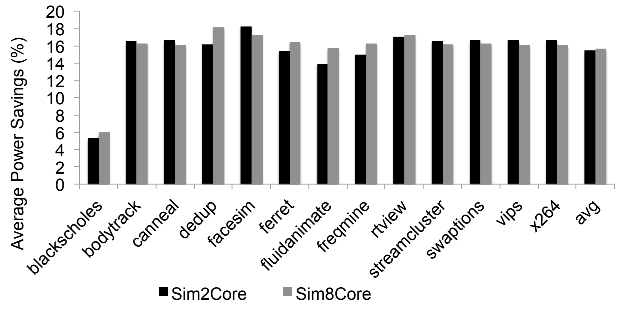


Fig. 10. Simulated Memory Power Savings of ViPZonE vs. Traditional Linux Kernel With PARSEC Benchmarks

By disabling channel interleaving and physically partitioning the address space, we can achieve power savings by allowing more DIMMs to be idle while also harnessing power variability across channels of DIMMs. This scheme is depicted in Fig. 9.

Using data on each DIMM's power consumption, obtained either through power sensors at boot time or offline profiling, the ViPZonE kernel constructs ordered lists of all the zones in the system by write and read power consumption. Together with the ViPZonE flags, it can choose the most suitable zone to grant an allocation at runtime. A number of allocation algorithms of varying complexity are possible. ViPZonE, in the interest of simplicity and performance, uses a simple thresholding scheme. For allocations with "high utilization" flags, it attempts to allocate in the low power zones first. For allocations that do not directly request it, it still prefers low power space, only if a given zone has at least *threshold* amount of free space that is reserved for allocations that ask for it.

C. Simulation Results and Ongoing Work

Our simulation of PARSEC benchmarks on a dual-core system with two DIMMs (30% power variation, as suggested by [19]) [3] indicated promising memory power savings of around 13.1% with only a 1% increase in execution time, compared to the traditional Linux kernel (see Fig. 10). In our ongoing work, we are evaluating ViPZonE on an Intel Core i7-based testbed with four DDR3 memory channels that are populated with eight 2 GB DIMMs. This system is fully instrumented for memory power measurements. The goal for this evaluation is to characterize the real performance and power impact of the ViPZonE software stack to achieve energy savings for typical workloads.

VI. Summary and Conclusions

In this paper, we focused on two examples of variability-aware memory management for on- and off-chip memories (VaMV and ViPZonE), in context of the larger NSF Variability Expedition project. The Expedition attempts to holistically and proactively exploit hardware variability across multiple abstraction levels, as well as across different subsystems of computing platforms, part of a larger push for the Underdesigned and Opportunistic (UnO) computing paradigm. Such solutions will be necessary to improve the power consumption and performance of future systems with ever-increasing amounts of variability due to technology scaling.

Acknowledgment

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

References

[1] C. Augustine et al. Spin-transfer torque mrams for low power memories: Perspective and prospective. *Sensors Journal, IEEE*, 2012.

[2] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. *Making Typical Silicon Matter with Razor*. Mar. 2004.

[3] L. Bathen, M. Gottscho, N. Dutt, P. Gupta, and A. Nicolau. ViPZonE: OS-level memory variability-driven physical address zoning for energy savings. In *ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2012.

[4] L. Bathen et al. E-RoC: Embedded RAIDs-on-Chip for low power distributed dynamically managed reliable memories. In *DATE*, 2011.

[5] L. Bathen et al. VaMV: Variability-aware Memory Virtualization. In *DATE*, 2012.

[6] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance cmos variability in the 65-nm regime and beyond. *IBM J. Res. Dev.*, 50:433–449, July 2006.

[7] D. Blaauw, S. Kalaiselvan, K. Lai, W.-S. Ma, S. Pant, S. Tokunaga, S. Das, and D. Bull. Razor ii: In situ error detection and correction for pvt and ser tolerance. In *IEEE ISSCC*, 2008.

[8] M. Breuer, S. Gupta, and T. Mak. Defect and error tolerance in the presence of massive numbers of defects. 21(3), 2004.

[9] T. B. Chan, P. Gupta, A. B. Kahng, and L. Lai. DDRO: A novel performance monitoring methodology based on design-dependent ring oscillators. In *IEEE ISQED*, 2012.

[10] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *IEEE/ACM DAC*, 2010.

[11] H. Cho, L. Leem, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. 31(4), 2012.

[12] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-aware task scheduling at the system software level. In *IEEE/ACM ISLPED*, 2007.

[13] M. R. Choudhury and K. Mohanram. Approximate logic circuits for low overhead, non-intrusive concurrent error detection. In *IEEE/ACM DATE*, 2008.

[14] V. Delaluz et al. Scheduler-based dram energy management. In *DAC*, 2002.

[15] A. Devgan and S. Nassif. Power variability and its impact on design. In *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, VLSID '05, pages 679–682, 2005.

[16] J. Dong et al. Variation-aware scheduling for chip multiprocessors with thread level redundancy. In *PRDC*, 2009.

[17] W. Felter et al. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS*, 2005.

[18] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. 2006.

[19] M. Gottscho, A. A. Kagalwalla, and P. Gupta. Power variability in contemporary DRAMs. *IEEE Embedded Systems Letters*, 2012.

[20] T. Granlund, B. Granbom, and N. Olsson. Soft error rate increase for new generations of srams. *Nuclear Science, IEEE Trans. on*, 50(6):2065 – 2068, dec. 2003.

[21] X. Gu et al. P-opt: Program-directed optimal cache management. In J. N. Amaral, editor, *LCPC*. 2008.

[22] H. Hanson et al. Benchmarking for power and performance. *2007 SPEC Workshop*, 2007.

[23] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *IEEE/ACM ISLPED*, 1999.

[24] J. Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC*, 2010.

[25] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6), 1984.

[26] E. Humenay et al. Impact of process variations on multicore performance symmetry. In *DATE*, 2007.

[27] I. Hur et al. A comprehensive approach to dram power management. In *HPCA*, 2008.

[28] ITRS. *http://www.itrs.net/*.

[29] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing processors from the ground up to allow voltage/reliability tradeoffs. In *International Symposium on High-Performance Computer Architecture*, 2010.

[30] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *The 15th IEEE/SIGDA Asia and South Pacific Design and Automation Conference*, 2010.

[31] M. Kandemir. Impact of data transformations on memory bank locality. In *DATE*, pages 10506–, 2004.

[32] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power in a multiplier architecture. *Journal of Low Power Electronics*, 2011.

[33] F. Kurdahi et al. Low-power multimedia system design by aggressive voltage scaling. *TVLSI*, 18(5), may 2010.

[34] M. S. Lau, K.-V. Ling, and Y.-C. Chu. Energy-aware probabilistic multiplier: design and analysis. 2009.

[35] C. Lee et al. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO '97*, 1997.

[36] K. Lee et al. Mitigating soft error failures for multimedia applications by selective data protection. In *CASES '06*, 2006.

[37] C. Lefurgy et al. Energy management for commercial servers. *Computer*, 2003.

[38] Y. Li, Y. Kim, E. Mintarno, D. Gardner, and S. Mitra. Overcoming early-life failure and aging challenges for robust system design. *IEEE DTC*, 26(6), 2009.

[39] X. Liang et al. Process variation tolerant 3t1d-based cache architectures. In *MICRO '07*, 2007.

[40] C.-G. Lyuh et al. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *DAC*, DAC '04, 2004.

[41] R. Mastipuram and E. C. Wee. Soft errors' impact on system reliability. In *http://www.edn.com/ article/ CA454636*, September 2004.

[42] M. Mutyam et al. Working with process variation aware caches. In *DATE '07*, 2007.

[43] S. Nassif. Modeling and analysis of manufacturing variations. In *Custom Integrated Circuits, 2001, IEEE Conf. on.*, pages 223 –228, 2001.

[44] K. V. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 54(9), 2005.

[45] A. Pant, P. Gupta, and M. v.-d. Schaar. AppAdapt: Opportunistic application adaptation to compensate hardware variation. *IEEE Transactions on VLSI*, 2012.

[46] A. Pant et al. Software adaptation in quality sensitive applications to deal with hardware variability. In *GLSVLSI '10*, 2010.

[47] V. J. Reddi, M. S. Gupta, M. D. Smith, G.-y. Wei, D. Brooks, and S. Campanoni. Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack. In *IEEE/ACM DAC*. ACM, 2009.

[48] F. Ruckerbauer and G. Georgakos. Soft error rates in 65nm srams–analysis of new phenomena. In *On-Line Testing Sym., 2007. IOLTS 07. 13th IEEE Int.*, pages 203 –204, july 2007.

[49] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN*, June 2008.

[50] J. Sartor et al. Cooperative caching with keep-me and evict-me. In *INTERACT*, 2005.

[51] J. Sartori et al. Variation-aware speed binning of multi-core processors. In *ISQED*, 2010.

[52] A. Sasan et al. Process variation aware sram/cache for aggressive voltage-frequency scaling. In *DATE*, 2009.

[53] D. Shin and S. K. Gupta. Approximate logic synthesis for error tolerant applications. In *IEEE/ACM DATE*, 2010.

[54] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, 2002.

[55] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *IEEE DSN*, 2010.

[56] H. Takase et al. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *DATE '10*, 2010.

[57] Tilera. Tilepro 64. *http://www.tilera.com/*, 2010.

[58] F. Wang et al. Variation-aware task allocation and scheduling for mpsoc. In *ICCAD*, 2007.

[59] Z. Wang et al. Power aware variable partitioning and instruction scheduling for multiple memory banks. In *DATE*, 2004.

[60] L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava. Hardware variability-aware duty cycling for embedded sensors. *IEEE Transactions on VLSI*, 2012.

[61] L. Wanner et al. A case for opportunistic embedded sensing in presence of hardware power variability. In *HotPower'10*, 2010.

[62] L. Wanner et al. Variability-aware duty cycle scheduling in long running embedded sensing systems. In *DATE '11*, 2011.

[63] J. Yue et al. Evaluating memory energy efficiency in parallel i/o workloads. In *CLUSTER*, 2007.

[64] H. Zheng et al. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *MICRO*, 2008.

[65] P. Zhou et al. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.

[66] P. Zhou et al. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09*, 2009.