Center for Embedded Computer Systems
University of California, Irvine

A Case for an Adaptive and Opportunistic Variability-Aware Memory Virtualization Layer

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

[1]Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
{lbathen, nicolau, dutt}@uci.edu

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

CECS Technical Report #11-09
November 15, 2011

A Case for an Adaptive and Opportunistic Variability-aware Memory Virtualization Layer

LUIS ANGEL D. BATHEN, University of California, Irvine
PUNEET GUPTA, University of California, Los Angeles
ALEX NICOLAU, University of California, Irvine
NIKIL D. DUTT, University of California, Irvine

Device variability in power consumption (e.g., sleep, active) and performance (e.g., frequency) is expected to continue to increase in the orders of magnitude over the next decades. In order to be opportunistic and account for hardware variability, designers must build an adaptive hardware/software stack that will efficiently manage the underlying hardware resources. This paper makes several contributions: 1) We propose a *first-of-its-kind* Hardware-assisted Variability-aware Memory Virtualization (*VaMV*) layer that allows programmers/applications to partition their address space into regions with different power, performance, and fault-tolerance guarantees (e.g., map look-up tables into low-power fault-tolerante space or pixel data in low-power non-fault-tolerant space). *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. 2) To the best of our knowledge, we are the first to explore the notion of variability-aware policy-driven memory allocation for $distributed$ on-chip and off-chip memories. 3) We propose a *proof-of-concept* hardware-module called *VaMVisor*, which allows us to minimize the overheads incurred by virtualization and dynamic allocation of the memory space. Finally, 4) We define an API to facilitate the creation and management of virtual ScratchPad Memories (vSPMs) and virtual Off-chip Memories (vOMs). Our experimental results on a set of benchmarks (Mediabench I/II and CHStone) show that our approach is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 45%.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based systems**]: Real-time and embedded systems; D.4.6 [**Security and Protection**]: Access Controls; Security Kernels; B.3 [**Design Styles**]: Virtual Memory; D.4 [**Storage Management**]: Distributed memories

General Terms: Design, Management, Performance, Security

Additional Key Words and Phrases: information assurance; security; chip-multiprocessors; policy; scratch-pad memory; virtualization; embedded systems

1. INTRODUCTION

Hardware variability is quickly becoming one of the major topics of concern in the design community. ITRS predicts that over the next decade performance variability will increase from 48% to 66% [1; 2], while sleep and total power consumption variability will increase by up to 500% and 100% [1; 3] respectively. There are many sources/-factors that influence the variation within and across devices. From parameters such as temperature, voltage, current, mask imperfections, wear-out mechanisms, to ven-

This research was partially supported by NSF Variability Expeditions Award CCF-1029783, and SFS/NSF Grant No. 0723955.
Authors' addresses: Luis Angel D. Bathen and Nikil Dutt, Center for Embedded Computer Systems, School of Information and Computer Science, University of California at Irvine, Irvine, CA 92697;

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

dors and processes used . Variability plays a major role not only on system performance and power consumption but also in design costs, since high degrees of variability might cause a device to be disregarded, hence reducing the production yield (and increasing production costs) [4]. Due to this expected increase in variability, designers must built adaptable and tunable software/hardware to deal with these variabilities. Various efforts in the community have shown that exploiting variability in off-the-shelf hardware may lead to very promising results [3; 2; 5; 6], thus motivating this work.

At the memory subsystem, designers have exploited aggressive voltage scaling to reduce power consumption at the cost of exponentially increasing the impact of process variation on memory cells [7]. Failures due to process variation worsen as process technology reaches its limits [8; 9; 10]. Although the probability of failure in SRAM technology is exponentially proportional to the decrease in voltage, research has shown how to efficiently exploit voltage scaling while handling process variations [11; 8; 12; 13] to save power consumption.

Finally, the rapid adoption of multiprocessor technology ([14; 15; 16]) along with the integration of distributed scratchpad memories (SPMs) into the memory hierarchy [14] (due to their increased predictability, reduced area and power consumption [17]) further exacerbates the need to built adaptable hardware/software to account with inter/intra die variability [2; 18].

In this paper we propose a *first-of-its-kind* Hardware-assisted Variability-aware Memory Virtualization (*VaMV*) layer that allows programmers/applications to partition their address space into regions with different power, performance, and fault-tolerance guarantees. *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. We define an API to facilitate the creation and management of virtual ScratchPad Memories (vSPMs) and virtual Off-chip Memories (vOMs) and show how a programmer can take advantage of our API by taking a commonly known application and tuning it to exploit the idea of variability-aware data partitioning and policy generation. The *VaMVisor* then takes the these policies, and makes real-time allocation decisions based on the application's priority, the block's priority, and the type of protection needed. To the best of our knowledge, we are the first to explore the notion of variability-aware policy-driven memory allocation for $distributed$ on-chip and off-chip memories. We propose a *proof-of-concept* hardware-module called *VaMVisor*, which allows us to minimize the overheads incurred by virtualization and dynamic allocation of the memory space.

The key contributions of this paper are:

— Introduced the concept of variability-aware application data partitioning
— A first attempt at exploiting and co-optimizing on-chip and off-chip memory variability
— A *first-of-its-kind* hardware-assisted dynamic variability-aware memory virtualization (*VaMV*) layer
— A dynamic and efficient resource-management mechanism built on the idea of policy-driven variability-aware allocation
— API for dynamic and transparent on-chip resource management to exploit the notion of memory variability

Our experimental results on a set of benchmarks (Mediabench I/II and CHStone) show that our approach is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 45%.

2. MOTIVATION

Although there are many types of variabilty across different components, in this paper we will focus on memory variability, primarily two types: 1) Off-the-shelf memory variability (same vendor and specs), and 2) On-chip memory variability (latency, power consumption, error rates) as an effect of voltage scaling. Our goal is to motivate the need for exploiting and co-optimizing on-chip and off-chip memory variability in a holistic manner with the goal of reducing power consumption.

2.1. Off-Chip Memory Variation

Like processors [19; 3; 2; 5], where power consumption variability across various dies has been reported despite following the same design specs, the same phenomena has been observed at the memory subsystem [19] and [18]. Gottscho et al. [18] reported that up to 20.25% power variability was observed across a series of 1GB DIMMs and up to 16.77% power variation across 1GB DIMMs belonging to the *same* vendor. As a result, just like [6; 2; 5] exploited variability in processor power consumption, our goal is to adapt our memory management layer and opportunistically take advantage of these opportunities to reduce power consumption. Figure 1 shows the power (READ, WRITE, IDLE) variability across seven 1GB @ 1066 MHz V1S1M{1/2} (Vendor, Specs, Manufacturer), where an average of 10.8% write power variation and up to 26.3% write power variation was observed.

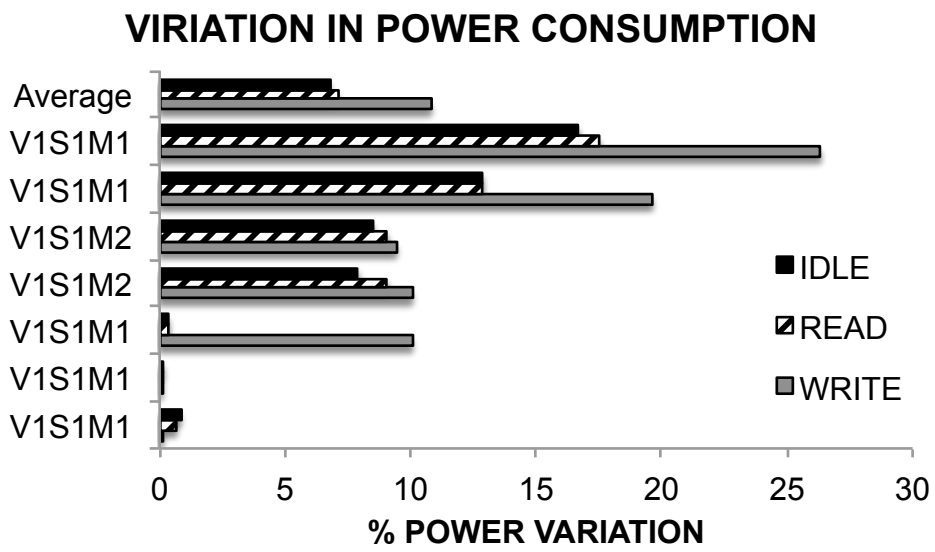

Fig. 1. Power Variation in off-the-shelf 1GB DIMMs. [18]

2.2. On-Chip Memory Process Variation

Figure 2 shows the probability of failure (P_{Fail}) on the $y-axis$ and voltage on the $x-axis$. As voltage is reduced from nominal Vdd (>0.9 Vdd) to 0.7 Vdd, P_{Fail} increases exponentially. Voltage scaling is a very promising technique to reduce on-chip memory power consumption at the cost of introducing errors on the memory subsystem [8; 7; 20]. Moreover, voltage scaling affects the access latencies of on-chip memories [8; 7],

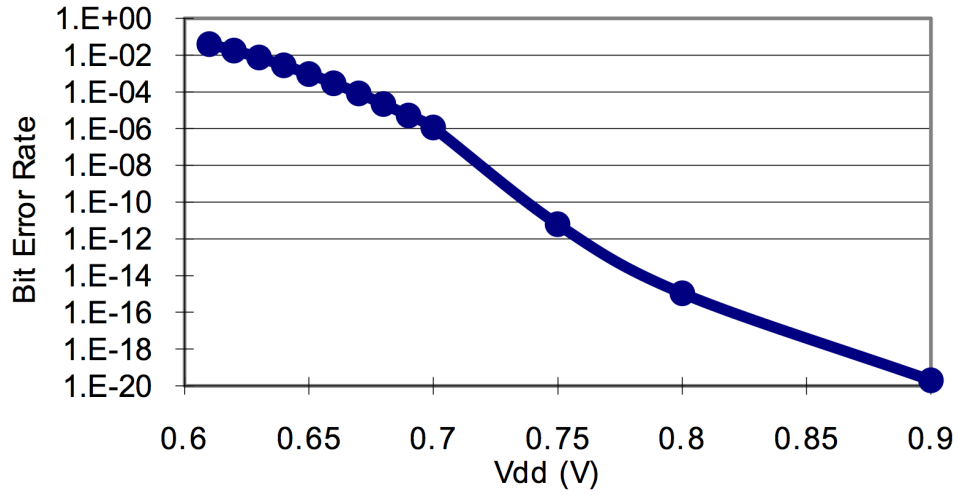

Fig. 2. Effects of Voltage Scaling [8].

as a result, it is possible to have different types of on-chip memory variability (access latency, static and dynamic power, error rate).

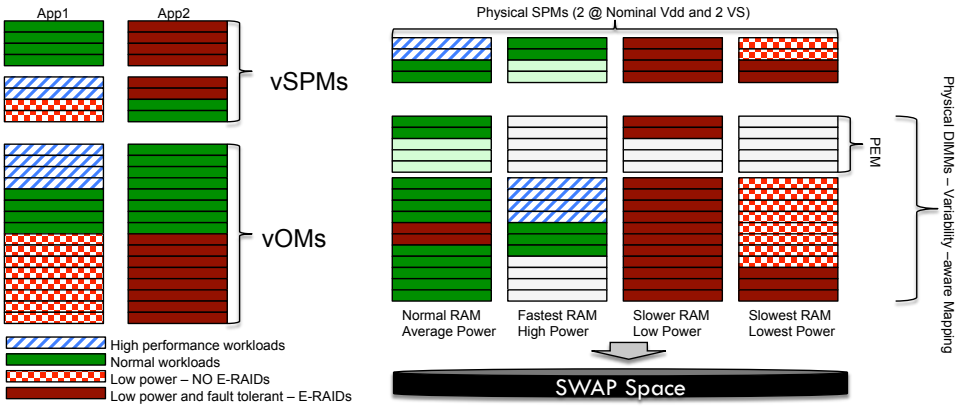

Fig. 3. Variability-aware Memory Virtualization.

3. VARIABILITY-AWARE MEMORY VIRTUALIZATION

3.1. Overview and Goals

The goal of of our Variability-aware Memory Virtualization (*VaMV*) layer is to opportunistically exploit variability across various levels of the memory hierarchy in order to reduce overall power consumption. The key idea is to provide a hardware-assisted virtualization infrastructure (via *VaMVisor*) to allow programmers/compilers/OSes the ability to partition their virtual address space into regions, where each region has different power, performance, fault-tolerance guarantees. Figure 3 shows a high-level view of *VaMV*'s memory space. In this figure, we have two applications with different needs: 1) High-performance (low-latency) address space represented by dashed blocks,

2) Normal (no fault-tolerance, no latency requirements) address space represented by light green blocks, 3) Pure low-power address space represented by squared blocks, and 4) Low power and fault-tolerant memory space represented by the dark red blocks. *To the best of our knowledge, this is the first piece of work to allow such partitioning of the address space in a transparent manner while exploiting variability across various levels of the memory hierarchy.* Due to limited space we will give a high-level overview of the various *VaMV* concepts, for more details please refer to [21].

3.2. Target Platform

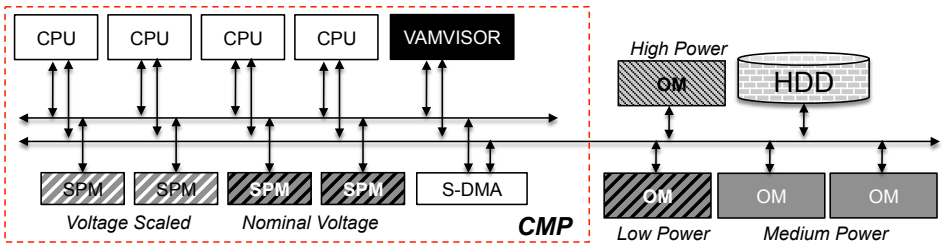

Fig. 4. Chip-Multiprocessor with Distributed Memories.

Figure 4 shows our target Chip-Multiprocessor platform ([22; 13]), which consists of a series of RISC-like processing cores, a set of on-chip *distributed* ScratchPad Memories, a secure DMA (S-DMA) engine to protect on-chip and off-chip memory space, a Crypto engine to protect memory transactions (not used in this work), a series of *distributed* off-chip memories (OMs), and a set of hard drives (HDDs). The black box represents the *VaMVisor*, a hardware module to help realize our variability-aware virtualization layer and efficiently manage the on-chip resources.

3.3. Assumptions

We assume variability in both on-chip and off-chip memories (denoted by different colored SPMs and OMs in Figure 4)). We assume that we can selectively voltage scale our on-chip memories and can lock part of off-chip memory space (via S-DMA) [23] to assist with the virtualization of the on-chip memory space, as a result, a subset of on-chip memories may have lower power consumption, higher access latencies and higher error rates than others. Furthermore, we assume that there is power consumption variability in off-chip memory [18]. We assume every application running on the system uses *some* if not *all* of the on-chip memory space, is able to partition their address space into regions with different requirements (power, performance, etc.) and on a context switch, the application's data would have to be flushed (to update mapping tables as in [24; 25]).

3.4. Embedded RAIDs (E-RAIDs)

The concept of Embedded RAIDs-on-Chip was first introduced in [13; 26], where the authors proposed a series of custom RAID-like policies (levels) that exploit aggressive voltage scaling to reduce power consumption (static and dynamic) while guaranteeing data correctness. They further showed that E-RAID levels can complement traditional schemes (e.g., ECC) to enhance the fault tolerance of the memory hierarchy. Due to limited space, we will briefly summarize a few E-RAID levels; for more information please refer to [13; 26].

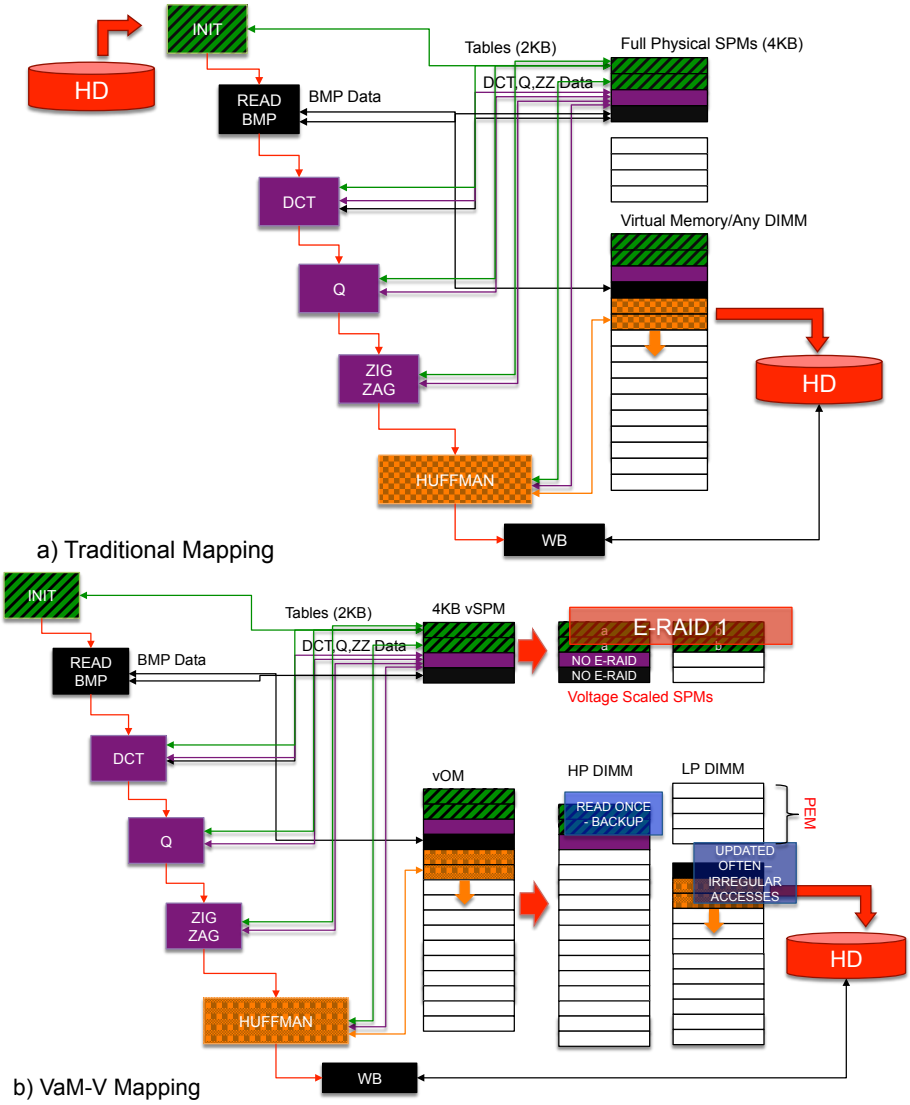

Fig. 5. Partitioning the Application's Memory Space

3.4.1. E-RAID Level 1. Like traditional RAID 1, also referred to as mirroring, two copies of each block are kept in the E-RAID, where each block being a 32bit word. Algorithm 1 shows a sample read/write policy for E-RAID 1, on a read request, the two data blocks are fetched, compared and the data is returned if correct, else the manager returns an error (Lines 2-6). The methods $DSPAM_x()$ and $DSPAM_Y()$ perform the address translation for the transaction and are used to fetched/write/update the copies of the data in their respective memory regions (Lines 2-3). On an error the master will be forced to fetch the data from off-chip memory, thereby paying the penalty of a main memory access (Line 8). E-RAID 1 achieves lower power consumption and lower performance overheads than parity checking schemes as the comparison of the two blocks requires a simple *AND* or *XOR* and the reads/writes can be done in parallel. This level assumes

Algorithm 1: E-RAID Level 1 Read/Write Policy

Require: $REQ\{CTRL, ADDR, DATA^*\}$
if $REQ.CTRL == READ\_CTRL$ **then**
  $A1 \leftarrow DSPAM_x(REQ.ADDR)$
  $A2 \leftarrow DSPAM_y(REQ.ADDR)$
  if $A1 == A2$ **then**
    $REQ.DATA^* \leftarrow A1$
    return $CHANNEL\_OK$
  else
    return SLV_ERR
  end if
else
  if $REQ.CTRL == WRITE\_CTRL$ **then**
    $DSPAM_x(REQ.ADDR) \leftarrow REQ.DATA^*$
    $DSPAM_y(REQ.ADDR) \leftarrow REQ.DATA^*$
  end if
end if

that the probability that two blocks have an error at the same bit location is low [27; 28], however, unlike traditional replication approaches (e.g., [27; 28; 29; 30]), E-RAIDs do not assume that the backup data is correct, thereby provide higher data-correctness guarantees.

3.4.2. E-RAID Level 1 + ECC. Like E-RAID 1, this level keeps two copies of the data, one backup and one original (protected by SEC). The idea is to minimize ECC overheads by comparing the two copies before the ECC check is done since a simple comparison incurs less overhead than the ECC check/correction, while minimizing off-chip accesses on an error detection (unlike E-RAID 1).

3.4.3. NO E-RAID. The NO E-RAID level consists of pure voltage scaled memory space with no reliability guarantee.

Because each E-RAID level has different power, performance, and fault-tolerance guarantees, it is possible to mix E-RAID levels to better utilize the on-chip memory space while satisfying an application's needs. For example, we can take a multimedia application which consists of non-critical (e.g., pixel) data and critical data (e.g., look up tables) an apply NO-ERAID and E-RAID 1+ECC levels respectively.

3.5. virtual ScratchPad Memories (vSPMs)

The concept of ScratchPad Memory virtualization was first introduced by [13] and later refined by [23] through the creation of virtual ScratchPad Memories (vSPMs). The idea was motivated in part by the need to provide access to on-chip memory space in a transparent manner (in the presence of multi-tasking environments). Unlike traditional SPM management approaches, which assumed an application had full control of the on-chip memory space, vSPMs allowed designers to still use their traditional SPM memory management schemes while not having to worry about their data being tampered with (evicted, modified, etc.). vSPMs are realized by locking part of off-chip memory space (Protected Evict Memory (PEM)) in order to $extend$ the available on-chip memory space. They defined priority-based allocation policies to efficiently used the on-chip memory space while minimizing both power and performance overheads of their virtualization layer.

3.6. Variability-aware Data Partitioning

Figure 5 shows the address space partitioning of the JPEG [31; 32] application. Lee et al. [33] proposed the idea of partitioning data into critical and non-critical data, and

based on this partition they decided whether to map the data to their protected cache (through ECC) or to traditional cache space. Note that they protected data against soft-errors, in our case, we propose a similar partitioning, however, our goal is to split data into various regions, where each region can support a *policy*. In our work, each data block is associated with a *policy* which dictates how to map the block into physical address space and the type of guarantees needed (power, performance, fault-tolerance). Figure 5 shows four separate partitions: 1) Read-only and highly utilized data (e.g., look up tables) represented by the dashed green blocks, 2) A temporary buffer space to hold inter-task communication denoted by the light purple block, 3) Read-only image data pixels which is represented by the black blocks, 4) Irregular frequently accessed address space denoted by the squared orange blocks. Figure 5 (a) shows a traditional mapping of these data blocks, where no notion of variability is taken into account. Figure 5 (b) shows our approach, where we have the *same* address space and same partitions, but we exploit the information provided by the compiler/programmer (referred to as *policies*) to help our *VaMVisor* make the right mapping decisions exploiting the device's present memory variability. Figure 5 (b) maps commonly used read-only data onto low-power memory space protected by an E-RAID 1 level, pixel data to low power memory space (NO ERAID), and irregular and commonly used data to off-chip low power memory.

Custom variability-aware *policy* generation (e.g., data mapping, type of protection) is a promising field of research as the compiler can derive so much more useful information from the application (than a knowledgeable programmer), and as a result, further studies are left for future work.

3.7. Virtual SPM and Virtual Off-chip Memory

In this paper we exploit the notion of virtual SPMs (vSPMs) and extend it with the notion of virtual Off-chip Memory (vOM) to fully virtualize the memory space. The idea is to provide a transparent means to manage the memory space without programmers having to worry about where their data is mapped. Unlike traditional memory virtualization schemes, our virtualization layer allows programmers to partition their virtual memory into regions (within virtual memories - vSPMs or vOMs) and define polices for each region requiring different guarantees (power, performance, fault-tolerance).

Method	Notes
v_mem_create(**uint** PID, **uint** AppPriority, **uint*** IPA, **uint** ACL, **uint** MemType)	Process ID (PID), Application Priority (AppPriority), Intermediate Physical Address (IPA), Access Control List (ACL), MemoryType: vSPM or vOM
v_mem_delete(**uint** PID, **uint** IPA)	Delete a memory at a given IPA
v_blk_malloc(**uint** PID, **uint** IPA, **uint** BlkSize, uint BlkPriority, **uint** MallocPolicy, **uint** BlkACL)	Block Size in Bytes (BlkSize), Priority of this block, (BlkPriority), Policy associated with block (MallocPolicy), Block Access Control List (BlkACL)
v_blk_delete(**uint** PID, **uint** IPA)	Delete a single block
v_blk_poll(**uint** PID, **uint** IPA)	Poll allocation status of a given block
v_mem_transfer(**uint** PID, **uint** SrcAddr, **uint** DstAddr, **uint** TxtType)	Transfer data between source address (SrcAddr) and destination address (DstAddr), and make transaction sync/async/secure (TxtType).

Fig. 6. *VaMV* Management API.

Figure 6 shows a subset of our supported API. The goal is to provide programmers with a simplified API that would allow them to take advantage of our virtualization layer without having to fully change their programming model. The key methods are: 1) *v_mem_create()*, which allows designers to define a virtual memory, and obtain an intermediate physical address (IPA) from the method, which is then used as an offset address to the given virtual memory. The ACL field allows designers to specify the type

of access control for the memory (can specify block level ACL too), which is then used to grant access to the virtual memory to only authorized processes. 2) The *v_blk_malloc()* method allows designers to allocate a block of data within a given virtual memory (vSPM/vOM) at the given IPA, along with a block level ACL (used for sharing/protecting the block), and a block priority (e.g., high-utilization) and an allocation *policy*. The policy determines what degree of protection to use (e.g., E-RAID level to use) and where to map the data (e.g., low power memory space). Finally, 3) The *v_blk_poll(PID, IPA)*, which allows programmers to poll the *VaMVisor* for the allocation status of a given block.

```
void i_zig_zag (int * omatrix) {                                    1
  int i,  *imatrix, *zz;                                            2
  unsigned int m_offset = SPMBASEADDR;                              3
  m_lock = 0;  // point zig zag matrix to SPM                       4
  status = init_dma_put(get_pid(),                                  5
    &zigzag_idx, m_offset);                                         6
  wait_dma_complete(&m_lock);                                       7
  zz = m_offset;                                                    8
  m_lock = 0; // point input matrix to SPM                          9
  status = init_dma_put(get_pid(),                                  10
    &input_matrix, m_offset +64*sizeof(int));                       11
  wait_dma_complete(&m_lock);                                       12
  imatrix = m_offset +64*sizeof(int);                               13
  // point omatrix to main memory                                   14
  omatrix = (int*) malloc (64*sizeof (int) );                       15
  for (i = 0; i < DCTSIZE2; i++)                                    16
  *(omatrix++) = *(imatrix + (zz + i ));                            17
  ...                                                               18
```

Function 1: Traditional programming model for SPM based systems

```
void i_zig_zag (int * omatrix) { // VaMV enabled                    1
  int i,  *imatrix, *zz;                                            2
  unsigned int m_offset, m_offset_off;                              3
  // create v spm                                                   4
  v_mem_create(get_pid(), MIN_PRIO,                                 5
    &m_offset, ((get_pid() << 6) | RW_ACL), V_SPM);                 6
  // block allocation w/min priority and same acl as vSPM           7
  v_blk_malloc(get_pid(), m_offset,                                 8
    256, MAX_PRIO, LP_ERAID1, V_SPM_D);                             9
  // create v off-chip memory - usually done once                   10
  v_mem_create(get_pid(), MIN_PRIO,                                 11
    &m_offset, ((get_pid() << 6) | RW_ACL), V_OM);                  12
 // block allocation w/min priority and same acl as vOM             13
  v_blk_malloc(get_pid(), m_offset_off,                             14
    256, MIN_PRIO, LP_NOERAID, V_OM_D);                             15
  // point zig zag matrix to vSPM                                   16
  m_lock = 0;                                                       17
  status = init_dma_put(get_pid(),                                  18
      &zigzag_idx, m_offset);                                       19
  wait_dma_complete(&m_lock);                                       20
  zz = m_offset;                                                    21
  m_lock = 0; // point input matrix to vSPM                         22
  status = init_dma_put(get_pid(),                                  23
      &input_matrix, m_offset +64*sizeof(int));                     24
  wait_dma_complete(&m_lock);                                       25
  imatrix = m_offset +64*sizeof(int);                               26
  // point omatrix vOM (irregular accesses by huffman)              27
  omatrix = m_offset_off;                                           28
  for (i = 0; i < DCTSIZE2; i++)                                    29
    *(omatrix++) = *(imatrix + (zz + i ));                          30
  ...                                                               31
```

Function 2: *VaMV* programming model

Function 1 shows the traditional programming model considering a traditional memory hierarchy (SPM and Off-chip Memory), and Function 2 shows our VaMV-aware programming model. This example shows the $i\_zig\_zag()$ method from DJPEG adapted to use SPMs. The key differences are: 1) Line 3 has two temporary variables to hold the IPAs needed to access the vSPM and vOM needed to execute this method, unlike Function 1, we do not need to keep track of absolute physical SPM addresses. 2) Lines 5-16 show the creation of the vSPM and vOM as well as their block allocations. Note that vSPMs and vOMs should be the first thing to be created, where the programmer can create memory regions within the vSPM and vOM with different requirements, i.e., low power E-RAID 1 space (LP_ERAID1) to protect the zig-zag

look up table, and low-power off-chip memory space ($LP\_NOERAID$) to hold highly-utilized irregularly accessed data (used by the i_huffman task). Disregarding the extra comments, the number of extra lines introduced into the method to take advantage of our virtualization layer is minimal. *To the best of our knowledge, no existing memory management layer exists to exploit variability*, so the traditional malloc() allocates memory space in a random off-chip memory, and does not take advantages of the variability in off-chip memory. 3) The allocation methods allow a programmer to define access controls to protect the memory space from unauthorized accesses.

3.8. *VaMVisor*: Dynamic Memory Management

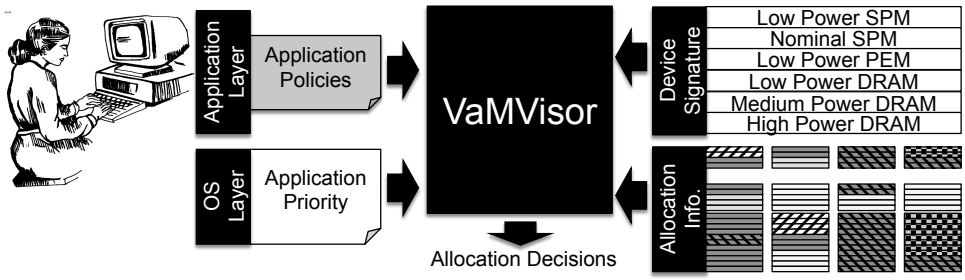

Fig. 7. Variability-aware Memory Virtualization.

Once a programmer/designer has partitioned the memory space and derived policies for each region, the next step is to provide an infrastructure to make real-time memory management decisions. In this paper, we propose the *VaMVisor*, a hardware module that can be embedded in today's bus-based Chip-Multiprocessors as an augmented arbiter similarly to [13; 23; 34; 35]. Figure 7 shows a high level view of our *VaMVisor*-asisted run-time environment. The *VaMVisor* exploits the notion of policy-driven variability-aware allocation to efficiently manage the on-chip and off-chip resources. On memory block allocation, the shows the *VaMVisor* takes in the policy associated with the data block, the application's privilege level/priority, the signature of the device (characteristics), and the system load. There are three key components here: 1) As discussed in Section 3.6, the type of policy associated with each block will determine how the block is mapped, 2) 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), 3) The signature of the device, which is device dependent as variability is random in nature. This signature allows our run-time environment to opportunistically exploit the variability present in the device, 4) The status of the system, which is useful when deciding how to allocate data blocks. A combination of these four parameters allows the *VaMVisor* to make real-time allocation decisions, which in some cases will lead to re-arrangement of data blocks to accommodate tasks with high priorities.

Figure 8 shows a set of applications being executed (App1-App4) by two CPUs (CPU0, CPU1) utilizing a total of two SPMs (4KB space each) and the status of the memories as vSPMs are created, and blocks are allocated (States S1 through S6). On arrival of the first application (App3), the vSPM is created and the SPMVisor maps the App3's blocks to SPM0, and the process continues up until S3. When App4 arrives, the SPMVisor looks at the priorities of the blocks belonging to App4 and decides to map them to PEM space. When App5 (red dotted block) needs to execute, rather than evicting all of App1 and App2's contents from SPM (as in traditional approaches [36]),

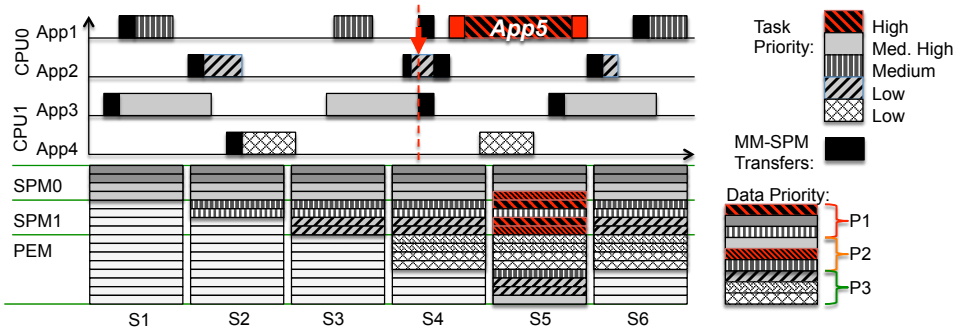

Fig. 8. Multi-tasking Policy-driven Variability-aware Allocation.

the *VaMVisor* looks at the priorities of the various blocks, and makes the decision to evict *some* of the lower priority blocks from SPM space (App1-3), and allocating the space to App5 blocks as shown in S5. After App5 completes and destroys the vSPM, the *VaMVisor* then re-loads the contents it had evicted prior to App5's execution. This example shows how our data-driven allocation policy works, as blocks may have different priorities (P1-P3) and its possible that applications with lower priority may have higher priority blocks than applications with higher priority. Moreover, the *VaMVisor* also partitions the off-chip memory (OM) space and prioritizes to exploit the variability present in the device. An example is to exploit an utilization-based priority policy, where two application's request virtual off-chip memory (vOM) space with the same priority (but utilization is higher for one than the other), so the *VaMVisor* would try to map the vOM to the physical OM with the lowest power consumption. If it cannot serve both request, 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.

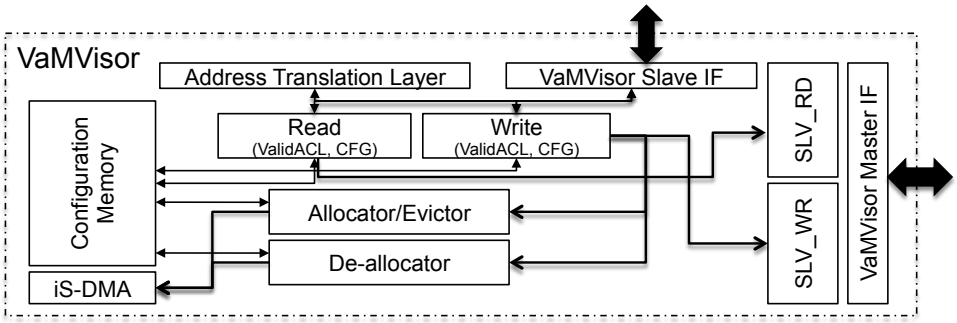

Fig. 9. *VaMVisor* Architecture.

Figure 9 shows a high-level block diagram of our *VaMVisor*, which includes a configuration memory that holds the metadata (which can be between 512B and 2KB) for vSPMs/vOMs and their blocks. Each block metadata requires up to 12 Bytes, up to 2 physical addresses and 4Bytes for control (ACL, policy, etc.), and is stored in VaMVisor's configuration memory. An address translation unit for quick IPA-PA translation. *VaMVisor* provides a vSPM address space of (2^{14}); however, the number of vSPMs is limited by the block size used (can be 256B, 512B, 1024B, 4KB, etc.), the total amount

of physical memory managed (# SPMs, # OMs, and PEM space), and the amount configuration memory storage. The Allocator/Evictor unit is responsible for making the allocation-decisions at run-time (as discussed earlier in the section).

4. RELATED WORK

Intra/Inter-Die variation has been explored in many contexts (processors, memories, integrated circuits, etc.). Borkar et al. [4] measured die-to-die V_t distribution and its resulting chip I_{sb} variation (about 30mV in a 180nm CMOS logic technology), which causes a significant variation in circuit performance and leakage as well as frequency variation across dies. Hanson et al. [19] measured the power consumption across five identical (same specifications and same workload) Intel M processor chips and found between 3% and 10% variation. Wanner et al. [3] measured the sleep power at room temperature for ten instances of Cortex M3 based Atmel SAM3U processor and observed more than 5x variation. Their work served as the basis for a variability-aware duty cycle scheduling scheme which showed further improvements energy efficiency [6]. Sartori et al. [2] looked at frequency variation across various processing cores. Pant et al. [5] proposed hardware signatures to adapt the software stack to deal with variability in the underlying hardware. Hanson et al. [19] looked at power consumption across five 512 MB DDR2 DRAM memories (different vendors) and observed up to 2x in active power variation. Gottscho et al. [18] modified Memtest86 and measured power (IDLE, READ, WRITE) for a series of DDR3 memories and observed up to 16.77% power variation across memories belonging to the same vendor and 14.58% across various vendors (same size DIMMs).

Exploiting voltage scaling to reduce on-chip memory power consumption has been explored primarily in the cache-architecture domain. Makhzan et al. [8] propose the idea of exploiting error maps to correct faulty cells on the main cache. Chakraborty et al. [11] exploit the idea of in-cache replication to reduce energy. Sasan et al. [9] proposed changes at the architectural level (SRAM) to tolerate errors due to voltage scaling, which was followed by a resizable data composer-cache architecture to operate at sub 500mV [7]. Kurdahi et al. [20] proposed an algorithmic solution (e.g., at the application level) to handle process variations in the memory subsystem. Mutyam et al. [37] proposed the concept of block rearrangement to minimize performance loss incurred by process variations on a cache. Liang et al. [38] proposed replacing 6T SRAM with 3T1D DRAM for data-caches to address physical device variation by exploiting cache refresh and placement schemes to deal with retention time variations. Meng et al. [39] proposed way prioritization to minimizes cache leakage in the presence of within-die leakage variation. At the system level, Bathen et al. [13] proposed the concept of Embedded RAIDs-on-Chip, which exploit voltage scaling to reduce power consumption.

Shalan et al. [35] looked at dynamic memory management for global memory through the use a hardware module. Francesco et al. [34] proposed a memory manager that supports dynamic allocation of SPM space, which supports block-based allocation (fixed and variable). Egger et al. proposed SPM management techniques for MMU supported [24] and MMU-less embedded systems [25], where code was divided into cacheable code and pageable (SPM) code, and the most commonly used code is mapped onto SPM space. [40] introduced an OS-level management layer that exploited hints from static analysis at run-time to dynamically map objects onto SPMs.

Our scheme is different from [3; 19; 2] in that we focus primarily in memory variability, however, our scheme could potentially be complemented by other schemes that consider processor variability (e.g., frequency, power consumption, etc.). Our approach is different from [8; 11; 9; 20; 7] in that we selectively voltage scale on-chip $distributed$ memories and take advantage of the variation (power, performance, error rates) opportunistically at the system level. Our approach differs from [13] in that our scheme goes

beyond using E-RAIDs on-chip, we also exploit off-chip memory variability, and fully virtualize the *entire* memory space. Our approach differs from [35; 34; 25; 40; 24] in that our approach exploits variability across the entire memory hierarchy, moreover, our *VaMVisor* exploits the idea of priority-driven variability-aware memory allocation. *To the best of our knowledge, we are the first to propose the idea of exploiting variability in on-chip and off-chip memory to reduce power consumption.*

5. EXPERIMENTAL EVALUATION AND RESULTS

5.1. Experimental Evaluation Goals

The goal is to show how we can exploiting and co-optimizing on-chip and off-chip memory variability in a holistic manner to reduce power consumption. First, we will partition the address space of each application to exploit on-chip/off-chip variability. We will show how E-RAIDs can be complemented by exploiting off-chip memory variability to reduce power consumption. Second, we explore the effects of variability in memory virtualization. Third, we explore various policies to illustrate how choosing the right policy will lead to power-consumption saving. Finally, we show the true potential of exploiting the notion of variability-aware *dynamic* policy-driven memory allocation for *distributed* on-chip and off-chip memories.

5.2. Experimental Setup

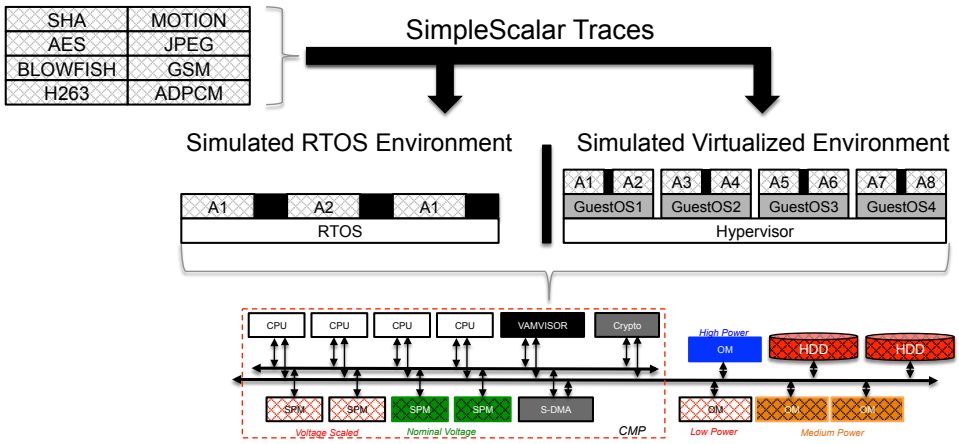

Fig. 10. Experimental Setup.

Figure 10 shows our simulation environment (implemented in SystemC TLM/C-CATB [41] and interfaces with Simplescalar [42] and CACTI [43]), where we can simulate a Chip-Multiprocessor with distributed on-chip SPMs, a set of distributed off-chip memories (OMs), and our *VaMVisor*. We assume 65nm process technology for our memories and a 1GB off-chip main memories. We cross-compiled a set of applications (ADPCM, AES, BLOWFISH, GSM, H.263, JPEG, MOTION, and SHA) from the CHStone [32] and Mediabench II [44] benchmark suites and analyzed them to obtain SPM mappable data sets. In order to support multiple applications running concurrently we used page tables (1KB mini-pages). The application's virtual addresses are translated by the CPU's MMU unit and generates physical addresses which point to physical SPMs, or intermediate physical addresses (IPAs) which then point to vSPMs.

Our environment can simulate a lightweight RTOS environment with context switching enabled and a light-weight hypervisor. Each OS/CPU instance can run anywhere between 1-4 simulated OSes and 1-8 applications. We chose four off-chip memories from (OM{1-4}) [18].

Table I: Sample Policies

Data Type	Desctiption
T1	Look-up tables (e.g., quantization variables, ziz-zag indices)
T2	Commonly used data (e.g., inter-task communication buffers)
T3	Non-Critical Data (e.g., pixels)
T4	Variables (e.g., frame width/height)
T5	All other
Policy	**Desctiption**
P1	*physical SPM (pSPM)* $\leftarrow$ *{T1,T2,T4}, off-chip memory (OM) 1* $\leftarrow$ *T5*
P2	*pSPM* $\leftarrow$ *{T1,T2,T4}, OM2* $\leftarrow$ *T5*
P3	*pSPM* $\leftarrow$ *{T1,T2,T4}, OM3* $\leftarrow$ *T5*
P4	*pSPM* $\leftarrow$ *{T1,T2,T4}, OM4* $\leftarrow$ *T5*
PLP	*pSPM* $\leftarrow$ *{T1,T2,T4}, low-power (LP) OM* $\leftarrow$ *T5*
PHP	*pSPM* $\leftarrow$ *{T1,T2,T4}, high-power (HP) OM* $\leftarrow$ *T5*
E1	*virtual SPM (vSPM)/E-RAID1/0.35V* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
E2	*vSPM/E-RAID1/0.35V* $\leftarrow$ *{T1,T2,T4}, LP OM* $\leftarrow$ *T5*
E3	*vSPM/E-RAID1+ECC/0.35V* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
E4	*vSPM/E-RAID1+ECC/0.35V* $\leftarrow$ *{T1,T2,T4}, LP OM* $\leftarrow$ *T5*
E5/VE1	*vSPM/E-RAID1/0.5V* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
E6	*vSPM/E-RAID1/0.5V* $\leftarrow$ *{T1,T2,T4}, LP OM* $\leftarrow$ *T5*
E7	*vSPM/E-RAID1+ECC/0.5V* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
E8	*vSPM/E-RAID1+ECC/0.5V* $\leftarrow$ *{T1,T2,T4}, LP OM* $\leftarrow$ *T5*
V1	*vSPM/Nominal Vdd* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
V2	*vSPM/Nominal Vdd/HP PEM* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
V3	*vSPM/Nominal Vdd/LP PEM* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
V4	*vSPM/Nominal Vdd/LP PEM* $\leftarrow$ *{T1,T2,T4}, LP virtual OM (vOM)* $\leftarrow$ *T5*
VE2	*vSPM/E-RAID1/0.5V* $\leftarrow$ *{T1,T2,T4}, LP vOM* $\leftarrow$ *T5*
VE3	*vSPM/E-RAID1/0.5V* $\leftarrow$ *{T1,T4}, vSPM/NO-ERAID/0.5V* $\leftarrow$ *{T2,T3}, HP OM* $\leftarrow$ *T5*
VE4	*vSPM/E-RAID1/0.5V* $\leftarrow$ *{T1,T4}, vSPM/NO-ERAID/0.5V* $\leftarrow$ *{T2,T3}, LP vOM* $\leftarrow$ *T5*
VE5	*2×vSPMs/E-RAID1/0.5V* $\leftarrow$ *{T1,T4}, 2×vSPMs/NO-ERAID/0.5V* $\leftarrow$ *{T2,T3}, LP vOM/E-RAID1* $\leftarrow$ *T5*
M1	*8×vSPMs/LP PEM* $\leftarrow$ *{T1,T2,T4}, HP OM* $\leftarrow$ *T5*
M2	*8×vSPMs: E-RAID1* $\leftarrow$ *{T1,T4}, 3×NO-ERAID & 4×LP PEM* $\leftarrow$ *{T2,T3}; HP OM* $\leftarrow$ *T5*
M3*	*8×vSPMs: E-RAID1* $\leftarrow$ *{T1,T4}, 3×NO-ERAID & 4×LP PEM* $\leftarrow$ *{T2,T3}; LP vOM* $\leftarrow$ *T5*

Table I shows the policies we will use during this section. The base-line policies start with P, the E-RAID policies start with E, the vSPM policies start with V, the hybrid (vSPMs+E-RAID) policies start with VE, and the hybrid policies that support multi-tasking start with M.

5.3. Exploiting DRAM Variability

For exploiting DRAM variability, we simulated a single application running on the system with 4x1GB DRAMs exhibiting variability from the same vendor/specs [18] and no data cache/SPMs because we wanted to emphasize the importance of exploiting DRAM power variability. 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 (*Random-Malloc*). Figure 11 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.

5.4. Exploiting On-Chip and Off-Chip Memory Variability

In this experiment we evaluate the effects of off-chip power consumption variability in the application's power consumption: 1) without voltage scaling the on-chip memo-

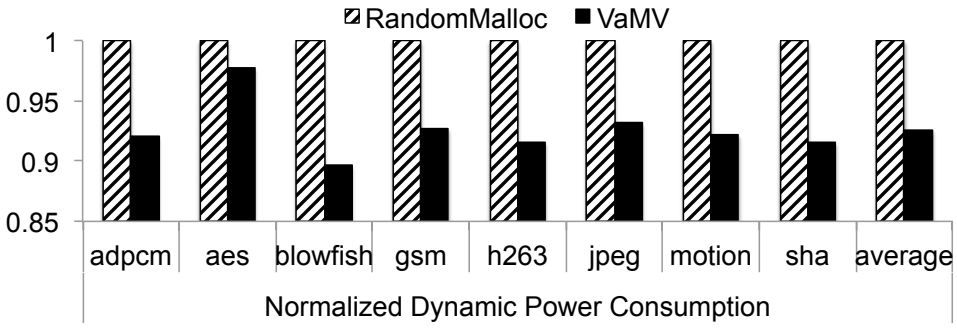

Fig. 11. Exploiting DRAM Variability.

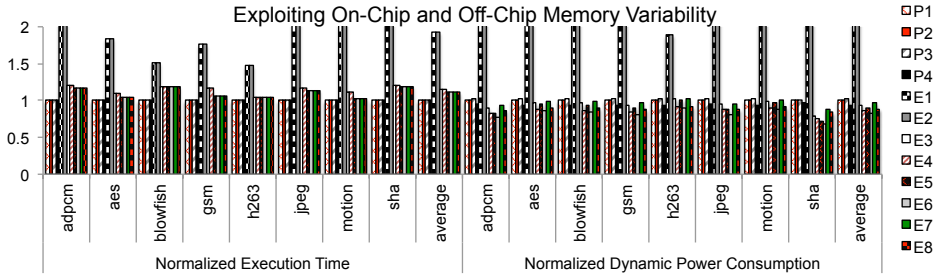

Fig. 12. Exploiting On-Chip and Off-Chip Memory Variability.

ries (P policies) and 2) with E-RAIDs (E policies with voltage scaled memories (350mV and 500mV [45]), using a high-power (HP) OM3 (worst case power consumption) and low-power (LP) OM4 (best case power consumption) as backup memory. We run a single application on a CPU with access to $2 \times 8KB$ SPMs. Our goal is to show how off-chip memory variability can complement E-RAIDs (on-chip memory variability). As expected, at ultra low Vdd (350mV), Policy $E1$ incurs orders of magnitude higher power consumption and performance overheads than all other schemes (due to the large amount of errors detected leading to extra off-chip memory accesses). As shown in Figure 12, we observe that solely exploiting off-chip variability leads to an average 5.5% power consumption reduction (P policies). Policy $E8$ shows an average 18% reduction in power consumption with an average 10% performance overheads. Moreover, in some cases (*E8/SHA*) we see up to 32% reduction in dynamic power consumption. This experiment shows that complementing E-RAIDs with off-chip memory variability can lead to promising results (reduced power consumption with minimal performance overheads).

5.5. Effects of Variability in Memory Virtualization

Figure 13 shows the effects of variability in memory virtualization. We run a single application on a CPU with access to $2 \times 8KB$ SPMs. The goal is to observe the overheads of our virtualization layer against the best case scenarios (P policies), where programmers can use the entire on-chip memory space and partition their data to exploit off-the-shelf hardware variability. On average, the $P4$ policy consumes less power than all other schemes because it uses the off-chip memory with lowest power consumption. We see that in gradually partitioning the virtual address space into low-power regions

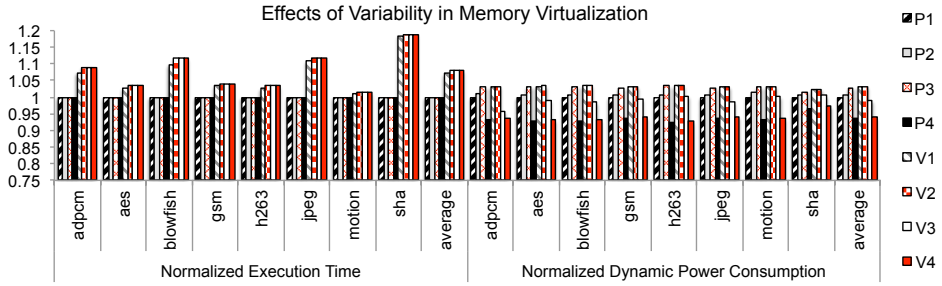

Fig. 13. Effects of Variability in Memory Virtualization.

progressively leads to lower power consumption (e.g., policies $V1 \rightarrow V4$). On average, our policies (V) achieve less than 8% overheads due to our virtualization layer and are within 2.8% of the best case power consumption ($P4$).

5.6. Custom Variability-aware Policy Generation

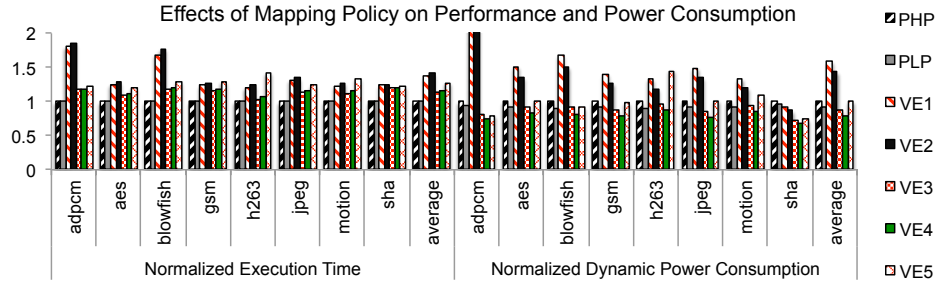

Fig. 14. Effects of Mapping Policy on Performance and Power Consumption.

Figure 14 shows the effects on performance and power consumption for a series of partitioning policies. Each policy determines how to partition the address space of a given application. For this experiment, we progressively augmented the complexity of our policies ($VE1 \rightarrow VE5$), and compare their power consumption/performance overheads to the bast-case/worst-case policies (PLP and PHP). The first two custom policies ($VE1/VE2$) incur high overheads in both power and performance primarily because we utilized the entire on-chip memory space for the E-RAID 1 level (we use half of the available space). 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 PLP). We observe up to 33% dynamic power consumption reduction for the $VE4$ policy. Memory intensive applications such as H.263 benefit the most from vOM-based policies (e.g., $VE4$), we observe up to 14% power consumption reduction for H.263 with minimal performance overheads (0.06%). The $VE5$ policy has higher performance overheads than the $VE4$ policy because of the off-chip memory protection (E-RAID 1 on top of vOM), but we observe less than 8% higher power consumption than the best base case policy (PLP). These experiments show that carefully

crafting variability-aware policies to meet an application's needs leads to reduced dynamic power consumption.

5.7. Dynamic Policy-driven Variability-aware Allocation

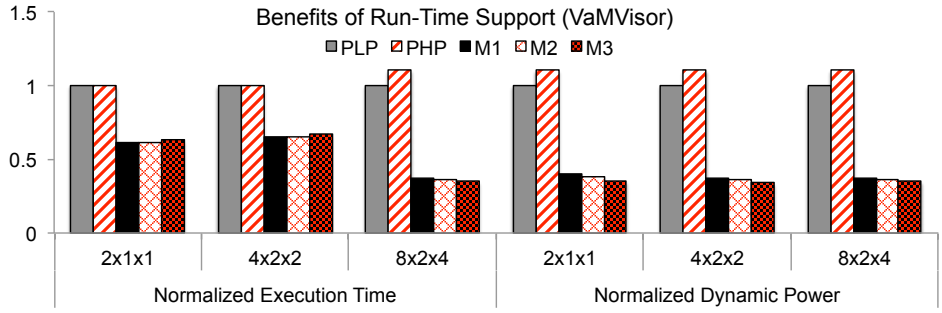

Fig. 15. Dynamic Policy-driven Variability-aware Allocation.

To show the true benefits of our approach, we simulated a series of virtualization environments running various applications across a CMP. Figure 15 shows on the $x-axis$ various configurations in the form: $\{\#Apps\}\times\{\#OSes\}\times\{\#CPUs\}$, each CMP configuration consisted of 4x8KB SPMs, with the base cases (PLP/PHP policies) utilizing the entire physical space with context-switching (CX) enabled to prevent data corruption. We assume the Application/OS context-switch costs similar to [**?**]. The remaining policies ($M1-3$) exploited vSPMs, $M3*$ refers to the last test case (8x2x4, where we created 16 vSPMs). We see that PHP incurs an average10% increase in power consumption (with respect to PLP). In contrast, our *VaMVisor*-supported policies ($M1-M3$) managed to reduce dynamic power consumption by 63% on average while reducing total execution time by an average of 45%. Be observe higher benefits in this experiment because we have up to 8 applications, 4 OSes, and 4 CPUs competing for memory resources, and thus, a traditional data-mapping approach will not be able to handle the demand. This experiment shows the true potential of exploiting the notion of variability-aware $dynamic$ policy-driven memory allocation for $distributed$ on-chip and off-chip memories.

6. CONCLUSION

This paper proposed a *first-of-its-kind* Hardware-assisted Variability-aware Memory Virtualization (*VaMV*) layer that allows programmers/applications to partition their address space into regions with different power, performance, and fault-tolerance guarantees. *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. *To the best of our knowledge, we are the first to explore the notion of variability-aware policy-driven memory allocation for* $distributed$ *on-chip and off-chip memories*. We propose a *proof-of-concept* hardware-module called *VaMVisor*, which allows us to minimize the overheads incurred by virtualization and dynamic allocation of the memory space. Finally, we define an API to facilitate the creation and management of virtual ScratchPad Memories (vSPMs) and virtual Off-chip Memories (vOMs). Our experimental results on a set of benchmarks (Mediabench I/II and CHStone) show that our approach is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 45%.

Since this is the first piece of work exploring variability in *distributed* on-chip and off-chip memories, we believe that there are many future directions for this work: 1) Studying variability across the *entire* memory hierarchy (e.g., caches, disks), 2) Exploit other types of variability (e.g., processor frequency, power consumption), and 3) Further explore the power of application-driven (custom) variability-aware policy generation (e.g., how to partition the memory space, what degree of protection to use, etc.).

REFERENCES

ITRS, "Process integration, devices and structures," *http://www.itrs.net/*, 2007.

J. Sartori, A. Pant, R. Kumar, and P. Gupta, "Variation-aware speed binning of multi-core processors," in *ISQED*, 2010, pp. 307–314.

L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, "A case for opportunistic embedded sensing in presence of hardware power variability," in *Proceedings of the 2010 international conference on Power aware computing and systems*, ser. HotPower'10, 2010, pp. 1–8.

S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th annual Design Automation Conference*, ser. DAC '03, 2003, pp. 338–342.

A. Pant, P. Gupta, and M. van der Schaar, "Software adaptation in quality sensitive applications to deal with hardware variability," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, ser. GLSVLSI '10, 2010, pp. 85–90.

L. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, and M. B. Srivastava, "Variability-aware duty cycle scheduling in long running embedded sensing systems," in *DATE*, 2011, pp. 131–136.

A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "A fault tolerant cache architecture for sub 500mv operation: resizable data composer cache (rdc-cache)," in *Proceedings of the 2009 Int. Conf. on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '09, 2009, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629431

M. Makhzan, A. Khajeh, A. Eltawil, and F. Kurdahi, "Limits on voltage scaling for caches utilizing fault tolerant techniques," in *Computer Design, 2007. ICCD 2007. 25th Int. Conf. on*, oct. 2007, pp. 488 –495.

A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "Process variation aware sram/cache for aggressive voltage-frequency scaling," in *Proceedings of the Conf. on Design, Automation and Test in Europe*, ser. DATE '09, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1874620.1874845

S. Nassif, "Modeling and analysis of manufacturing variations," in *Custom Integrated Circuits, 2001, IEEE Conf. on.*, 2001, pp. 223 –228.

A. Chakraborty, H. Homayoun, A. Khajeh, N. Dutt, A. Eltawil, and F. Kurdahi, "$e<mc^2$: less energy through multi-copy cache," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES '10, 2010, pp. 237–246. [Online]. Available: http://doi.acm.org/10.1145/1878921.1878956

A. K. Djahromi, A. M. Eltawil, F. J. Kurdahi, and R. Kanj, "Cross layer error exploitation for aggressive voltage scaling," in *Proceedings of the 8th Int. Sym. on Quality Electronic Design*, ser. ISQED '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 192–197. [Online]. Available: http://dx.doi.org/10.1109/ISQED.2007.53

L. Bathen and N. Dutt, "E-RoC: Embedded raids-on-chip for low power distributed dynamically managed reliable memories," in *Design, Automation Test in Europe Conf. Exhibition (DATE), 2011*, march 2011.

IBM, "The cell project," *IBM, http://www.research.ibm.com/cell/*, 2005.

Intel, "Single-chip cloud computer," *Intel, http://techresearch.intel.com/ProjectDetails.aspx?Id=1*, 2009.

Tilera, "Tile gx family," *Tilera, http://www.tilera.com/products/processors/TILE-Gx_Family*, 2010.

R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth Int. Sym. on Hardware/software codesign*, ser. CODES '02, 2002. [Online]. Available: http://doi.acm.org/10.1145/774789.774805

M. Gottscho et al., "Analyzing power variability of ddr3 dual inline memory modules for applications," *TR UCLA EECS*, 2011.

H. Hanson et al., "Benchmarking for power and performance," *2007 SPEC Workshop*, 2007.

F. Kurdahi, A. Eltawil, K. Yi, S. Cheng, and A. Khajeh, "Low-power multimedia system design by aggressive voltage scaling," *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, vol. 18, no. 5, may 2010.

blind, "A case for an adaptive and opportunistic variability-aware memory virtualization layer," in *– Technical Report #–*, 2011.

V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *Proceedings of the 2006 Int. Conf. on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06, 2006, pp. 401–410. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176809

L. Bathen and N. Dutt, "Towards distributed on-chip memory virtualization," *UCI Center for Embedded Computer Systems TR #11-2*, 2011.

B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu," *ACM Trans. Embed. Comput. Syst.*, vol. 7, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1331331.1331335

B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad memory management techniques for code in embedded systems without an mmu," *Computers, IEEE Trans. on*, vol. 59, no. 8, 2010.

L. Bathen and N. Dutt, "Towards embedded raids-on-chip," *UCI Center for Embedded Computer Systems TR #10-12*, 2010.

W. Zhang, "Enhancing data cache reliability by the addition of a small fully-associative replication cache," in *Proceedings of the 18th annual Int. Conf. on Supercomputing*, ser. ICS '04, 2004. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006212

W. Zhang, S. Gurumurthi, M. Kandemir, and A. Sivasubramaniam, "Icr: in-cache replication for enhancing data cache reliability," in *Dependable Systems and Networks, 2003. Proceedings. 2003 Int. Conf. on*, june 2003.

F. Angiolini, D. Atienza, S. Murali, L. Benini, and G. De Micheli, "Reliability support for on-chip memories using networks-on-chip," in *Int. Conf. on Computer Design (ICCD) 2006.*, oct. 2006.

F. Li, G. Chen, M. Kandemir, and I. Kolcu, "Improving scratch-pad memory reliability through compiler-guided data block duplication," in *Proceedings of the 2005 IEEE/ACM Int. Conf. on Computer-aided design*, ser. ICCAD '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1002–1005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1129601.1129742

JPEG, "Iso standard, iso/iec 10918-1 itu-t recommendation t.81," *JPEG, http://www.jpeg.org/*, 1986.

Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE Int. Sym. on*, May 2008, pp. 1192 –1195.

K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 Int. Conf. on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06, 2006. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176810

P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An integrated hardware/-software approach for run-time scratchpad management," in *Proceedings of the 41st annual Design Automation Conf.*, ser. DAC '04, 2004. [Online]. Available: http://doi.acm.org/10.1145/996566.996634

M. Shalan and V. J. Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip," in *Proceedings of the 2000 Int. Conf. on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '00, 2000. [Online]. Available: http://doi.acm.org/10.1145/354880.354905

H. Takase, H. Tomiyama, and H. Takada, "Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems," in *Proceedings of the Conf. on Design, Automation and Test in Europe*, ser. DATE '10, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1870926.1871199

M. Mutyam and V. Narayanan, "Working with process variation aware caches," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '07, 2007, pp. 1152–1157.

X. Liang, R. Canal, G.-Y. Wei, and D. Brooks, "Process variation tolerant 3t1d-based cache architectures," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–26. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.33

K. Meng and R. Joseph, "Process variation aware cache leakage management," in *Proceedings of the 2006 international symposium on Low power electronics and design*, ser. ISLPED '06, 2006, pp. 262–267.

R. Pyka et al., "Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications," in *Proc.of the 10th Int. workshop on Software & compilers for embedded systems*, ser. SCOPES '07, 2007.

S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based communication architectures at the ccatb abstraction," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 22:1–22:32, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1331331.1331346

T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59 –67, Feb. 2002.

S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Hp labs cacti v5.3," *CACTI 5.1, TR, http://www.hpl.hp.com/ techreports/2008/ HPL-2008-20.html*, 2004.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE Int. Sym. on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335. [Online]. Available: http://portal.acm.org/citation.cfm?id=266800.266832

S. Jahinuzzaman, T. Shakir, S. Lubana, J. Shah, and M. Sachdev, "A multiword based high speed ecc scheme for low-voltage embedded srams," in *Solid-State Circuits Conf., 2008. ESSCIRC 2008. 34th European*, sept. 2008.

Received July 2010; revised November 2010; accepted March 2011