ViPZonE: OS-Level Memory Variability-Driven Physical Address Zoning for Energy Savings

Luis Angel D. Bathen, Nikil Dutt, Alex Nicolau
Donald Bren School of Information and Computer Science
University of California, Irvine
{lbathen,dutt,nicolau}@ics.uci.edu

Mark Gottscho, Puneet Gupta
Department of Electrical Engineering
University of California, Los Angeles
mgottscho@ucla.edu,puneet@ee.ucla.edu

ABSTRACT

ITRS predicts that over the next decade, hardware power variation will increase at alarming rates. As a result, designers must build software that can adapt to and exploit these variations to reduce power consumption and improve system performance. This paper presents ViPZonE, a system-level solution that opportunistically exploits DRAM power variation through physical address zoning. ViPZonE is composed of a variability-aware software stack that allows developers to indicate to the OS the expected dominant usage patterns (write or read) as well as level of utilization (high, medium, or low) through high-level APIs. ViPZonE's variability-aware page allocator, implemented in the Linux kernel, is responsible for interpreting these high-level requests for memory and transparently mapping them to physical address zones with different power consumption. Our experimental results across various configurations running PARSEC workloads show an average of 13.1% memory power consumption savings at the cost of a modest 1.03% increase in execution time over a typical Linux virtual memory allocator.

Categories and Subject Descriptors

B.3 [**Design Styles**]: Virtual Memory; D.4.2 [**Storage Management**]: Allocation/deallocation strategies

General Terms

Design, Experimentation, Management

Keywords

power; variability; memory management; DRAM

1. INTRODUCTION

Over the past decade, inter-die and intra-die process variations have become more significant [11, 8, 12]. The ITRS predicts that over the next decade, both performance and power consumption variation will increase by up to 66%, and 100%, respectively [23]. Variations can stem from semiconductor manufacturing processes, ambient conditions, device wear out, and in case of multi-sourced systems, vendors. Despite considerable hardware variability, the software stack assumes 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. For example, the device community has resorted to binning processors by operating frequencies to reduce the impact of inter-die variation. However, even with guardbanding, binning, and dynamic voltage and frequency scaling, variability is inherently present in any set of identically specified chips. Furthermore, with the emergence of multi-core technology, intra-die variation has also become an issue [21]. To minimize the overheads of guardbanding, recent efforts have shown that exploiting the inherent variation in devices [19, 42, 14, 39] yields significant improvements in both the energy-delay product and overall system performance.

Memory subsystems also suffer from power and timing variations. Off-chip DRAM memory may consume as much power as the processor in a server-class system [25, 47, 22, 49], and this problem may worsen for future many-core platforms (e.g., Tilera's TILEPro64 [41], Intel's Single Chip Cloud Computer (SCC) [20]). A recent study observed up to approximately 20% power variation in an off-the-shelf set of nineteen 1 GB DDR3 DIMMs [17]. On-chip memory designers have tried to create process variation-aware memory subsystems [32, 27, 40, 4] to address this issue, and multiple efforts have been made to minimize off-chip memory accesses via caching [43, 24, 38, 18], OS-level [50, 15, 22], and DRAM-level power management [13, 30, 22]. 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 [23] and emerging [51, 2] memory technologies (e.g., phase-change memory).

In this work, we present ViPZonE (OS-Level Memory *V*ariability-Driven *P*hysical Address *Z*oning for *E*nergy Savings), a system-level variability-aware solution that adapts to the power variability inherent in a set of DRAM memory modules. Although our approach focuses on harnessing variability in DDR3 memory at the DIMM modular level, our approach could be generalized to work at finer granularities of memory, if variability data and hardware support are available. Our experimental results across various configurations running PARSEC workloads show an average of 13.1%

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

memory power consumption savings at the cost of a modest 1.03% increase in execution time over a typical Linux virtual memory allocator, running on a simulated high-end PC platform. The key contributions of this work are as follows:

- An application annotation scheme complemented by a modified GLIBC library for power-aware memory allocations
- A power variation-aware physical address zoning and page allocation scheme implemented inside the kernel
- A study of the challenges traditional systems will face as variation in power consumption reaches 100% and beyond

To the best of our knowledge, ViPZonE is the first OS-level, pure-software, and portable solution to exploit DRAM power variation through memory zone partitioning.

This paper is organized as follows. We begin with a summary of related work and how our approach is novel in Sec. 2, followed by a brief background section discussing memory system terminology and the Linux kernel physical page allocator zoning scheme in Sec. 3. In Sec. 4, we describe our target platform and assumptions, and then move to a discussion of the ViPZonE system architecture and implementation of both the front-end and back-end. Sec. 5 discusses the results of our simulations, and we summarize this work and discuss future research in Sec. 6.

2. RELATED WORK

Most efforts dealing with variation have focused on exploiting frequency variation in processors [42, 21, 14, 39, 35] and process variations (due to voltage scaling) in on-chip memory [32, 27, 40, 4]. Hanson et al. [19] measured the power consumption across identical Intel M processors and found between 3%-10% variation and up to 2x active power variation across various DRAMs. Wanner et al. [44] found over 5x sleep power variation across various Cortex M3 processors and proposed a variability-aware duty cycle scheduling algorithm [45]. Sartori et al. [39] looked at frequency variation across processing cores. Pant et al. [35] proposed hardware signatures to adapt the software stack to deal with performance variation in the underlying hardware in the context of media applications. Pan et al. [34] has proposed a selective wordline voltage boosting for caches to manage yield under process variations. Mutyam et al. [32, 33] and Sasan et al. [40] proposed process variation-aware cache architectures (traditional and NUCA). Bennaser et al. [7] proposed a variable-cycle-latency cache architecture to mitigate the impact of process variations on access latency. Liang et al. [27] proposed replacing 6T SRAM with 3T1D DRAM for caches to address physical device variation. Zhao et al. [48] proposed cache migration schemes that utilizes fast banks while limiting the cost due to migration to address access latency variations in a 3D DRAM stacked non-uniform cache. Meng et al. [31] proposed way prioritization to minimize cache leakage to address within-die leakage variation. Li [26] proposed repairing only important bits to address process variations.

Traditional main memory management has focused primarily on minimizing accesses to main memory through smart caching schemes (hardware) or compiler/OS optimizations (software), yet none of these methods have taken memory variability into account. Sartor et al. [38] and Gu et al. [18] proposed exploiting program hints for optimal cache management to minimize off-chip memory accesses. Wang et al. [43] proposed a DSP partitioning scheme and instruction scheduler to minimize energy in multi-bank memories. Kandemir et al. [24] explored the impact of data transformations on memory bank locality. Hur et al. [22] and Felter et al. [15] exploited throttling at the memory controller level to reduce DRAM power consumption. Delaluz et al. [13] predicted the idle duration of various memory devices (at the controller level) to control the use of low power mode. Zheng et al. [49] proposed the concept of mini-banks to significantly reduce memory power consumption with minimum performance overheads (within 10%). Memory access scheduling has also been explored to increase performance and minimize power consumption [30]. Zhou et al. [50] designed a hardware monitor to track cache misses to improve system performance. Ahn et al. [1] proposed an energy efficient memory module that divides DIMMs into separate virtual entities, improving energy efficiency with minimal performance overheads. Finally, Bathen et al. [5] proposed the introduction of a hardware engine to virtualize on-chip and off-chip memory space to exploit the variation in the memory subsystem.

Our approach is different from the related works, as it specifically optimizes for inter-device variability in memory power, although it may be complemented by other memory management schemes. Compiler-level and directed-cache techniques [43, 24, 38, 18] could exploit our custom APIs (e.g., GLIBC) to define low/high DRAM power consumption zones and map their data to their preferred zones to minimize power consumption. Similarly, OS-level schemes [30, 50] could exploit our variation-aware allocator to map pages with highest cache miss ratio to low power space to minimize power consumption. Our approach can complement [5]; however, we do not need to modify existing architectures to support our physical zoning strategy (our scheme is portable and could be exploited by any system running a Linux kernel, if power data is available). Moreover, our scheme can differentiate between low read and write-power zones, if they happen to be different. Similarly, our scheme can complement [1] by prioritizing among virtual zones within the various DIMMs. Finally, with hardware monitoring support such as [50], our scheme could perform variation-aware page migration with negligible performance/energy overheads.

3. BACKGROUND

3.1 Memory System Architecture

When discussing memory systems and devices, terminology often becomes confusing and is misinterpreted by readers. To avoid this, we will briefly define relevant terms in the memory system. In this work, we use DDR3 DRAM memory technology.

In a typical server, desktop, or notebook system, the memory controller accesses DRAM-based main memory through one or more memory *channels*. Each channel may have one or two *DIMMs*, which is a user-serviceable memory module. Each DIMM may have one or two *ranks* which are typically on opposing sides of the module. Each rank is independently accessible by the memory controller, and is composed of several DRAM devices (Figure 1 shows non-ECC with 8 devices). Inside each DRAM are multiple *banks*, where each bank has an independent memory *array* composed of *rows* and *columns*. A memory location is a single combination

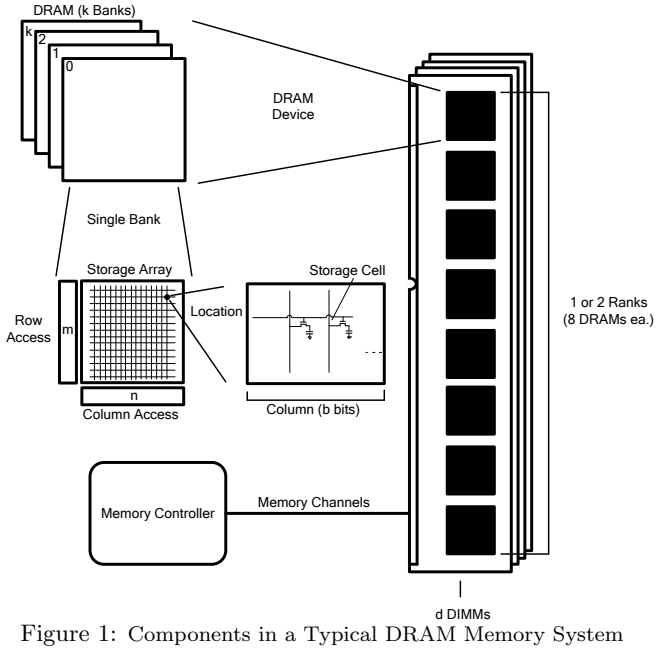


Figure 1: Components in a Typical DRAM Memory System

of rank, bank, row, and column in the main memory system, where an access is issued in parallel to all DRAMs in the selected rank. This hierarchy is depicted in Fig. 1. In this work, we are optimizing for variability measured at the DIMM level. However, our approach can be adapted for finer granularities, as previously mentioned.

3.1.1 Zoning in the Traditional Linux x86-64 Kernel

In order to understand our ViPZonE back-end implementation, we now discuss how the traditional Linux kernel partitions physical memory space. At the core of the Linux kernel memory management subsystem is the physical page allocator. When faced with an allocation request for one or more pages with certain constraints, the system tries to find the most suitable allocation in the least amount of time. The kernel may pass through multiple stages during an allocation attempt, with greater performance penalties as it tries harder to find suitable memory.

The page allocator relies on several important constructs, including, but not limited to: page structures, memory zones, page freelists, and constraint bitmasks [29]. The kernel utilizes several zones to group regions of contiguous physical memory (see Fig. 2a), required for legacy hardware support [29]. In direct memory access (DMA), devices talk directly with physical memory, bypassing the CPU. However, many legacy devices can only address the lowest 16 MB of memory, and must be able to receive page allocations in this region. The kernel, if configured to support DMA, needs to reserve this space accordingly. There are also newer devices that are capable of addressing up to 4 GB of memory, and the kernel must be able to accomodate these DMA32 devices as well, albeit with more headroom.

The kernel does this by representing these spaces with physically contiguous and adjacent DMA and DMA32 memory zones, each of which tracks pages in its space independently of other zones [29, 28]. This allows for separate bookkeeping for each zone as well, such as low-memory watermarks, buddy system page groups, performance metrics, etc. Thus, if both are supported, the DMA zone occupies the first 16 MB of memory, while the DMA32 zone spans 16 MB to 4096 MB. This means that for 64-bit systems with less than 4 GB of memory, all of memory will be in DMA or DMA32-capable zones.

The rest of the memory space not claimed by DMA or DMA32 is left to the "Normal" zone[1]. On x86-64, this will contain all memory above DMA and DMA32. Because the kernel cannot split allocations across multiple zones [29], each allocation must come from a single zone. Thus, each zone maintains its own page freelists, least-recently-used lists, and other metrics for its space.

The kernel tries to fulfill page allocation requests in the most suitable zone first, but it can fall back to other zones if required [29, 28]. For example, a user application will typically have its memory allocated in the normal zone. However, if memory there is low, it will try DMA32 next, and DMA only as a last resort (see Fig. 4a for a simplified decision flow). The kernel can also employ other techniques if required and permitted by the allocation constraints (if the request cannot allow I/O, filesystem use, or blocking, they may not apply) [29, 28]. However, the reverse is not true. If a device driver needs DMA space, it must come from the DMA zone or the allocation will fail. For this reason, the kernel does its best to reserve these restricted spaces for these situations [29].

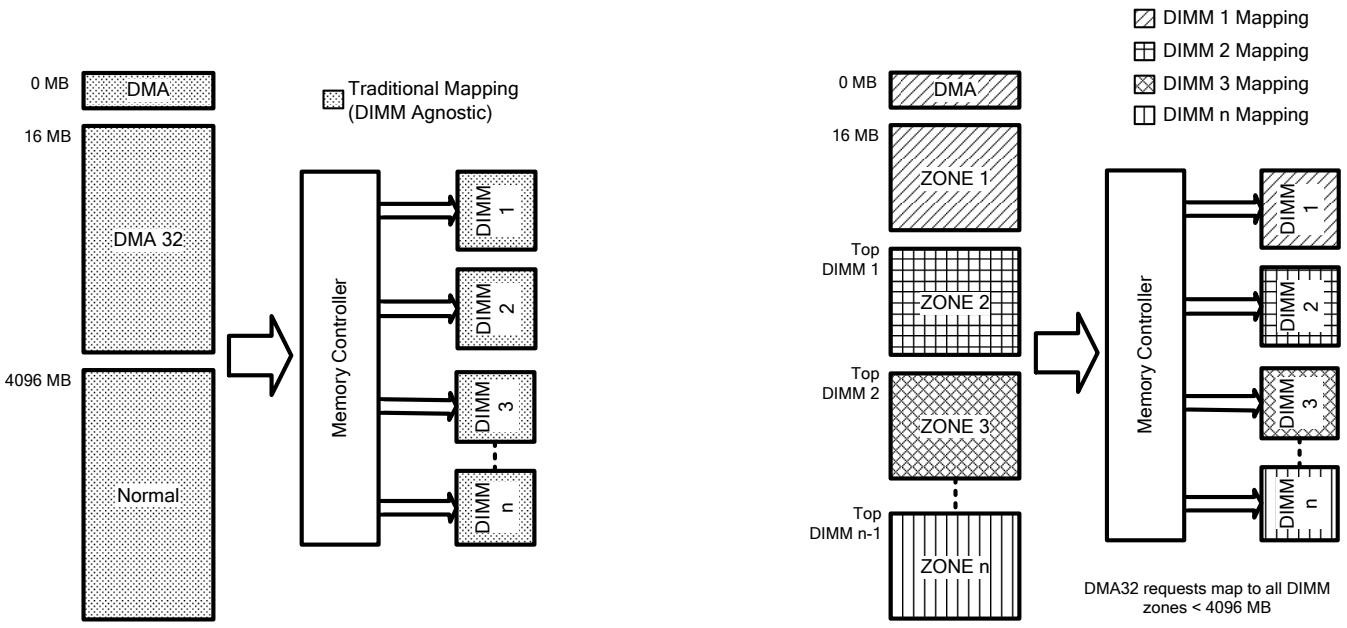
4. VIPZONE: EXPLOITING OFF-CHIP MEMORY POWER VARIATION

ViPZonE is composed of several different components in the software stack which work together to achieve power savings in the presence of DIMM variability. We refer to these separate components by the "back-end" and the "front-end" and describe them in Sec. 4.2 and Sec. 4.3, respectively. ViPZonE uses source code annotations at the application level[2], which work together with a modified GLIBC library to generate special memory allocation requests which indicate the expected use patterns (write or read dominance, and high-/medium/low utilization) to the OS. Inside the back-end Linux kernel memory management system, ViPZonE can make intelligent physical allocation decisions with this information to reduce DRAM power consumption. By choosing this approach, we are able to keep overheads in the OS a minimum, as we place most of the burden of power-aware memory requests to the application programmer. With our approach, no special hardware support is required beyond available power data or sensors that are software-accessible to the kernel.

An alternative approach could avoid requiring a modified GLIBC library and application-level source annotations, while still utilizing the ViPZonE physical page allocator. However, such an approach would place the burden of smarter page allocations on the kernel. This would likely cause a significant performance and memory overhead, as the kernel would be required to monitor applications' memory access behaviors with hardware support from the mem-

[1]The "HighMem" zone present in x86 32-bit systems is not used in the x86-64 Linux implementation.

[2]Our scheme does not currently support kernel memory allocations (e.g., kmalloc()). The goal of this paper is to provide programmers with the means to opportunistically exploit off-chip memory power variation. We are considering additional support for both kernel-level power variation-aware allocation (requires major changes to the kernel), as well as per-process allocation of memory space to the different zones through compile-time hints.



(a) Typical (b) ViPZonE

Figure 2: Typical vs. ViPZonE Physical Address Space Partitioning (Zoning) in Linux x86-64

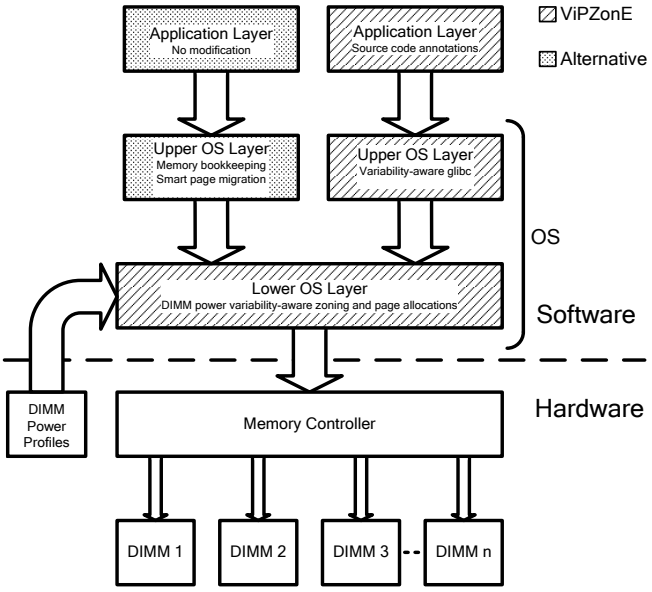


Figure 3: Layered Architecture of ViPZonE With Alternative OS-Level Scheme

ory controller. We leave the second approach as future work, and focus primarily on the use of zoning with application programmer support. The two approaches can be compared in Fig. 3.

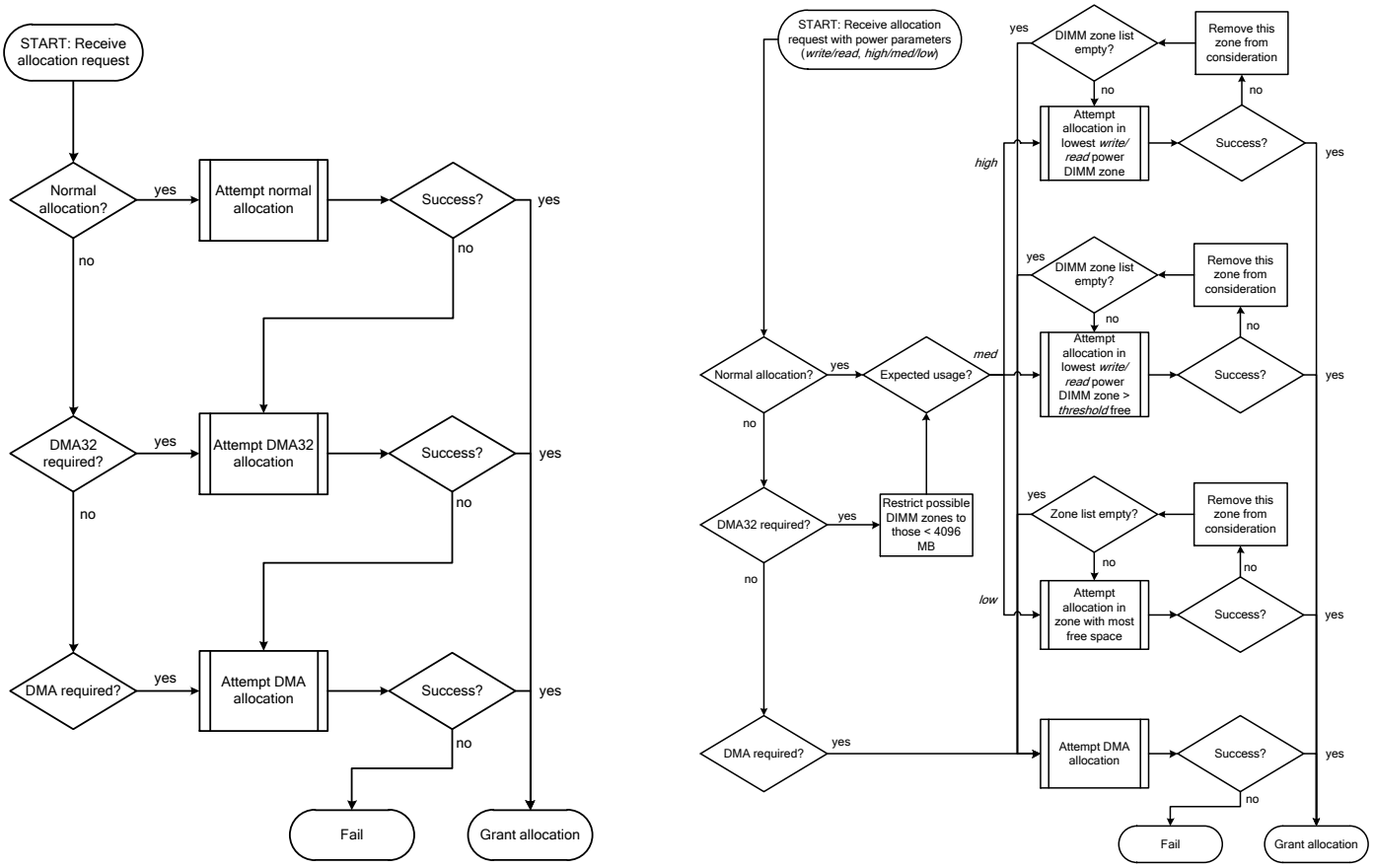
4.1 Target Platform and Assumptions

We target generic x86-64 PC and server platforms, that run on a Linux kernel and have access to two or more DIMMs (the more DIMMs, the greater the potential benefit of this work). If device-level power variation is available, then this approach could be adapted to finer granularities, depending on the memory architecture. We make the following assumptions:

- ViPZonE's page allocator has prior knowledge of the approximate write and read power of each DIMM (for an identical workload). We could detect off-chip memory power variation, obtained by one of the following methods: (1) via embedded power data in each DIMM, measured and set at fabrication time, or (2) through embedded or auxiliary power sensors sampled during the startup procedure.
- As DIMM-to-DIMM power variability is mostly fixed post-manufacturing [17], there is little need for real-time monitoring of memory power use for each module. However, if power variation changes (e.g., due to aging and wear-out), we assume these changes can be detected through power sensors in each module (left as future work).
- We can perform direct mapping of the address space (e.g., select which DIMM each read/write request goes through). This is achieved by disabling rank and channel interleaving (though our method should work in a multi-channel system with some minor adaptations). It should be possible to overcome any overheads due to this by deploying multiple memory controllers as in [41, 37, 46].
- There is programmer application-level support through the use of our power-variability enhanced GLIBC library.

4.2 Back-End: ViPZonE Implementation in the Linux Kernel

We will begin the discussion of the back-end implementation of ViPZonE by describing how the generic Linux kernel handles the physical part of page allocation requests, and how it divides the physical address space into multiple contiguous zones for this purpose. We then discuss the implementation of the ViPZonE address partitioning scheme and



(a) Typical (b) ViPZonE

Figure 4: Typical vs. ViPZonE Memory Allocation Flow in Linux x86-64 (Simplified)

allocation approach. Our implementation is based on the Linux 3.2.14 kernel [28].

4.2.1 Enhancing Physical Memory Zoning to Exploit Variability in ViPZonE Linux x86-64

In order to support memory variability-awareness, the kernel must be able to distinguish between physical regions of different power consumption. If the kernel has knowledge of these power profiles, it can construct separate physical address zones corresponding to each region of different power characteristics. The kernel can then serve allocation requests using the suggestions defined by the front-end of ViPZonE (see Section 4.3).

In the ViPZonE kernel on an x86-64 platform, we have explicitly removed the Normal and DMA32 zones, while still allowing for DMA32 allocation support. Regular DMA-able space is retained. Instead, zones are added for each physical DIMM in the system (Zone 1, Zone 2, etc.), with page ranges corresponding to the actual physical space on each DIMM. Allocations requesting DMA32-capable memory are translated to using certain DIMMs that use the equivalent memory space. Fig. 2b depicts our revised memory zoning scheme for the ViPZonE back-end. For example, in an Linux x86-64 system supporting DMA and DMA32, with 8 GB of memory located on four DIMMs (4x2 GB), the ViPZonE back-end would divide the memory space into zones as follows:

1. *DMA zone*: 0-16 MB, located on DIMM 1 physically.
2. *Zone 1*: 16-2048 MB, located on DIMM 1 physically.
3. *Zone 2*: 2048-4096 MB, located on DIMM 2 physically.
4. *Zone 3*: 4096-6144 MB, located on DIMM 3 physically.
5. *Zone 4*: 6144-8192 MB, located on DIMM 4 physically.

4.2.2 Modifying the Physical Page Allocation Algorithm in ViPZonE Linux x86-64

With this scheme, the ViPZonE back-end now has the ability to control allocations to lower power DIMMs, compared to the traditional kernel, which makes no distinction. With zones set up for each DIMM, the kernel has the essential tools it needs to make power variability-aware page allocations. The modified allocation decision flow is depicted in Fig. 4b. For example, in a system with four DIMMs, each with 2 GB of space, the ViPZonE kernel would make allocation decisions as follows:

1. *Request for DMA-capable space*: Grant in DMA zone.
2. *Request for DMA32-capable space (DMA subset implicit)*: Grant in Zone 2 or Zone 1, with the order determined by DIMM power consumption and usage hints (*write/read* dominance, and *high/med/low* utilization) passed from the front-end (see Sec. 4.3), or DMA zone (as a last resort).

3. *Request for Normal space*: Grant in Zone 4, Zone 3, Zone 2, or Zone 1, with the order determined by DIMM power consumption and usage hints (*write/read* dominance, and *high/med/low* utilization) passed from the front-end (see Sec. 4.3). Again, the DMA zone is only used as a last resort.

4.3 Front-End: High-Level APIs

The other major component of ViPZonE is implemented as the front-end, in the form of upper layer OS functionality in conjunction with annotations to application code. The front-end allows the programmer to make suggestions regarding intended use for memory allocations, so that the kernel can prioritize low power zones for frequently written or read pages.

4.3.1 Modified C-Library

We took the GNU C library (GLIBC [16]) and implemented our power variation enhanced allocation/de-allocation methods as part of the standard library (available at [3]). We will briefly describe the methods and their use. For more details please refer to our technical report [6].

Table 1: Enhanced GLIBC Methods

Function	Parameter	Type	Return Type
*void * vip_malloc*	__size	size_t	Request size
	__vflag	size_t	Bitmap flag used by ViPZonE back-end page allocator
void vip_free	__ptr	void *	Pointer to block to be freed
*void * vip_mmap*	__addr	void *	Address to be mapped (best effort mapping)
	__length	size_t	Size to be allocated
	__prot	int	PROT_READ | PROT_WRITE
	__flags	int	MAP_PRIVATE | __vflag
	__fd	int	MAP_ANONYMOUS
	__offset	off_t	
int munmap	__addr	void *	Address to be freed
	__length	size_t	Size to be allocated

There are two key functions shown in Table 1, which are exposed to the programmer and allow him/her to tell the memory manager to allocate the object in low power memory space. We implement *vip_malloc* and *vip_free* as separate calls to allow the use of custom *mmap* and *munmap* system calls (*vip_mmap* and *munmap*) that serve as hooks into the kernel. We used the kernel's *mmap/unmap* functions defined in the kernel's memory manager (mm) as templates. Furthermore, the kernel consists of *ViPZonE* helper functions that allow us to pass down the flags from the upper layers in the software stack down to the lower levels, from custom *do_vip_mmap_pgoff*, *vip_mmap_region* down to the page allocator (*__alloc_pages_nodemask*). For this purpose, we reserved the four most significant bits in the __flags field to contain the flags passed down from the upper levels of the software stack (__vflags).

Because [17] showed that there is a significant difference between read and write power consumption for the various DIMMs tested, our *vip_malloc* exploits the notion of memory pooling to construct a two-region low power memory space (high-read and high-write utilization) for a given process. These pools are constructed on the first call to *vip_malloc* via the *vip_mmap* function (which then makes the call to the kernel's virtual memory allocator – going all the way to the ViPZonE back-end physical page allocator). These pools are used to serve small requests to minimize the number of system calls (*vip_mmap*). When the pool space is exhausted or the pool space is insufficient to serve a memory allocation request, *vip_malloc* makes a call to *vip_mmap* to serve the request. These pools are managed separate from the memory pools managed by traditional *malloc*, which typically use *sbrk* to grow the heap. The *vip_free* function is used to free low power space. If the space is small enough, it is added back to the read/write pool it belongs to. If the memory space is large enough, it is freed via the *munmap* function.

Because we know that low power memory space is precious, we try to give as much space back to the OS as possible (via *munmap*) when the application no longer needs it. As a result, we prefer the use of *mmap* over traditional *sbrk*, which assumes the heap grows contiguously, and often memory is not really freed (e.g., given back to the OS) until a large amount of contiguous virtual address space is found.

The size of the low power pools is tunable as it depends upon the domain in which the GLIBC library is deployed. In embedded systems, the memory footprint tends to be smaller than the desktop domain, and much smaller than the server domain, thus the pools may range from a few KBs (at least 4 KB) of space to MBs (multiples of the page size, e.g., 4 KB) of space. One key issue is memory fragmentation. To minimize fragmentation due to many *mmap* calls, we can exploit existing patches that re-arrange the virtual address space where the *mmap* area grows from the stack downward, while the heap grows upward as normal [3].

```
int lame_encode_buffer_interleaved(                    1
   lame_global_flags *gfp,                             2
   short int buffer[], int nsamples,                   3
   char *mp3buf, int mp3buf_size)                      4
{                                                      5
  static int frame_buffered=0;                         6
  int mp3size=0,ret,i,ch,mf_needed;                    7
  ...                                                  8
                                                       9
  if (gfp->resample_ratio!=1)  {                       10
    short int *buffer_l;                               11
    short int *buffer_r;                               12
                                                       13
    /* Create two read/modify buffers with             14
           medium utilization in low power             15
           memory space                                16
    */                                                 17
    buffer_l=vip_malloc(                               18
       sizeof(short int)*nsamples,                     19
       READ | MED_UTIL);                               20
    buffer_r=vip_malloc(                               21
       sizeof(short int)*nsamples,                     22
       READ | MED_UTIL);                               23
    if (buffer_l==NULL||buffer_r==NULL) {              24
      return -1;                                       25
    }                                                  26
    ...                                                27
                                                       28
   ret = lame_encode_buffer(gfp,buffer_l,              29
      buffer_r,nsamples,mp3buf,mp3buf_size);           30
   ...                                                 31
   /* Free up the low power memory space */            32
   vip_free(buffer_l);                                 33
   vip_free(buffer_r);                                 34
                                                       35
   return ret;                                         36
  }                                                    37
}                                                      38
```

Function 1: Sample Annotated Source Code from LAME Library

4.3.2 User Annotations/Programming Model

Table 2 shows the sample set of flags supported by our GLIBC functions, passed down to the ViPZonE back-end kernel to allocate pages from the preferred zone. Function 1 shows sample code from the LAME Codec library and how to use them via our *vip_malloc* function (underlined red functions in Lines 18-23). The (*READ, WRITE*) flags tell the

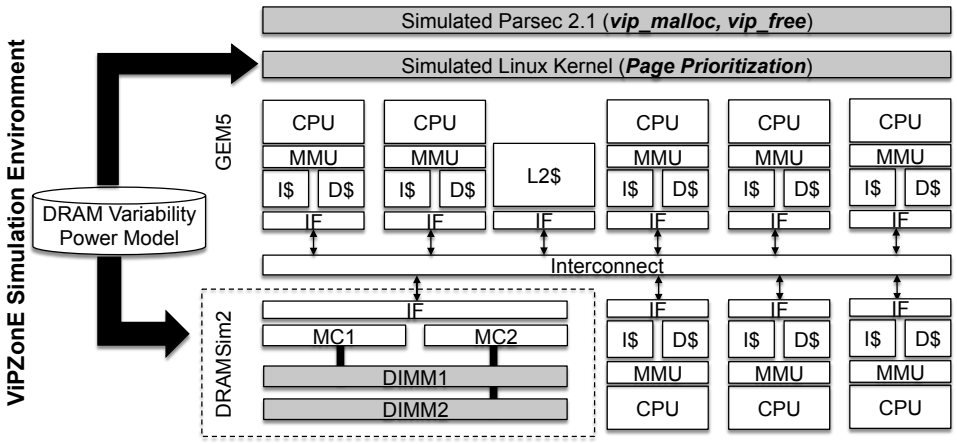


Figure 5: ViPZonE Simulation Framework

Table 2: ViPZonE Supported Flags

Parameter	Flag	Description
Access Type	WRITE	The memory space will have high write to read ratio
	READ	The memory space will have high read to write ratio
Utilization	LOW_UTIL	Low utilization
	MED_UTIL	Medium utilization
	HIGH_UTIL	High utilization

allocator that the expected workload is heavily read or write intensive. If neither write nor read is dominant, the choice of flag is left to the developer[3].

These are used to decide whether to allocate space from the read or write pools (if there is enough space) or from DIMMs with low read power or low write power. Similarly, the utilization flags (*LOW_UTIL, MED_UTIL, HIGH_UTIL*) are used to prioritize among pages from low power memory space. We decided to support these flags (only two bits) rather than using a different metric (e.g., measured utilization), since keeping track of page utilization would require higher storage and logic overheads. To free up the low power memory space, programmers would need to invoke our *vip_free* function (Function 1 Lines 33-34). As shown by this example, the necessary programming changes to exploit our variability-aware memory allocation scheme are minimal.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Setup and Goals

Our goal is to show that ViPZonE is capable of reducing power consumption with minimal (almost negligible) performance overheads. In this section we will investigate the overheads of 1) ViPZonE's software stack (Sec. 5.2) and 2) direct address mapping versus interleaved address mapping at the memory controller level (Sec. 5.3). Next, we will investigate the benefits of our approach in terms of energy savings in two scenarios: a small platform (Table 3) and a large platform (Table 3). Sec. 5.4 will investigate ViPZonE's power savings due to zone prioritization (the ideal case), then ViPZonE's power savings due to zone prioritization and selective page allocation (with the help of program annotations). Sec. 5.4 will conclude with a what-if study that investigates the effect of variation on power consumption as it reaches 100% as predicted by ITRS.

The experimental setup consists of two parts: 1) A real Linux-based ViPZonE implementation to test for execution time overheads and 2) a simulation environment to test the overheads of our direct address mapping (DIMM selection), and the benefits of our allocation mechanisms in the presence of power variation. Figure 5 shows our simulation environment, which consists of interfacing and modifying GEM5 [10] with DRAMSim2 [36] to simulate our page allocation. We simulated two DDR3 1 GB DIMMs (a total of 2 GB physical address space), each DIMM with different power consumption (due to variation). Our DRAM memory variation models are obtained from [17] and used to do realistic simulations and a what-if analysis. Table 3 shows the different platform configurations we used and their workloads, the *Sim2Core* platform has lower resources than the *Sim8Core* platform, as a result, we expect higher main memory utilization.

5.2 ViPZonE Performance Overheads

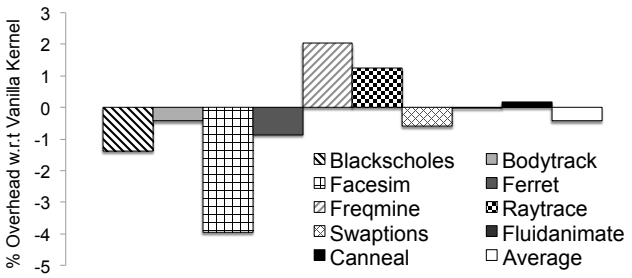


Figure 6: ViPZonE Execution Time Overheads With Respect to Typical Linux Kernel

We have implemented ViPZonE's front-end by modifying GLIBC 2.15 [16] and the Linux 3.2.14 kernel [28]. We then ran PARSEC 2.1 benchmarks [9] with simlarge workloads to estimate the performance overheads of the software stack on a real-system (*Native* configuration in Table 3, running Ubuntu 10.04 on a VirtualBox VM with 2 virtual CPUs).

As we can see from Figure 6, performance overheads due

[3]Some algorithms may allocate a common space for both write and read intensive operation, such as a buffer used in streaming applications. In these cases, the usage of vip_malloc() is left to the application programmer (e.g. he/she might split the buffer in two if possible, or arbitrarily pick write or read priority for a single space).

Table 3: ViPZonE Platform Configurations

Configuration	Cores	Threads	L1 Caches (I/D)	L2 Cache	Workload	Frequency
Sim2Core	2	4	24kB/32KB	1MB	simsmall	1.8GHz
Sim8Core	8	8	32kB/32KB	4MB	simsmall	2.0GHz
Native (i5-540M)	2 Virtual	4	-	-	simlarge	2.53GHz

to ViPZonE's zone prioritization are minimal. On average, we observe a slight performance improvement (less than 0.5%), which is in the noise. The main point is that ViPZonE's performance overheads are minimal and should not have a major effect on system performance, as the virtual page allocator is best effort. In order to effectively exploit the power variation in the memory subsystem we must support physical address mapping, i.e., we must be able to prioritize physical accesses to a particular DIMM based on how frequently a virtual page is accessed (see Sec. 5.3).

5.3 Direct Address Mapping Vs. Interleaved Address Mapping

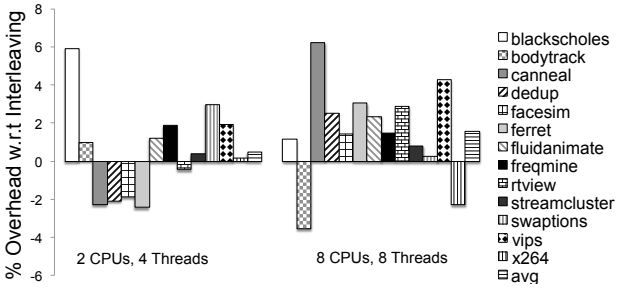


Figure 7: Execution Time Overheads of Non-Interleaved DIMM Accesses

In order to truly test the overheads of DIMM page access prioritization, we need to change the way the virtual to physical address mapping works. Typical mapping assumes rank interleaving to improve performance. As a result, in a typical system, with n DIMMs, it is possible that all DIMMs/ranks are accessed in a pipeline fashion. While one rank is opening a row, another rank may be accessed, thus hiding row access latencies. In order to address the variation of the system, we must disable this interleaving and rely on direct address mapping, i.e., each page is mapped to a single DIMM rank physically. To investigate these overheads, we explored two mappings in GEM5's memory controller and ran PARSEC benchmarks on top of two different simulation configurations (*Sim2Core* and *Sim8Core* in Table 3).

Fig. 7 shows the performance overheads due to direct versus interleaved mapping. We observe an average 1.6% performance overhead for the *Sim8Core* configuration and less than 0.5% for the *Sim2Core* configuration, which gives us a total of 1.03% execution time overhead across all benchmarks with respect to the interleaved mapping. Though a maximum of 6.2% has been observed for the single channel configuration, this overhead might increase for multi-channel configurations. To address this issue, we could exploit multiple memory controllers as shown in [41, 37, 46].

5.4 Power Savings through DIMM Prioritization

ViPZonE's goal is to save power consumption by prioritizing accesses to the low power DIMMs via zoning as discussed in Sec. 4.2.1. In order to show the true benefits of DIMM access prioritization (zoning), we simulated an ideal case

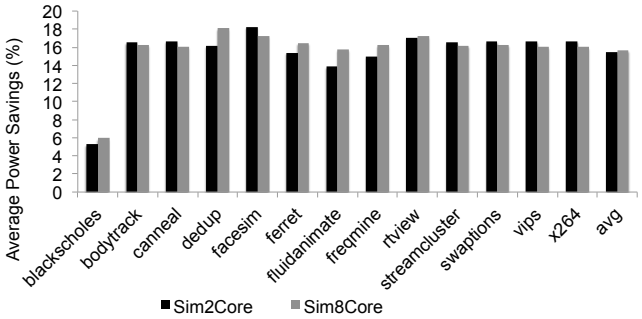


Figure 8: Average Power Savings (%) - Mapping All Data to the Ideal Zone vs. Typical Linux Kernel

Table 4: ViPZonE Maximum Physical Space

Benchmark	Required (MB)	Allocated (MB)
blackscholes	34	16
bodytrack	42	32
canneal	78	64
dedup	107	64
facesim	341	256
ferret	57	32
fluidanimate	52	32
freqmine	83	64
rtview	127	64
streamcluster	32	16
swaptions	32	16
vips	32	16
x264	32	16

where all applications had unlimited space to the low-power zone. In this configuration we mapped all pages used by the PARSEC benchmarks to the low-power DIMM modeled in DRAMSim2, while the second DIMM remained idle. In this experiment, we have one memory controller per DIMM. The base case for comparison in the next set of experiments (Sec. 5.4-5.6) is the typical scheme where a single memory controller interleaves physical pages across the two DIMMs modeled in the system and the virtual memory allocator is unaware of the power variation in the memory subsystem. We will refer to this scheme as *Typical* scheme.

Figure 8 shows the ideal savings with respect to the typical case (no DIMM prioritization and interleaving enabled). As we can observe, we see an average of 15.4% average power consumption savings for the *Sim2Core* configuration and 15.6% savings for the *Sim8Core* configuration with respect to the *Typical* scheme. The average power consumption reflects the total power consumed by the system over the entire execution time of the given workload, so it takes into account any overheads due to direct mapping with respect to interleaved mapping.

5.5 Power Savings through Annotations and Zone Prioritization

Section 5.4 discussed the power consumption savings in an ideal scenario. However, we cannot dedicate the entire physical memory space to a single application since the low power memory space is precious. Moreover, as memory utilization grows past the DIMM capacity, we will be unable to map virtual pages to the low power DIMM(s). As a result,

we must prioritize access to the low power memory space in order to minimize the power consumption of the system.

Table 4 shows the total physical space given to each of the PARSEC benchmarks we tested in our system. Some of the benchmarks have smaller footprints as the workloads we simulated were small (*streamcluster-x264*). That is, even if we ran 4 to 8 threads (*Sim2Core* and *Sim8Core*), the maximum physical low power address space will be limited to the amount shown in Table 4 for each application. We simulate the annotation behavior by letting our page allocator know the expected usage of individual virtual pages, the page allocator then decided where to map these virtual pages, be it low power memory space (e.g., low power zone/DIMM) or high power memory space. These annotations are hints, as the allocator will enforce them with best-effort given that the memory power variations will also vary among systems.

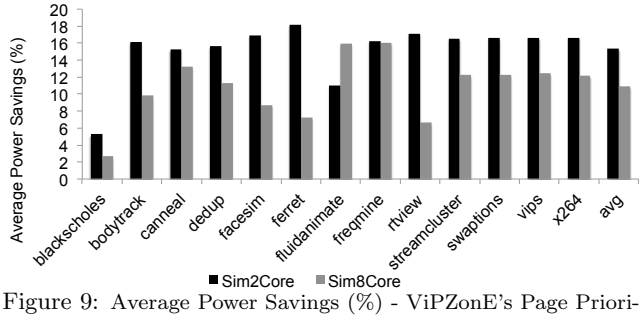


Figure 9: Average Power Savings (%) - ViPZonE's Page Prioritization vs. Typical Linux Kernel

Like Sec. 5.4, the *Sim8Core* configuration had lower DIMM utilization than the *Sim2Core* configuration due to the larger amount of on-chip memory resources. Figure 9 shows the savings due to page prioritization with respect to the traditional case (no prioritization, rank interleaving enabled). We observe an average of 15.3% power consumption savings for the *Sim2Core* configuration and 10.8% savings for the *Sim8Core* configuration (an average 13.1% across both configurations) with respect to the *Typical* scheme. The savings will grow as the memory utilization grows. As memory utilization shrinks, idle power takes over, and ViPZone will perform as good as typical memory allocators (e.g., traditional Linux memory management).

5.6 What-if Scenario: As Power Variation Reaches 100%

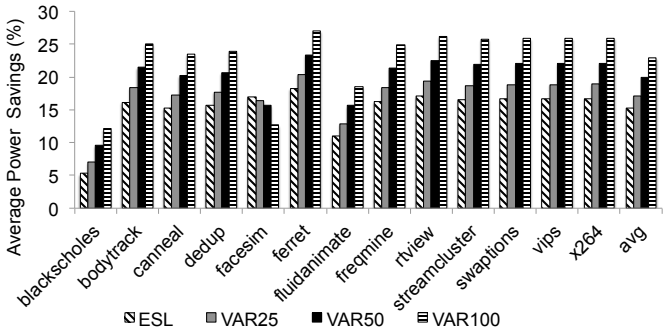


Figure 10: Average Power Savings (%) As Variation Increases - ViPZonE's Page Prioritization vs. Typical Linux Kernel

The experimental results thus far have been based on realistic variation models. In this section, we will investigate the effects of increased variation in average power consumption. Notice that like Sections 5.4 and 5.5, the benefits of our scheme are directly proportional to the main memory utilization. For this experiment, we limited the available physical low power space to the same amounts for each benchmark as in Sec. 5.5.

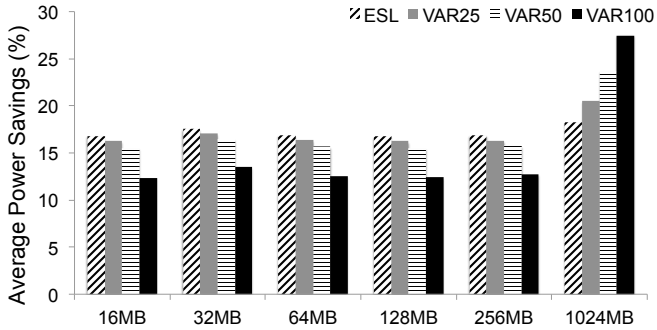


Figure 11: Average Power Savings (%) As Dedicated Memory Space Increases - ViPZonE's Page Prioritization vs. Typical Linux Kernel

Figure 10 shows the savings due to page prioritization with respect to the traditional case (no prioritization, rank interleaving enabled) as variation reaches 100% for the *Sim2Core* configuration. Here the *ESL* label refers to the variation reported in [17], while *VAR25-VAR100* refer to variation increases of 25%, 50%, and 100% respectively. As expected, the applications with the highest memory utilization (e.g., the enterprise based benchmark *dedup*) see larger average power savings. Over all benchmarks we see up to 26% average power savings as variation increases by 100%, with an average of 22.8%.

The reason *facesim* does not see the upward improvement in average power savings the other applications see (as shown in Figure 10) is that its memory requirement is higher. Figure 11 shows the average power savings of *facesim* with respect to the *Typical* scheme as the dedicated physical space increases for the different projections (*ESL-VAR100* running the *Sim2Core* configuration. Though the average power consumption savings increase with more dedicated memory, the true benefits of our scheme are more noticeable as we dedicate more space to the *facesim*. In the cases of 16 MB to 256 MB, we notice that though most accessed pages are mapped to low-power DIMM, the second DIMM consumes much more power due to the variation. Since idle power is dominant, it will grow equally for both the *Typical* scheme and our scheme, which leads to the behavior seen in Figures 10 and 11.

6. CONCLUSION AND FUTURE WORK

In this paper we presented ViPZonE (OS-Level Memory *V*ariability-Driven *P*hysical Address *Z*oning for *E*nergy Savings), a system-level variability-aware solution that adapts to the power variability inherent in a set of DRAM memory modules. ViPZonE is portable enough to be deployed in any environment running Linux (from laptops to high-end servers). Our experimental results across various configurations running PARSEC workloads show an average of 13.1% memory power consumption savings at the cost of a modest 1.03% increase in execution time over a typical Linux virtual memory allocator. Our ongoing and future work includes: 1) The complete deployment of our software prototype on an Atom-board and an Intel i7 board to truly evaluate the ben-

efits of our scheme (with support for more than 4 DIMMs). 2) The adoption of server- and scientific-based workloads. 4) Bookkeeping and power variability-aware dynamic page migration between zones. 5) Kernel-space and compiler-driven per-process zone prioritization. 6) Possibly exploring and exploiting memory power variation in mobile platforms (e.g., Android OS). 7) If the data is available, intra-DIMM power variation.

Acknowledgment

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

7. REFERENCES

[1] J. Ahn et al. Multicore DIMM: an energy efficient memory module with independently controlled DRAMs. *IEEE Computer Architecture Letters*, 2008.

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

[3] Bathen and Gottscho. ViPZonE: Memory Power Variation-aware Software Stack. *http://vipzone.sourceforge.net*, 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] L. Bathen et al. A variability-aware software stack for linux-based system. *UCI Center for Embedded Computer Systems TR*, 2012.

[7] M. Bennaser et al. Designing memory subsystems resilient to process variations. In *ISVLSI*, 2007.

[8] S. Bhardwaj et al. Modeling of intra-die process variations for accurate analysis and optimization of nano-scale circuits. In *DAC*, 2006.

[9] C. Bienia et al. The parsec benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[10] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[11] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *DAC*, 2003.

[12] K. Bowman et al. Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. In *ISLPED*, 2007.

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

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

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

[16] GNU. Glibc, the gnu c library. *http://www.gnu.org/software/libc/*, 2012.

[17] M. Gottscho et al. Power variability in contemporary DRAMs. *IEEE Embedded Systems Letters*, 2012.

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

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

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

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

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

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

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

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

[26] X. Li. Rethinking memory redundancy: optimal bit cell repair for maximum-information storage. In *DAC '11*, 2011.

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

[28] Linux. Kernel 3.2.14. *http://www.kernel.org/*, 2012.

[29] R. Love. *Linux Kernel Development*. Pearson Education, Inc., 3 edition, 2010.

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

[31] K. Meng et al. Process variation aware cache leakage management. In *ISLPED '06*, 2006.

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

[33] M. Mutyam et al. Process-variation-aware adaptive cache architecture and management. *IEEE Trans. Comput.*, 2009.

[34] Y. Pan et al. Selective wordline voltage boosting for caches to manage yield under process variations. In *DAC*, 2009.

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

[36] P. Rosenfeld et al. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[37] J. Sancho et al. Analyzing the trade-off between multiple memory controllers and memory channels on multi-core processor performance. In *IPDPSW*, 2010.

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

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

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

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

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

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

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

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

[46] T. Xu et al. Optimal memory controller placement for chip multiprocessor. In *CODES+ISSS*, 2011.

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

[48] B. Zhao et al. Variation-tolerant non-uniform 3d cache management in die stacked multicore processor. In *MICRO*, 2009.

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

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

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