ViPZonE: Exploiting DRAM Power Variability for Energy Savings in Linux x86-64[1]

Mark Gottscho
mgottscho@ucla.edu

M.S. Project Report
NanoCAD Lab, UCLA Electrical Engineering
nanocad.ee.ucla.edu

Advised by Dr. Puneet Gupta
puneet@ee.ucla.edu

Research Collaborators: Dr. Nikil Dutt, Dr. Alex Nicolau, Dr. Luis A. D. Bathen
dutt@ics.uci.edu, nicolau@ics.uci.edu, lbathen@uci.edu

Wednesday 5th March, 2014

[1]NOTICE: Some portions of this material have appeared in peer-reviewed and invited conference and journal publications [1, 2, 3, 4]. Some content may appear in future publications. ©Mark Gottscho 2014

Abstract

Hardware variability is predicted to increase dramatically over the coming years as a consequence of continued technology scaling. I apply the Underdesigned and Opportunistic Computing (UnO) paradigm by exposing system-level power variability to software to improve energy efficiency. ViPZonE is a Linux-based memory management solution in conjunction with application annotations that opportunistically performs memory allocations to reduce DRAM energy. In this report, I discuss in depth the relevant details of DDR3 DRAM memory and the Linux virtual memory system before moving on to my research in DRAM power variability and the ViPZonE innovations. ViPZonE's components consist of a physical address space with DIMM-aware zones, a modified page allocation routine, and a new virtual memory system call for dynamic allocations from userspace. ViPZonE was implemented in the Linux kernel with GLIBC API support, running on an x86-64 hardware testbed with significant access power variation in its DDR3 DIMMs. I demonstrate that on the testbed, ViPZonE can save up to 27.80% memory energy, with no more than 4.80% performance degradation across a set of PARSEC benchmarks tested with respect to the baseline Linux software. Furthermore, through a hypothetical "what-if" extension, I predict that in future non-volatile memory systems which consume almost no idle power, ViPZonE could yield even greater benefits, demonstrating the ability to exploit memory hardware variability through opportunistic software.

Chapter 1

Introduction

Modern digital integrated circuits (ICs) exhibit significant variability as a consequence of imperfections in the fabrication processes [5, 6], use patterns, aging, and the environment [7]. Inter-die and intra-die process variations have become significant as a result of continued technology scaling into the deep submicron region [8, 9]. The International Technology Roadmap for Semiconductors (ITRS) predicts that over the next decade, both performance and power consumption variation will increase by up to 66%, and 100%, respectively [10].

System design typically assumes a rigid hardware/software interface contract, hiding physical variations from higher layers of abstraction [3]. However, the typical approach of guardbanding for variability is expensive [11]. Guardbanding is a method that ensures reliable and consistent components over all operation and fabrication corners. There are many associated costs from over-design, such as chip area and complexity, power consumption, and performance. Despite considerable hardware variability, the rigid hardware/software contract results in software assuming strict adherence to the hardware specifications.

The overheads of guard-banding are reduced, but not eliminated, through the practice of binning, where manufacturers market parts with considerable post-manufacturing variability as different products. For example, manufacturers have resorted to binning processors by operating frequencies to reduce the impact of inter-die variation [3]. However, even with guardbanding, binning, and dynamic voltage and frequency scaling (DVFS), variability is inherently present in any set of manufactured chips. Furthermore, with the emergence of multi-core technology, intra-die variation has also become an issue. To minimize the overheads of guardbanding, recent efforts have shown that exploiting the inherent variation in devices [12, 13] yields significant improvements in overall system performance.

As a result, there is growing interest in software as well as hardware mechanisms to adapt to variations or compensate for them. Examples of explicit variation-awareness in the software stack include power management [14]; embedded sensing [15]; and video encoding [16].

This has led to the notion of the Underdesigned and Opportunistic (UnO) computing paradigm [3], depicted in Fig. 1.1. In UnO systems, design guardbands are reduced while some hardware variations are exposed to a flexible software stack. This allows the system to tune itself to suit the unique characteristics of its hardware that arise from process variations (part variability), aging effects, and environmental factors such as voltage and temperature fluctuations (time variability).

To develop effective methods of addressing variations (especially in the software layers), it is important to understand the extent of variability in different components of computing systems and their dependence on workload and environment. Though variability measurements through simple silicon test structures abound (e.g., [17], [18]), variability characterization of full components and systems have been scarce. Moreover, such measurements have been largely limited to processors (e.g., 14X variation in sleep power of embedded microprocessors [15] and 25% performance variation in an experimental 80-core Intel processor [19]). Hanson et al. [20] found up to 10% power variation across identical Intel Pentium M processors, and up to 2X active power variation across various DRAMs.

Efforts dealing with variation have focused on mitigating and exploiting it in processors [12, 13, 16, 15] or in on-chip memory [21, 22, 23, 24, 25]. Fewer papers have looked at variability in off-chip, DRAM-based memory subsystems. As off-chip DRAM memory may consume as much power as the processor in a server-class system [26] (e.g., 48% of total power in [27]) and is likely to increase for future many-core platforms

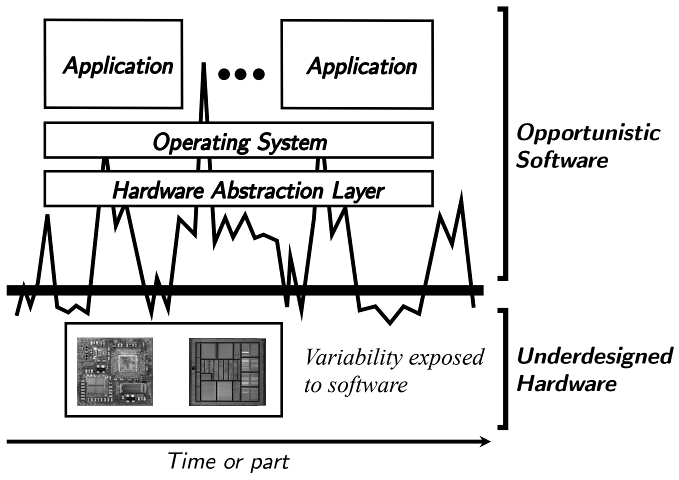


Figure 1.1: Underdesigned and Opportunistic (UnO) computing concept adapted from [3], where hardware variability across different parts or over time is deliberately exposed to software layers to improve energy efficiency, performance, cost, etc.

(e.g., Tilera's TILEPro64 and Intel's Single Chip Cloud Computer (SCC)), memory power variations could have a significant impact on the overall system power.

This has motivated several efforts to reduce DRAM (dynamic random access memory) power consumption (e.g., power-aware virtual memory systems [26, 28, 29]). Many works on main memory power management have focused on minimizing accesses to main memory through smart caching schemes and compiler/OS optimizations [30, 31, 32, 33, 34], yet none of these methods have taken memory variability into account. Bathen et al. [35] recently proposed the introduction of a hardware engine to virtualize on-chip and off-chip memory space to exploit the variation in the memory subsystem. These designs require changes to existing memory hardware, incurring additional design cost. As a result, designers should consider software implementations of power and variation-aware schemes whenever possible. Moreover, a variability-aware software layer should be flexible enough to deal with the predicted increase in power variation for current and emerging memory technologies.

A "variability-aware" operating system could exploit variability in DRAM, making physical allocation decisions in real time to reduce overall memory power consumption. Memory virtualization techniques such as [25] could be extended to account for power variations in off-chip memories. To the best of my knowledge, [20] is the only prior study to present measured DIMM power variability; it explored running systems, including component-level sources such as CPUs and DDR2 DRAM but primarily focused on vendor-dependent variations. [36] included an investigation on operation and data dependence of memory power, but used SRAMs on an older 0.35μm process node.

1.1 Contributions and Report Organization

This work presents ViPZonE, an OS-based, pure software memory (DRAM) power variability-aware memory management solution with a full software implementation that is evaluated on a real hardware testbed. ViPZonE adapts to the power variability inherent in a given set of commodity DRAM memory modules by harnessing disjunct regions of physical address space with different power consumption. My approach exploits variability in DDR3 memory at the DIMM modular level, but the approach could be adapted to work at finer granularities of memory, if variability data and hardware support are available. My experimental results across various configurations running PARSEC [37] workloads show an average of 27.80% memory energy savings at the cost of no more than a modest 4.80% increase in execution time over an unmodified Linux virtual memory allocator.

The key components and contributions of this work are as follows:

- Background material on DRAM operation, memory hierarchy organization, and main memory interleaving, to better understand the following other contributions. This is presented in Chapter 2.

- An in-depth discussion of the mechanisms and policies implemented in the Linux kernel virtual memory system as of version 3.2, for better understanding the ViPZonE innovations, presented in Chapter 3.
- An empirical study of power variability in contemporary DDR3 DRAMs, which is based on my previously published work [1], presented in Chapter 4. This includes the following:
  - An analysis of instance, vendor and temperature dependent power variability in contemporary DRAMs.
  - A characterization of power dependence on inputs and operation.
- The ViPZonE power variability-aware memory management scheme (originally proposed in my 2012 paper [2]), discussed in Chapter 5, which includes the following:
  - A detailed description and complete implementation of ViPZonE, including modifications to the Linux kernel and standard C library (GLIBC). These changes allow programmers control of power variability-aware dynamic memory allocation.
  - An analysis of DDR3 DRAM channel and rank interleaving advantages and disadvantages using my instrumented x86-64 testbed, and the implications for variability-aware memory systems.
  - An evaluation of power, performance, and energy of the ViPZonE implementation using a set of PARSEC benchmarks on my testbed.
  - A hypothetical evaluation of the potential benefits of ViPZonE when applied to systems with negligible idle memory power (e.g., emerging non-volatile memory technologies).
- Conclusions and discussion of future work in Chapter 6.

ViPZonE is the first OS-level, pure-software, and portable solution to allow programmers to exploit main memory power variation through memory zone partitioning. The source code is available at [38].

Chapter 2

Background: DRAM Operation and Memory Architecture

Today's general purpose computers and servers utilize dynamic random access memory (DRAM) for the primary memory architecture. In a modern hierarchical memory system, DRAM lies between one or more high-performance, small, and expensive caches and slow, large, and cheap non-volatile storage (i.e. hard drives and flash). Currently, DRAM is the ideal tradeoff between capacity, performance, and price for main memory. Due to its random access capabilities, data and instructions can be stored at any location within the storage arrays.

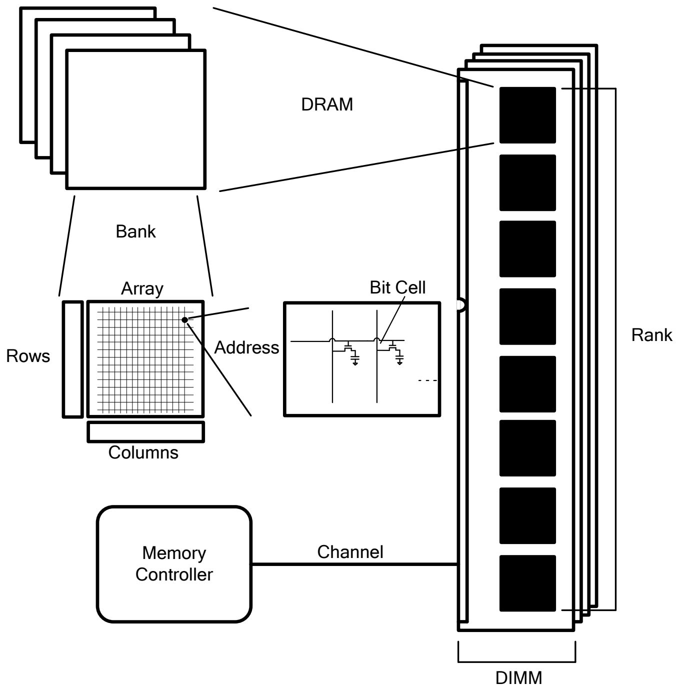


Figure 2.1: Components in a typical DDR3 DRAM memory system.

To avoid confusion, I briefly define relevant terms in the memory system. In this work, I 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 more *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, typically eight for non-ECC modules. 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 of DIMM, rank, bank, row, and column in the main memory system, where an access is issued in parallel to all DRAMs, in lockstep, in the selected rank. This organization is depicted in Fig. 2.1.

For example, when reading a DIMM in the DDR3 standard, a burst of data is sent over a 64-bit wide bus for 4 cycles (with transfers occurring on both edges of the clock). During each half-cycle, one byte is obtained from each of eight DRAMs operating in lockstep, thus composing 64 bits that can be sent over the channel. Note that this is not "interleaving" in the sense that we use throughout the paper, meaning that it is not configurable or software-influenced in any way; it is hard-wired according to the DDR3 standard.

2.1 DRAM Operation

A bit in DRAM is maintained as charge on a capacitor, accessed through an NMOS transistor connected to a *bit line*. A full charge of the cell capacitor at V_{DD} represents a 1, while a discharged capacitor represents a 0. The NMOS transistor acts as a switch for the charge, and is off when the *word line* is low. This maintains the bit stored in the cell. When the word line is asserted (pulled up to $V_{DD} + V_T$), the NMOS transistor turns on and allows writes or reads from the cell. Fig. 2.2 depicts the internal DRAM architecture, which will be a useful reference throughout this subsection.

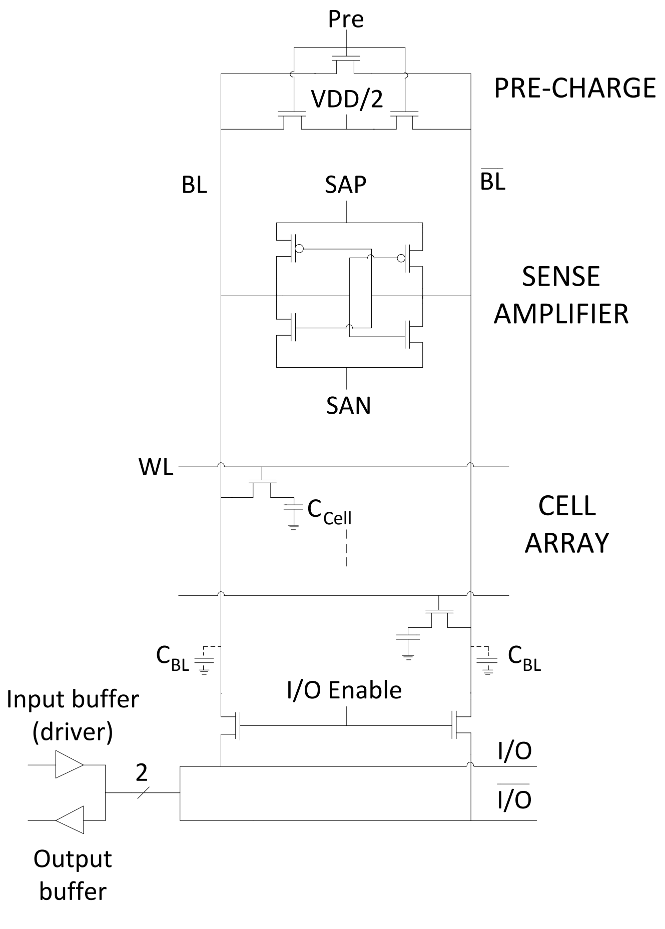


Figure 2.2: DRAM internals. Based on material from [39].

There are three primary stages in a simplified DRAM operation. Firstly, all of the bit lines in a bank must be *pre-charged*, where the voltage is set to $V_{ref} = V_{DD}/2$ [40]. Next, an *access* occurs on the word line corresponding to the decoded row address. This causes all the cells along the selected word line to share their charges with their bit lines. In the third stage, a *sense* operation occurs, which is then followed by *restoration* of the accessed cells and either a *read* or *write* on the selected column.

2.1.1 Pre-Charge

A DRAM cell typically has a much smaller capacitance than the bit line to which it is connected; in a simple array, a bit line might be connected to all m rows in the bank [39]. For this reason, the bit line must be pre-charged to V_{ref}, as mentioned above, so that the value stored in the cell may be detected in the sense and restore phase (see Access as well as Sense and Restore below). This is done by asserting a signal Pre that turns on three NMOS transistors to connect BL to $\overline{BL}$. This effectively shorts the two bit lines together such that they stabilize at V_{ref} (after a sense and restore operation and before pre-charge, one bit line will be at V_{DD} and the other at GND).

2.1.2 Access

In the access stage, a word line (WL is used to select an entire row of cells, by turning all of them on when asserted. A column, consisting of several bit lines, selects which section of that row to access. When a cell is accessed, it shares its charge with the bit line, contaminating the data within the cell. Assuming the pre-charge stage has occurred and the bit line voltage is initially V_{ref}, the bit line voltage is perturbed by a small signal [39]

$$v_s = V_{ref} \cdot \frac{C_{cell}}{C_{bitline} + C_{cell}}.$$

If the cell stored a 1, then let $V_{ref}^{+} = V_{ref} + v_s$. Otherwise, let $V_{ref}^{-} = V_{ref} - v_s$. After an access, both the bit line and the cell settle at either V_{ref}^{+} or V_{ref}^{-}, depending on whether the cell stored a 1 or 0, respectively.

2.1.3 Sense and Restore

Because cell accesses are fundamentally destructive of data, each cell along the selected word line must have its charge restored. Because an access only causes a small perturbation of the bit line voltage, the difference must be amplified such that the cell voltage can be restored to its initial value. Each bit line, BL, has a corresponding "dummy" bit line, $\overline{BL}$, that serves as its data complement. Like the main bit line, $\overline{BL}$ is also pre-charged to V_{ref}. When an access occurs, only BL shares charge with the cell. During the sense and restore stage, a *sense amplifier* amplifies the differential voltage v_s between BL and $\overline{BL}$. The sense amplifier then holds the bit lines to their saturated values until the selected cell on each line is charged up (restored) to the initial value.

For example, if a cell that containing a 1 is accessed, BL goes to V_{ref}^{+} and $\overline{BL}$ stays at V_{ref}. The sense amplifier, through the use of positive feedback, amplifies the difference between the two bitlines (v_s). BL is pulled up to V_{DD} and $\overline{BL}$ is pulled down to GND. Similarly, for accessing a cell containing a 0, BL is pulled down to GND and $\overline{BL}$ is pulled up to V_{DD}.

After accessing, sensing, and restoring a row, it is open for input and output. Only one row within a bank may be open at any time. Once the cell restoration is complete, any combination of read and write operations can be performed on the open row. When all operations on the row are complete, the row must be closed by a pre-charge operation.

2.1.4 Read Operation

A read operation may follow a sense and restore by simply multiplexing a (fully driven) column onto an input/output bus. When the bit lines are saturated and the accessed cells fully restored, the bit lines in the selected column can then be read out through an I/O bus to an output buffer. A simple control signal controls access from each bit line to the I/O bus. The other accessed cells on the word line that were not requested for a read do not place their data on the I/O bus lines. An example involving a read 1 operation follows, along with a graphical depiction in Fig. 2.3:

- *Pre-charge*: Pre is asserted and the bit lines are ready for an access, with voltages equal to V_{ref}.

- *Access*: WL is asserted to $V_{DD} + V_T$ to account for the threshold voltage of the NMOS cell transistor. Charge sharing occurs between C_{cell} and C_{BL}, bringing the voltage on both to V^+_{ref}. Other bit lines also undergo charge sharing.
- *Sense and Restore*: The sense amplifier pulls BL and C_{cell} to V_{DD} and $\overline{BL}$ to GND. Other cells on the row are concurrently sensed and restored.
- *Read*: Once BL and $\overline{BL}$ are stable, I/O_enable is asserted and column data is transferred to the output buffer. Other reads or writes may follow on the open row.
- *Pre-charge*: Pre is asserted once again and the row is closed by a pre-charge operation.

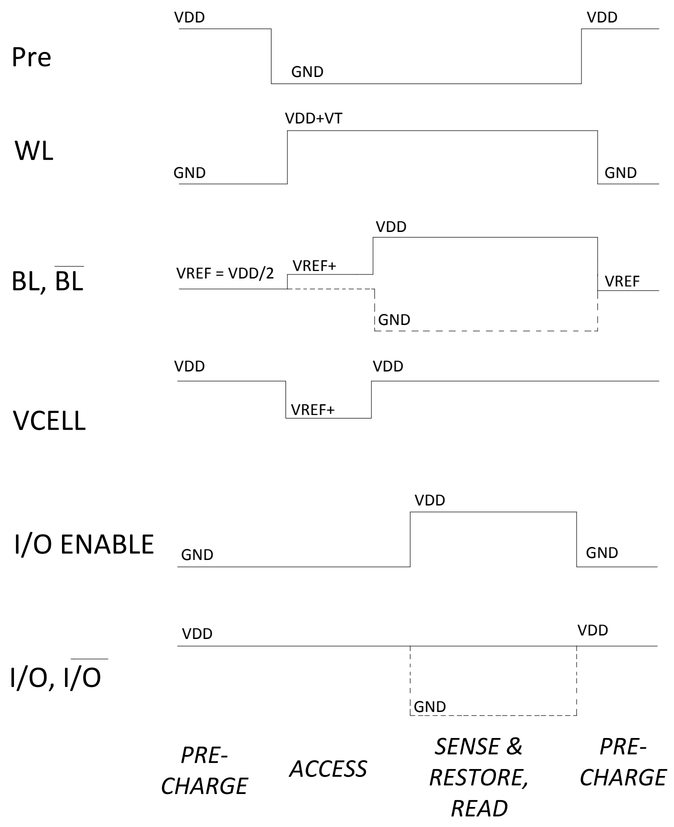


Figure 2.3: Read 1 waveform, assuming no rise/fall time. Based on material from [39].

2.1.5 Write Operation

Due to the nature of the DRAM array architecture, a write operation in DRAM is also preceded by a sense and restore operation on the selected row, because the non-requested cells must have their charges replenished [39]. After the sense and restore stage is complete, BL and the open cell are driven to the input voltage, and $\overline{BL}$ to its complement, and the write of the new value is completed. Note that this may cause a bit line to be charged to V_{DD} or GND during the restoration phase, only to be forced to fully charge or discharge in the opposite direction during a write. An example involving a write 0 over 1 operation follows, along with a graphical depiction in Fig. 2.4:

- *Pre-charge*: Pre is asserted and the bit lines are ready for an access, with voltages equal to V_{ref}.

- *Access*: WL is asserted to $V_{DD} + V_T$ to account for the threshold voltage of the NMOS cell transistor. Charge sharing occurs between C_{cell} and C_{BL}, bringing the voltage on both to V_{ref}^{+}. Other bit lines also undergo charge sharing.
- *Sense and Restore*: The sense amplifier pulls BL and C_{cell} to V_{DD} and $\overline{BL}$ to GND. Other cells on the row are concurrently sensed and restored.
- *Write*: I/O and $\overline{I/O}$ are set to the input value and its complement, respectively. I/O_enable is asserted. BL and C_{cell} are pulled down to GND, while $\overline{BL}$ is pulled up to V_{DD}. Other cells in the column are simultaneously written with the input data. Other reads or writes may follow on the open row.
- *Pre-charge*: Pre is asserted once again and the row is closed by a pre-charge operation.

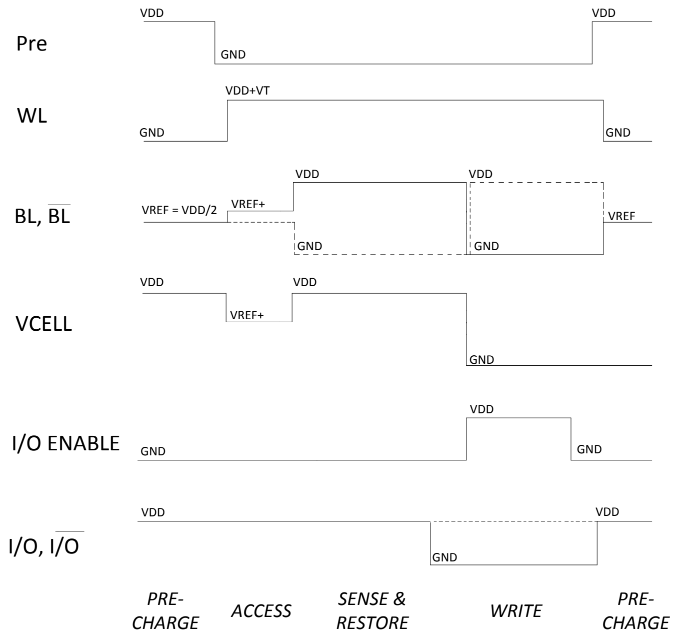


Figure 2.4: Write 0 over 1 waveform, assuming no rise/fall time. Based on material from [39].

2.1.6 Refresh Operation

The *refresh* operation is required to maintain data integrity over periods of inactivity. Because data is represented as charges on capacitors, they leak over time. Because of this, cells will eventually discharge and corrupt the original data. To counter this problem, the DRAM must be periodically refreshed at a rate sufficient to maintain data integrity in each cell in the device. A refresh operation is essentially implemented as an access, sense, and restore on each row in the array, and is detrimental to performance.

2.2 Main Memory Interleaving

It is common practice for system designers to employ interleaved access to parallel memories to improve memory throughput, particularly for parallel architectures, e.g. vector or SIMD machines [41]. This is done by mapping adjacent chunks (the size of which is referred to as the *stride*) of addresses to different physical memory devices. Thus, when a program accesses several memory loctions with a high degree of spatial (in the linear address space) and temporal locality, the operations are overlapped via parallel access, yielding a speedup.

Many works have explored interleaving performance, generally in the context of vector and array computers, but also with MIMD machines as the number of processors and memory modules scale [42, 43]. While widely used today, interleaving does not necessarily yield improved performance. For example, the technique makes no improvement in access latency, and there is little performance gain when peak memory bandwidth requirements or memory utilization are low (e.g., high arithmetic intensity workloads as defined by the roofline model of computer performance [44]).

Furthermore, interleaving may yield negligible speedup when access patterns do not exhibit high spatial locality (e.g., random or irregular access), and is also capable of *worse* performance when several concurrent accesses have module conflicts as a result of the address stride, number of modules, and interleaving stride [45, 41]. Researchers have come up with techniques to mitigate or avoid this issue, usually through introducing irregular address mappings. For example, the Burroughs Scientific Processor used a 17-module parallel memory and argued that prime numbers of memory modules allowed several common access patterns to perform well [46]. Other approaches suggested skewed or permutation-based interleaving layouts [47], and clever data array organization for application-specific software routines [48].

Chapter 3

Background: Linux Virtual Memory

At the core of the Linux kernel virtual memory (VM) subsystem is the physical page allocator. While the upper layers of the VM system primarily focus on process page tables, page faults, disk swap handling, file or device-mapped space, slab caching, etc., all memory must eventually be mapped to physical space to be used. After the kernel image has been loaded into memory and VM paging has been enabled through architecture-dependent bootstrapping procedures, all memory is referenced virtually, even inside the kernel memory management code itself. After paging is enabled, all kernel and user code uses virtual addresses and the page tables.

In order to make this work, physical pages must be allocated dynamically for processes in the system. This is done by a central physical page allocator that lies below the rest of the VM system. When faced with an allocation request for one or more pages with a set of 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 increasing performance penalties as it tries harder to find suitable memory.

We concern ourselves with the Intel x86-64 symmetric multi-processor (SMP) or uniform memory access (UMA) architecture.

3.1 Relevant Files

The Linux memory management code is under the *mm/* directory in the source root directory [49]. This directory encompasses the entire VM system implementation, and not all files are directly relevant to the ViPZonE implementation. Headers for the mm system are located in the *include/linux* directory. The most relevant files for the physical page allocator are listed below with a description of the portions of interest to ViPZonE:

1. ***include/linux/gfp.h*** This header contains the definitions for the *gfp_mask* type, which is important to the physical page allocation system for representing allocation constraints. This also includes small helper functions and macros for extracting fields from the *gfp_mask*.

2. ***include/linux/mmzone.h*** This header defines the *struct zone* type. Zones are used to partition the physical memory address space into contiguous regions suitable for different types of allocations.

3. ***include/linux/mm_types.h*** This file includes some important type definitions such as *struct page* and helper functions.

4. ***mm/page_alloc.c*** This is the core code for the physical page allocator using the buddy system algorithm. Most of the functionality is implemented here. The heart of the allocator is the *alloc_pages_nodemask()* function. There are several "APIs" within the kernel to allocate pages, which all boil down to this function.

5. ***arch/x86/mm/init_64.c*** Used for the architecture-dependent x86-64 bootstrapping of the mm system. Among many tasks, it sets up the page ranges for all zones in the system. A function of particular interest is *paging_init()*.

6. ***arch/x86/kernel/sys_x86_64.c*** Used to implement the *mmap()* syscall.
7. ***arch/x86/kernel/syscall_table_32.S*** Contains an in-order syscall lookup table. This is critical for the kernel to be able to process syscalls. The table is indexed using unique syscall numbers.
8. ***arch/x86/include/asm/syscalls.h*** Header file containing syscall declarations, without definitions. Functions are prefixed with *asmlinkage sys_*.
9. ***arch/x86/include/asm/unistd_32.h*** This file contains the *#define*s for syscall numbers, used to index into the syscall table. 32-bit.
10. ***arch/x86/include/asm/unistd_64.h*** This file contains 64-bit syscall numbers.
11. ***include/linux/syscalls.h*** Header file containing syscall function declarations. This serves a similar purpose to *arch/x86/include/asm/syscalls.h* but is not architecture dependent.
12. ***include/asm-generic/unistd.h*** This file contains some generic assembly syscall definitions.
13. ***include/asm-generic/mman.h*** This file contains some *#define* definitions relevant to virtual memory.
14. ***mm/mmap.c*** This file defines the code for *mmap()*.
15. ***mm/mempolicy.c*** Implements the *move_pages()* system call.
16. ***mm/migration.c*** Implements the various functions needed for page migration.

3.2 Describing Physical Memory: Nodes, Zones, and Pages

Linux supports non-uniform memory access (NUMA) and uniform memory access (UMA) architectures [50]. NUMA introduced the concepts of "nodes", which represent regions of physical memory of different performance, CPU locality, etc. In UMA we just have one node of memory. That is, all DRAM space is considered to be uniformly accessible (a single node) in the Linux kernel for x86. UMA is used in the x86-64 systems in this report, so NUMA/nodes will not be discussed further. Fig. 3.1 depicts the structural relationships between nodes, zones, and pages [50].

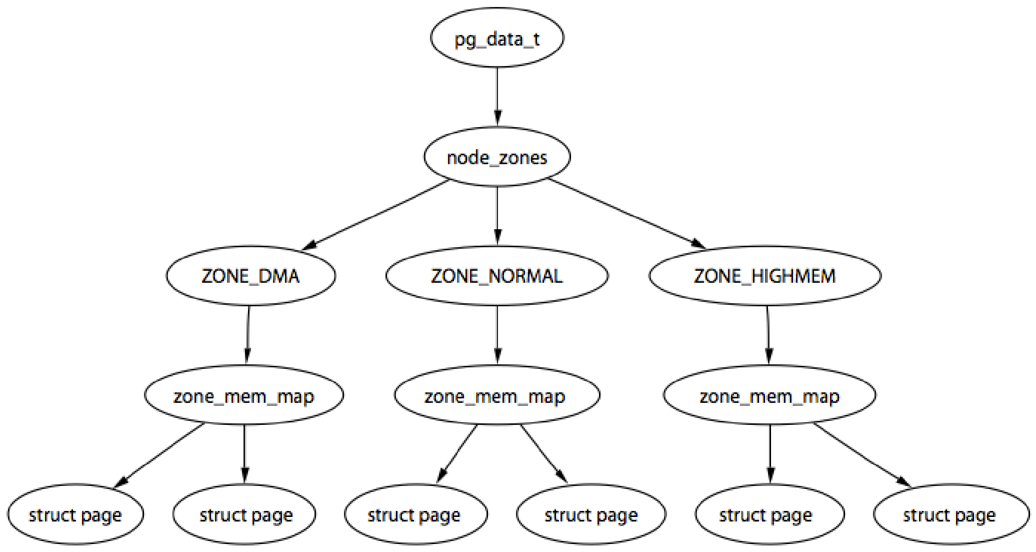


Figure 3.1: Relationship between nodes, zones, and pages. From [50].

3.2.1 Zones

The Linux kernel divides each node into several zones to group regions of contiguous physical memory (see Fig. 3.1 and Fig. 3.2). This is required for legacy device support and older 32-bit kernels. In Direct Memory Access (DMA), devices talk directly with physical memory, bypassing the CPU. Legacy devices can only address 16 bits; therefore, they must receive page allocations in the lowest 16 MB[1] of physical memory. The kernel, if configured to support DMA, needs to reserve this space accordingly. It does this by representing this space with a DMA memory zone, which can track pages independently of any other zones. This allows for separate statistics bookkeeping for each zone as well, such as low-memory watermarks. 32-bit DMA capable devices can use the DMA32 zone, if configured. Thus, if both are supported, the DMA zone occupies the first 16 MB of memory while DMA32 occupies 16 MB-4096MB (or whatever the maximum physical memory address is). This means that for 64-bit systems with less than 4 GB of memory, all of it is allocated in DMA or DMA32-capable zones.

In a 32-bit Linux system, it is possible that the system is not able to virtually address all of the physical address space (for example, running 32-bit kernel on 64-bit hardware, or a system using Physical Address Extension, also known as PAE). In this case, the "High Memory" zone is used. In this case, only a portion of the physical memory is mapped into the kernel's virtual address space at a time. Memory is dynamically remapped on demand. However, as we are looking only at x86-64, the virtual address space is larger than the physical address space. Thus, HighMem is not part of the kernel [49], and it will not be discussed further. See Fig. 3.2 for a visual depiction of x86-64 zoning.

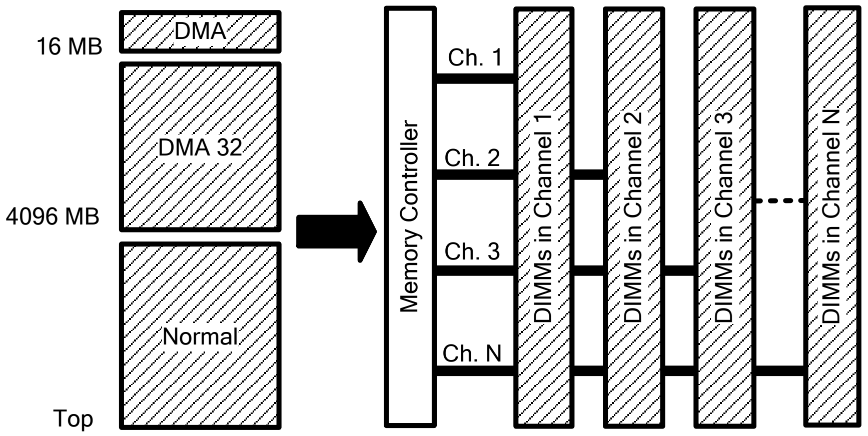


Figure 3.2: Linux physical address space zoning for x86-64. Zone boundaries do not necessarily fall between DIMM boundaries.

Each zone spans a contiguous linear physical address space (see Fig. 3.2). Not all addressable locations may be valid, however. The zone structure makes this distinction using "spanned" and "present" page counts, where the difference between the spanned pages and present pages indicates the number of page "holes". The zone structure also contains a pointer to its parent node structure as well as the start of the zone in the global *struct page* memory map (*mem_map*, an array of all physical page structures in the system). Zone structures have spin locks for controlling race conditions in the structure, a buddy allocator bitmap of free pages, hashed wait queues of processes waiting on pages, and watermarks indicating when free space is dangerously low (and when the *kswapd* daemon needs to wake up and free zone space).

The *struct zone* definition is shown in Listing 3.1, from Linux version 3.2 [49] for x86-64[2]:

```
struct zone {
    unsigned long           watermark[NR_WMARK];
    unsigned long           percpu_drift_mark;
    unsigned long           lowmem_reserve[MAX_NR_ZONES];
```

[1]Here, we refer to the binary meaning of megabyte, gigabyte, etc., not to be confused with the decimal meaning often used in non-volatile storage literature.

[2]All comments and some fields not shown for disabled kernel configuration features.

```
    struct per_cpu_pageset       __percpu *pageset;
    spinlock_t                   lock;
    int                          all_unreclaimable;
    struct free_area             free_area[MAX_ORDER];

#ifndef CONFIG_SPARSEMEM
    unsigned long                *pageblock_flags;
#endif

#ifdef CONFIG_COMPACTION
    unsigned int                 compact_considered;
    unsigned int                 compact_defer_shift;
#endif

    ZONE_PADDING(_pad1_)
    spinlock_t                   lru_lock;

    struct zone_lru {
        struct list_head         list;
    } lru[NR_LRU_LISTS];

    struct zone_reclaim_stat     reclaim_stat;
    unsigned long                pages_scanned;
    unsigned long                flags;
    atomic_long_t                vm_stat[NR_VM_ZONE_STAT_ITEMS];
    unsigned int                 inactive_ratio;
    ZONE_PADDING(_pad2_)

    wait_queue_head_t            * wait_table;
    unsigned long                wait_table_hash_nr_entries;
    unsigned long                wait_table_bits;


    struct pglist_data           *zone_pgdat;
    unsigned long                zone_start_pfn;

    unsigned long                spanned_pages;
    unsigned long                present_pages;

    const char                   *name;
} ____cacheline_internodealigned_in_smp;
```

Listing 3.1: Zone structure fields in Linux 3.2 for x86-64.

3.2.2 Pages

As mentioned previously, each zone in the kernel has a pointer to its first *struct page* in the global *mem_map* array. Each *struct page* contains information such as process reference count, lists to which it belongs (for bookkeeping), mappings (if part of some file or device memory map), status flags (such as dirty, swappable, on LRU list, used by slab allocator, etc.), LRU lists (it may be an active or non-active page, or neither), etc [50]. Pages can also link back to their parent zones but not via a direct pointer. Rather, an arithmetic offset macro is used, that operates on the high order bits of the page flags.

The *mem_map* array is not the page table for a process. Rather, it simply describes physical memory, with no knowledge of the overhead virtual memory mappings. However, page structures do contain information on the page table entries (PTEs) in which they are referenced (called PTE chains, managed by the slab allocator). This general technique is called "reverse mapping", or rmap. See Sec. 3.4.1 for more information on page tables.

The *struct page* definition is shown in Listing 3.2, from Linux version 3.2 [49] for x86-64:

```
struct page {
    unsigned long flags;
    struct address_space *mapping;

    struct {
        union {
            pgoff_t index;
            void *freelist;
        };
        union {
            unsigned long counters;

            struct {
                union {
                    atomic_t _mapcount;

                    struct {
                        unsigned inuse:16;
                        unsigned objects:15;
                        unsigned frozen:1;
                    };
                };
                atomic_t _count;
            };
        };
    };

    union {
        struct list_head lru;
        struct {
            struct page *next;
#ifdef CONFIG_64BIT
            int pages;
            int pobjects;
#else
            short int pages;
            short int pobjects;
#endif
        };
    };

    union {
        unsigned long private;

#if USE_SPLIT_PTLOCKS
        spinlock_t ptl;
#endif
        struct kmem_cache *slab;
        struct page *first_page;
    };

#if defined(WANT_PAGE_VIRTUAL)
    void *virtual;
#endif

#ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
    unsigned long debug_flags;
#endif

#ifdef CONFIG_KMEMCHECK
    void *shadow;
#endif
}
```

Listing 3.2: Page structure fields in Linux 3.2 for x86-64.

3.3 The Physical Page Allocator

We have discussed the data structures for representing physical memory. Now, we will turn to a discussion of the physical page allocator, which is used to get pages when higher level page management resources require them (e.g., the slab layer has run out of cached space, etc.). The physical page allocator relies on several important constructs, including zones, pages, page freelists, and constraint bitmasks.

3.3.1 The Buddy System

In order to mitigate the gradual fragmentation of physical memory, the kernel uses the "buddy system" algorithm [50]. When serving an allocation, it tries to find the largest contiguous block of pages (grouped in powers of two) that will serve the request. This is even done for requests that do not explicitly require contiguous physical memory, as there are performance benefits to spatial (physical) locality of data. When pages are freed, the kernel recursively recombines adjacent groups of free pages of the same size into larger groups.

This scheme has the benefit of quickly allocating a large chunk of contiguous pages for a request. The memory management system maintains a lookup table for number of available blocks for each block size in the buddy system, as well as page free-lists that allow a quick retrieval of a given page block.

However, the buddy system suffers from internal and external fragmentation [50]. External fragmentation is a problem that occurs when there is no block of contiguous pages that is large enough to serve an allocation request. Internal fragmentation, on the other hand, is defined by the wasted space when a large block must be used to service a small allocation request. Internal fragmentation is partially mitigated by the slab allocator mechanism [50], which acts a pre-allocated cache for small allocation requests or commonly used objects. External fragmentation is not as big as an issue, as requests for large groups of contiguous pages are relatively rare [50].

3.3.2 The Allocation Flow

As mentioned in Sec. 3.2.1, the rest of the memory space not claimed by the DMA or DMA32 zones (any of which may or may not be configured/supported) is left to the "Normal" zone. This space is permanently mapped into the kernel's virtual memory space. On x86-64, this will contain all memory above DMA and DMA32. Most allocations will land in this zone if possible, unless DMA/DMA32 support is explicitly required (described in the constraint bitmask that was mentioned earlier).

The kernel will try to fulfill a page allocation request in the most suitable zone first, but it can fall back to other zones if required. For example, a user application will typically have its memory allocated in *ZONE_NORMAL*; but if memory there is low, it will try *ZONE_DMA32* next, and *ZONE_DMA* only as a last resort. The kernel can also employ page migration and swap if required and permitted by the allocation constraints (if the request cannot allow I/O, filesystem use, or blocking, these might not apply). However, the reverse is not true. If a device driver needs DMA space, it MUST come from *ZONE_DMA* or the allocation will fail. For this reason, the kernel does its best to reserve these restricted spaces for these situations. Fig. 3.3 depicts the zone fallback order for page allocation.

3.4 Virtual Memory: Page Management

So far, we have covered the representation of physical memory in the kernel and how pages are actually allocated. Now we will turn to a discussion of page management in the kernel and how the virtual memory system interfaces with physical memory.

3.4.1 The Page Tables

The page table is essentially a translation mechanism to convert virtual addresses for a process to the actual physical addresses needed by the memory controller. Because virtual memory separates the virtual address space for each process (for program simplicity as well as inter-process protective barriers), each process must also have its own unique page table [51]. These tables are stored in the kernel address space so that user processes cannot modify their own or access others' tables, including that of the kernel. When context switching, the PGD pointer is set to the top of the process' own page table [50]. In general, page table loads are architecture-dependent; there is a special register *cr3* in x86 that supports this.

Linux uses a three-level nested page table structure in order to compact the memory requirements for the page table [50]. It can omit entire branches of the page table tree when no virtual pages in that address range have been allocated, thus allowing an efficient representation of sparse allocations. To facilitate this approach, the kernel can dynamically allocate structures for the page tables as necessary.

A linear address is converted into a page table lookup as follows [50]. The high order bits of the address index into the page global directory (PGD) for the current process. The next set of bits index into the page middle directory (PMD). The next lower order bits index into the page table entry (PTE) page frame, which returns the PTE of interest. The PTE then references a physical address for the page, and the least significant bits of the linear address specify the exact location within the physical page. The PTE can be used to derive its corresponding *struct page* (as mentioned earlier, they are maintained separately) and vice versa. See Fig. 3.4 for a visual depiction of the page table hierarchy [50].

A PTE is simply an unsigned integer, with a typedef to prevent misuse. Each PTE contains metadata about the virtual-to-physical page mapping. Bitwise flags that may be used include, but are not limited to:

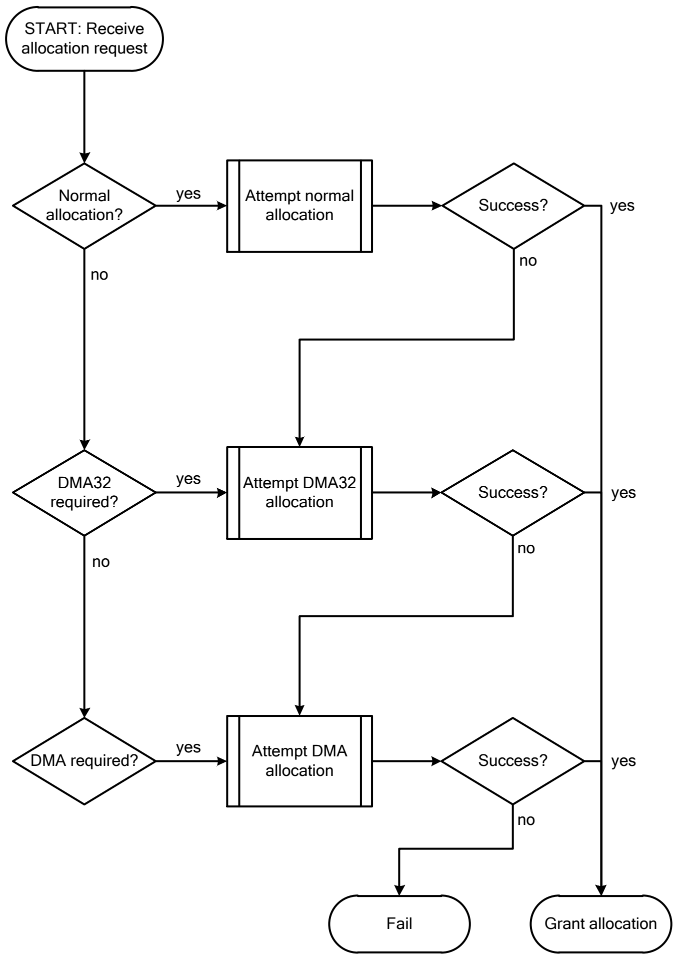


Figure 3.3: Linux x86-64 physical page allocation decision flow.

(1) whether the page is in memory or swap, (2) whether the page is present, but not accessible, (3) if the page is read-only, (4) whether the page is user-space accessible, (5) whether the page is dirty, (6) whether the page is accessed, etc.

Initializing the page tables at boot time is an architecture-dependent procedure [50]. After the kernel image is loaded at the first MB of physical memory and paging is bootstrapped, *paging_init()* is called which formally sets up the full page tables. Paging is then enabled by setting a bit in the *cr0* register, and a jump instruction is executed to set the instruction pointer to the correct virtual address to continue. Among other tasks, *paging_init()* includes zone initialization.

3.4.2 The Translation Look-aside Buffer (TLB)

The translation look-aside buffer (TLB) is used to cache page table lookups, benefiting from the principle of spatial locality. Like the L1 cache used for instructions and data, all virtual-to-physical translations are read out of the TLB by the processor. The TLB thus acts very similarly to an L1 cache, using a write-back approach to the process' page table.

When modifying kernel page tables, which are global in nature, the TLB and L1/L2/L3 caches must be flushed so that shared data remains synchronized [50]. The kernel uses architecture-dependent hooks for these operations, if the architecture does not automatically manage its caches (the x86-based architectures do).

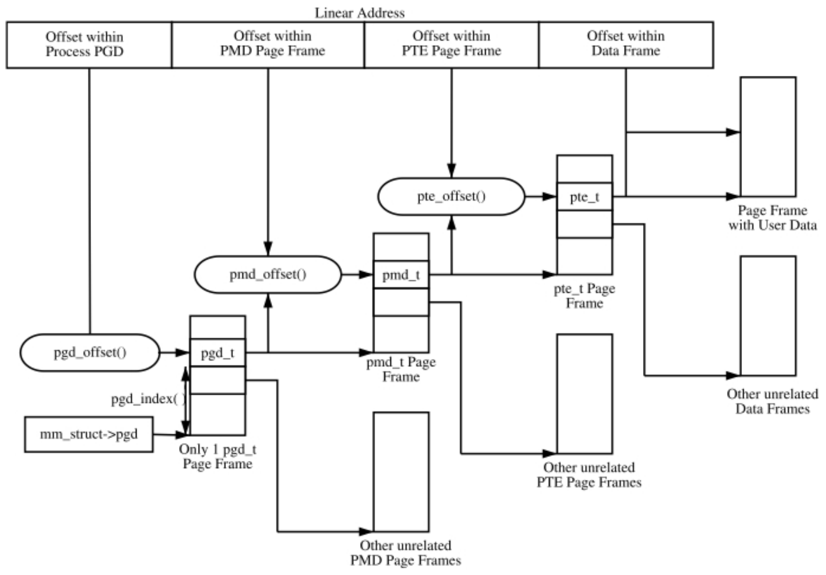

Linear Address
Offset within Process PGD
Offset within PMD Page Frame
Offset within PTE Page Frame
Offset within Data Frame
pte_offset()
pte_t
Page Frame with User Data
pmd_offset()
pmd_t
pte_t Page Frame
pgd_offset()
pgd_t
pmd_t Page Frame
pgd_index()
mm_struct->pgd
Only 1 pgd_t Page Frame
Other unrelated Data Frames
Other unrelated PTE Page Frames
Other unrelated PMD Page Frames

Figure 3.4: Page table hierarchy. From [50].

3.4.3 Page Replacement Policy

Because of the tremendous penalty of a page fault, page tables are typically fully associative, meaning that a physical page can be located anywhere in memory [51]. Furthermore, an OS keeps LRU/dirty/use bits to improve page replacement policy. For example, during a page fault, it is preferable to kick out an LRU page that is not dirty, as a dirty page would need to be written to swap space before it could be repurposed. An OS typically uses an LRU approximation per page. Periodically, it clears the "use bit". If a page is accessed, the bit is set. This coarsely partitions the space into LRU and non-LRU, which should be enough for page replacement during a page fault.

Linux uses two page lists that resemble an LRU scheme, called *active_list* and *inactive_list* [50]. When pages are allocated, they are placed onto the *inactive_list*. When a page is accessed, it has its reference bit set in the *struct page*. When a page reaches the tail of the list, the kernel checks its reference bit. If it is set, it goes back to the head of the list. If it is not set, the page is eligible for reclaim by *kswapd*. If a page on the *inactive_list* is referenced and the reference bit is already set, then the page is moved to the *active_list*, and the reference bit cleared.

On the active list, when a page is referenced, the reference bit is set as before. Periodically, a function *refill_active()* attempts to purge pages from the active list back to the inactive list. It pops pages off the tail of the active list. If the page has its reference bit set, then it is cleared and the page is moved to the head of the list. If it was not set, the page is bumped down to the top of the inactive list, but with its reference bit set, so that it can be quickly promoted back to the active list if needed. Fig. 3.5 depicts this process.

Essentially, the LRU-like scheme employed by Linux partitions pages into hot and cold lists. Cold pages are better candidates for reclaim, swap, etc. By using both lists and the reference bit, pages can be thought of having one of four possible tiers of activity: (1) inactive, non-referenced, (2) inactive, referenced, (3) active, non-referenced, and (4) active, referenced. Although hot/cold is principally used to decide which pages to reclaim, the four tiers allow pages to be more finely ranked by reference activity.

3.4.4 Page Migration

According to [49], there are two primary benefits to page migration that justified its kernel implementation. The first is NUMA support, where pages can be dynamically relocated to memory space near a processor that requires them, boosting performance. The second reason is for the allocation of huge pages, where pages can be relocated (defragmented) to make room, instead of reclaiming them.

The page migration mechanism is implemented as a system call (syscall) named *move_pages()* [49]. This

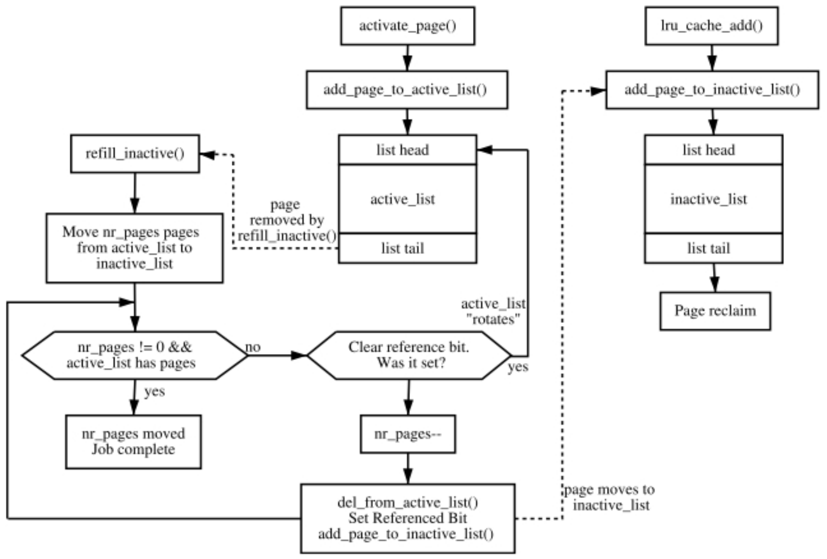

activate_page()
add_page_to_active_list()
list head
active_list
list tail
lru_cache_add()
add_page_to_inactive_list()
list head
inactive_list
list tail
Page reclaim
refill_inactive()
page removed by refill_inactive()
Move nr_pages pages from active_list to inactive_list
active_list "rotates"
nr_pages != 0 && active_list has pages
no
Clear reference bit. Was it set?
yes
yes
nr_pages moved Job complete
nr_pages--
del_from_active_list() Set Referenced Bit add_page_to_inactive_list()
page moves to inactive_list

Figure 3.5: Page LRU-like scheme. From [50].

enables applications to manually move pages between nodes to improve performance, depending on the particular processor the application is running on. Much of the page migration code is implemented in *mm/migration.c*, while the syscall itself is defined in *mm/mempolicy.c*.

As we are concerned only with the x86-64 SMP architecture, the page migration functionality is configured but not particularly relevant without a NUMA system.

3.4.5 The Slab Allocator – Handling Kernel Memory Allocations

The physical page allocator, based on the buddy system algorithm, suffers from internal and external fragmentation, as mentioned previously. The slab allocator serves the purpose of handling virtual memory allocation inside the kernel to minimize its impact on buddy system fragmentation and to maximize performance for frequently used objects [50].

Because the kernel must be frugal on its memory footprint, it cannot afford to waste precious memory due to fragmentation. Furthermore, many kernel objects are very heavily utilized, requiring frequent allocations, initializations, and deallocations. The slab allocator maintains a cache of commonly used objects and memory chunks in order to reduce the burden on the physical page allocator.

The slab allocator maintains caches for each type of relevant object, such as *mm_struct* that are used to represent regions of virtual memory. It also can cache buffers from 32 bytes up to 131,072 bytes for arbitrary dynamic kernel memory needs. Each cache tracks three lists of slabs: *slabs_full*, *slabs_partial*, and *slabs_free*. This hastens slab selection for serving an allocation request. Each slab maintains a list of contiguous pages which are carved up into the objects served by the cache. The objects are separated with “colored padding” that optimize the slabs for the hardware CPU caches [50]. See Fig. 3.6 for a visual depiction of the slab structure relationships.

The kernel services its own dynamic allocations using *kmem_cache_alloc()* for the allocation of objects from the slab, and *kmalloc()* for the allocation of generic sized buffers from the size-based slabs. The slab allocator is not used for user allocations. See Fig. 3.7 for an example of how the slab allocator is used by the kernel. It is important to remember that the even the slab allocator relies on the physical page allocator as its back-end.

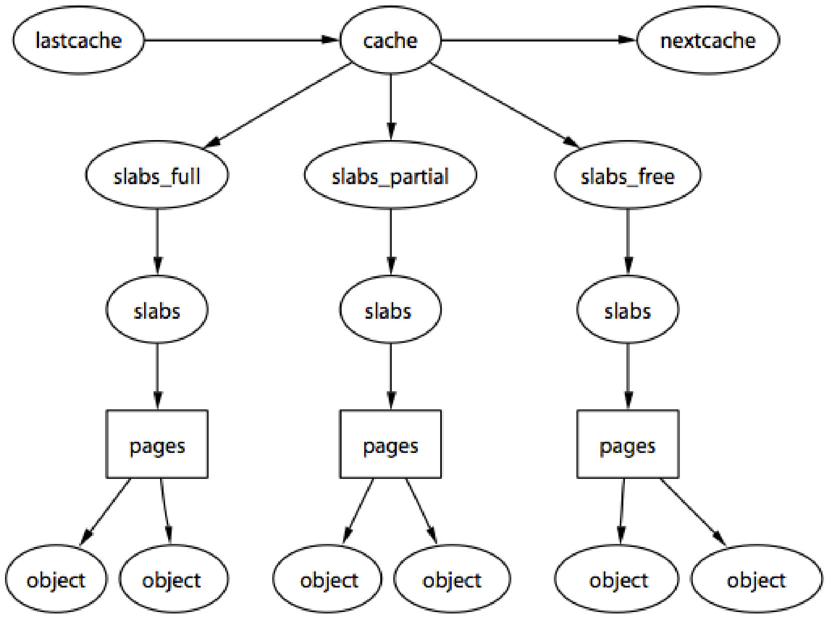


Figure 3.6: Slab allocator structures. From [50].

3.5 User-Space APIs to Allocate Virtual Memory

Above the virtual memory system implemented in the kernel discussed thus far in brief, the kernel provides an interface to userspace code to allocate memory. The kernel provides two main APIs, implemented as system calls (syscalls), that userspace can call to allocate memory dynamically. The programming language runtime (libraries) abstracts the syscall semantics to the user by providing an intermediate API. In C, this API consists of the *malloc()* family of function calls to get blocks of memory, while in C++, programmers should use the *new* operator to instantiate objects.

In this section, we will discuss a process' address space layout, the syntax and semantics of the two major syscalls, and then the *malloc()* implementation in GLIBC. We finish the section with a overview on how *malloc()* calls eventually reach the physical page allocator inside the kernel.

3.5.1 The Process Address Space

When a process is loaded into memory, there are several "segments" of address space that are used for different purposes [52]. On a traditional x86 32-bit system, the top 1 GB is reserved for the kernel (although we use an x86-64 system for our research, an example using a 32-bit system will suffice), while the lower 3 GB are used for the process[3].

At the bottom of the address space is the text segment, which contains the binary image of the process' code. Directly above it is the data segment, which contains global (static) variables initialized by the programmer. Directly adjacent is the BSS segment, which contains uninitialized global variables. Due to Address Space Layout Randomization (ASLR), there is a random address offset to the heap. The top of the heap is pointed to by the *brk* program break. Further up in the address space is the memory-mapped segment which contain device and file mappings (e.g. a portion of the GLIBC routines) or anonymous data, which are not backed by a file or device. Above the memory mapped segment is the stack, which grows downwards, and finally, at the top, is the kernel space. Fig. 3.8 depicts the layout of the user address space [52].

Unlike the stack, which grows automatically (with OS help) when a segmentation fault occurs, the heap and memory mapped segments must be explicitly expanded when dynamic memory is required.

[3]The kernel space is global to all processes, but is not accessible from userspace.

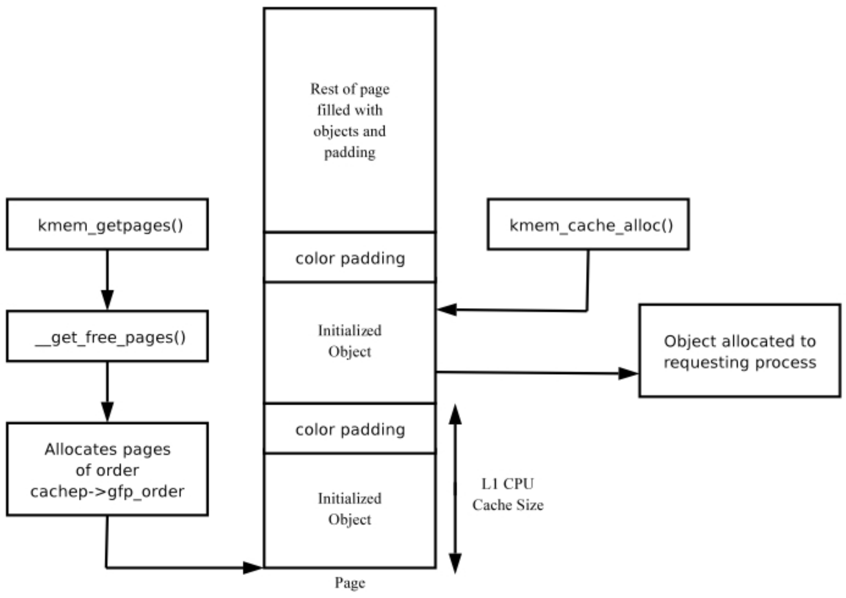


Figure 3.7: Slab allocator usage. From [50].

3.5.2 System Calls for Allocation

The kernel provides two primary syscalls for memory allocation to a user process, known as *sbrk()* and *mmap()*. *sbrk()* and its sister function *brk()* are used to grow or shrink the process' heap, while *mmap()* is used to allocate memory in the memory mapped segment for devices, files, or anonymous data. Note that the heap region must be contiguous in virtual space, as it is represented only by a starting address and a length. For this reason, fragmentation can become a major issue due to freed space in the middle of the heap.

The APIs for *sbrk()* and *mmap()* are described in Listings 3.3 and 3.4.

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

Listing 3.3: *sbrk()* System Call API, Wrapped by GLIBC [49]

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
//prot is used to set read, write, and/or execute permissions
//flags is used to set anonymous, file mapping, shared space, etc.
int munmap(void *addr, size_t length);
```

Listing 3.4: *mmap()* System Call API, Wrapped by GLIBC [49]

3.5.3 GLIBC *malloc()* Semantics and API

A user program in C should not invoke *sbrk()*, *brk()*, or *mmap()*. Instead, they should use *malloc()* or one of its related calls such as *calloc()* to handle allocations [49]. *malloc()* abstracts some of the complexities of using the system calls directly as well as improves allocation performance by attempting to reduce the number of syscalls invoked by a program. It does this by utilizing both *sbrk()* and *mmap()* syscalls to allocate virtual memory [49], usually requesting more memory than explicitly asked for by a program; in a sense, it maintains a user-level hidden cache.

The kernel man pages put it rather nicely: "Normally, *malloc()* allocates memory from the heap, and adjusts the size of the heap as required, using *sbrk(2)*. When allocating blocks of memory larger than

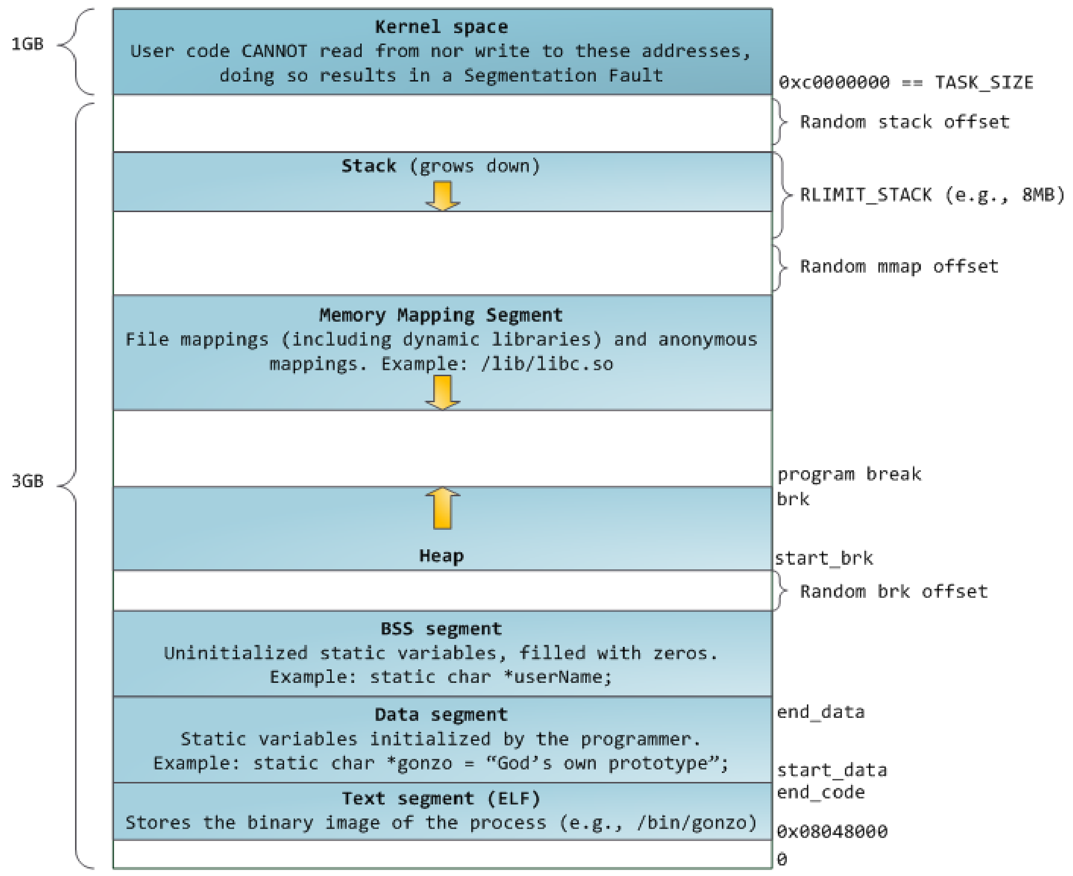

1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Figure 3.8: Process address space. From [52].

MMAP_THRESHOLD bytes, the glibc *malloc()* implementation allocates the memory as a private anonymous mapping using *mmap(2)*. *MMAP_THRESHOLD* is 128 kB by default, but is adjustable using *mallopt(3)*. Allocations performed using *mmap(2)* are unaffected by the *RLIMIT_DATA* resource limit (see *getrlimit(2)*)," [49].

The *malloc()* API is described in Listing 3.5.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Listing 3.5: *malloc()* GLIBC API [49]

3.5.4 How Requests for Virtual Memory Reach the Physical Allocator

When one calls *malloc()* from the GLIBC library, there is a good deal of complexity in userspace to try to satisfy the request without falling back to a syscall to the kernel. This is because syscalls are expensive and can lead to fragmentation, blocking, I/O, etc [53]. As mentioned before, *malloc()* uses pools of memory to satisfy smaller requests. In this way, it acts similarly to the slab allocator, discussed in Sec. 3.4.5. As a last resort, *malloc()* falls back to the *mmap()* and *sbrk()* syscalls discussed in Sec. 3.5.2.

The *mmap()* or *mmap2()* syscall is defined in the kernel as a wrapper for *sys_mmap_pgoff()* [49]. *sys_mmap_pgoff()* ends up calling *do_mmap_pgoff()* after some complicated syscall code, which is defined in *mm/mmap.c*. Similarly, *sbrk()* ends up at *do_mmap_pgoff()*.

Essentially, both *mmap()* and *sbrk()* end up "allocating" memory for the user, but this might not actually correspond to physical pages until a page fault occurs. In this sense, the kernel allocates physical memory lazily. Both syscalls end up storing allocation flags in the *vm_area_struct* that represents a virtual memory region for the process.

Chapter 4

Power Variability in DDR3 DRAMs

Technology scaling has led to significant variability in chip performance and power consumption. In this phase of the project, I measured and analyzed the power variability in DRAMs. The goal was to find the extent of power variability in contemporary devices, and gain insight as to the sources of variation, with the ultimate goal of applying the knowledge towards ViPZonE, which is discussed in Chapter 5. I tested 22 DDR3 DIMMs, and found that power usage in DRAMs is dependent both on operation type (write, read, and idle) as well as data, with write operations consuming more than reads, and 1s in the data generally costing more power than 0s. Temperature had little effect (1-3%) across the -50 °C to 50 °C range. Variations were up to 12.29% and 16.40% for idle power within a single model and for different models from the same vendor, respectively. In the scope of all tested 1 gigabyte (GB) modules, deviations were up to 21.84% in write power.

4.1 Test Methodology

I now describe the test methodology for this phase of the project, including the instrumentation setup, DIMM power components that were measured, and the selection of memory devices.

4.1.1 Instrumentation Setup

Each DDR3 DIMM used a 240-pin connector to interface with the motherboard. I inserted a "DIMM riser" device between the DIMM and the motherboard, which essentially is a pass-through bridge for most of the DIMM's electrical connections. In order to measure power consumption of a DIMM, there is a current sensing resistor (CSR) inserted on the VDD lines going to the DIMM. This is done by tying all motherboard-to-riser VDD pins together into one net, passing all VDD current through the 0.02 Ohm CSR, and then tying the "output" side of the CSR to all riser-to-DIMM VDD pins (one common net). This has no functional consequence for the DIMM. This power measurement scheme is often referred to as "high-side" current sensing. Table 4.1 summarizes the specifications of the DIMM riser modules.

Fig. 4.1 depicts how the DIMM riser is used with the DIMM. It is inserted between the module and the motherboard, with most pins having a directly mapped relationship (that is, pin X on the motherboard side is shorted to pin X on the DIMM side of the riser). Fig. 4.2 depicts how VDD current flows through the CSR for power measurement purposes.

4.1.2 DIMM Power Components Measured

As I only measured DIMM power through the VDD pins, I did not account for any on-die-termination (ODT) power. However, all power spent by the DIMM when driving the memory bus (e.g., completing a read operation) is accounted for by measuring VDD current. My measurement method does *not* account for any power consumed by the DIMM when another DIMM or the memory controller is driving the bus. Because I measured the power on all DIMMs in the system, this means the only power not accounted for in

Spec	**Value**
Manufacturer	Adex Electronics
Website	www.adexelec.com
Model	DDR3-L riser card with CSR option
Pins	240
Voltage	1.5 V
CSR Value	0.02 Ohms
CSR Tolerance	1%
CSR Power Rating	0.5 W
CSR Net	VDD (high-side)

Table 4.1: DIMM riser specifications.

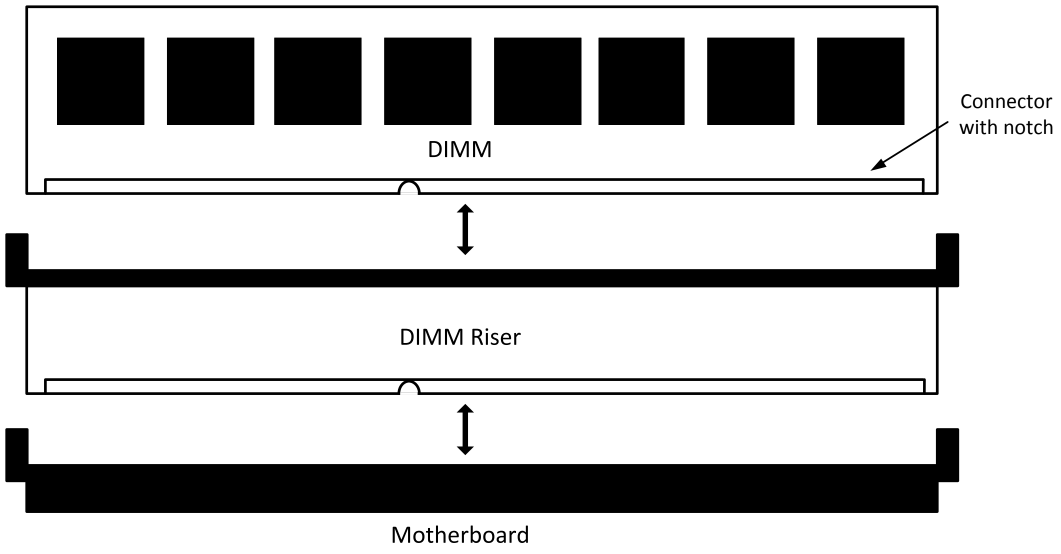


Figure 4.1: DIMM riser application.

our scheme is that of the memory controller. Please refer to Micron's "Calculating Memory System Power for DDR3" for a detailed power breakdown of a Micron DDR3 SDRAM device [54].

4.1.3 Memory Equipment

The DIMMs were comprised of several models from four vendors (see Table 4.2), manufactured in 2010 and 2011 (the particular process technologies are unknown). For five of the DIMMs, I could not identify the DRAM suppliers. Most models were 1 GB[1] DDR3 modules, rated for 1066 MT/s (except for the Vendor 4 models, rated for 1800 MT/s) with a specified supply voltage of 1.5 V. We also included three 2 GB specimens from Vendor 1 to see if capacity had any effect on power consumption. The DIMMs are referred to henceforth by abbreviations such as V1S1M1 for Vendor 1, Supplier 1, Model 1.

4.1.4 Test Platform & Data Acquisition

The test platform utilized an Intel Atom D525 CPU running at 1.8 GHz, running on a single core. Only one DIMM was installed at a time on the motherboard, and all other hardware was identical for all tests.

[1]To avoid confusion in terminology, I refer to the gigabyte (GB) in the binary sense, i.e., 1 GB is 2^{30} bytes, not 10^9 bytes.

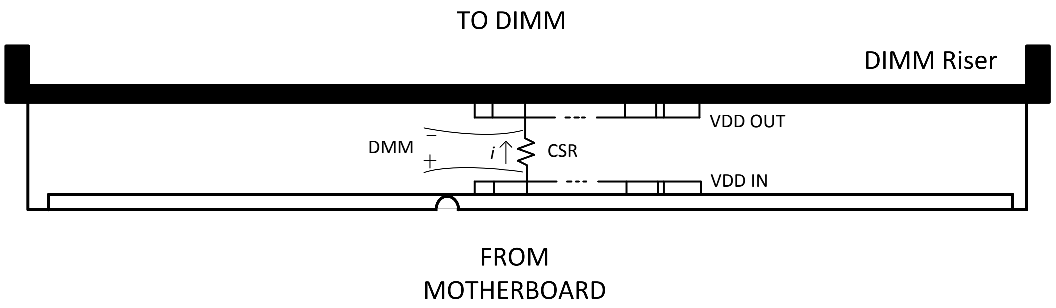


Figure 4.2: DIMM riser high-side current sensing scheme.

Table 4.2: DDR3 DIMM Selection

Category	**Quantity**
Vendors	4 (V1-V4)
Suppliers	3 known (S1-S3, SU)
Capacities	1 GB (V1-V4), 2 GB (V1 only)
Models	Up to 3 per vendor, 8 total (7 were 1 GB)
Total DIMMs	22 (19 were 1 GB)

No peripherals were attached to the system except for a keyboard, VGA monitor, and a USB flash drive containing the custom test routines. An Agilent 34411A digital multimeter sampled the voltage at 10 ksamples/s across the DIMM riser CSR, and this was used to derive the power consumption. Ambient temperature was regulated using a thermal chamber.

Because I required fine control over all memory I/Os, I developed custom modifications to Memtest86 v3.5b, which is typically used to diagnose memory faults [55]. The advantage of using Memtest86 as a foundation was the lack of any other processes or virtual memory, which granted me the flexibility to utilize memory at a low level.

I created a function which wrote memory sequentially with a specified bit pattern, but never read it back. Similarly, a read function was created which only read memory sequentially without writing back. Each word location in memory could be initialized with an arbitrary pattern before the executing the read test. The bit fade test – which was originally designed to detect bit errors over a period of DRAM inactivity – was modified to serve as an idle power test, with minimal memory usage.[2] For all tests, the cache was enabled to allow for maximum memory bus utilization. With the cache disabled, we observed dramatically lower data throughput and were unable to distinguish power differences between operations. As the intent was primarily to measure power variability between different modules, I used sequential access patterns to avoid the effects of caches and row buffers.

Each test was sampled over a 20 second interval, during which several sequential passes over the entire memory were made. This allowed us to obtain the average power for each test over several iterations. Each reading had an estimated accuracy of 0.06 mV [56], which corresponds to approximately 4.5 mW assuming a constant supply voltage and resistor value. Table 4.3 summarizes the important test environment parameters.

4.2 Experimental Results

4.2.1 Data Dependence of Power Consumption

Since DRAM power consumption is dependent on the type of operation as well as the data being read or written [39], I conducted experiments to find any such dependencies. Note that the background, pre-charge,

[2]Although there are different “idle” DRAM states, they are not directly controllable through software; I did not distinguish between them.

Table 4.3: Testbed and Measurement Parameters

Parameter	**Value**
Testbed CPU	Intel Atom D525 @ 1.8 GHz
Number of CPU Cores Used	1
Cache Enabled	Yes
DIMM Capacities	1 GB, (2 GB)
DIMM Operating Clock Freq.	400 MHz
Effective DDR3 Rate	800 MT/s
DIMM Supply Voltage	1.5 V
Primary Ambient Temp.	30 °C
Secondary Ambient Temp.	-50, -30, -10, 10, 40, 50 °C
Primary Memory Test Software	Modified Memtest86 v3.5b [55]
Custom Test Routines	Seq. Write Pattern, Seq. Read, Idle
Digital Multimeter	Agilent 34411A
Sampling Frequency	10 ksamples/sec
Reading Accuracy	approx. 4.5 mW
Number of Samples Per Test	200000

and access power consumed in a DRAM should have no dependence on the data [54]. Note that this test is similar to one performed on SRAMs in [36].

Seven tests were performed on four DIMMs, each from a different vendor, at 30°C to explore the basic data I/O combinations. The mean power for each test was calculated from the results of the four DIMMs. Fig. 4.3 depicts the results for each test with respect to the idle case ("Write 0 over 0" refers to continually writing only 0s to all of memory, whereas "Write 1 over 0" indicates that a memory full of 0s was overwritten sequentially by all 1s, and so on). Note that for the idle case, there was negligible data dependence, so we initialized memory to contain approximately equal amounts of 0s and 1s.

Interestingly, the power consumed in the operations was consistently ordered as seen in Fig. 4.3, with significant differences as a function of the data being read or written. There was also a large gap in power consumption between the reading and writing for all data inputs.

I presume that the difference between the write 0 over 0 case and the read 0 test is purely due to the DRAM I/O and peripheral circuitry, as the data in the cell array is identical. This would also apply to the write 1 over 1 case and its corresponding read 1 case. In both the read 1 and write 1 over 1 cases, more power was consumed compared to the corresponding read 0 and write 0 over 0 cases. These deltas may be due to the restoration of cell values. Because a sense operation is destructive of cell data due to charge sharing [39], cells that originally contain 1s must be restored using additional supply current. In contrast, cells containing 0s need only be discharged.

Note that the write 0 over 1 test consumed *less* power than the write 0 over 0 test, whereas the write 1 over 0 case consumed *more* than the write 1 over 1 case. The write 1 over 0 case likely consumes the most power because the bit lines and cells must be fully charged from 0 to 1. In the write 0 over 1 case, it probably uses the least power because the bit lines and cells need only be discharged to 0. Further research and finer-grained measurement capabilities are required to fully explain these systematic dependencies. Nevertheless, these results indicate strong data and operation dependence in DRAM power consumption.

Because of the data dependencies in write and read operations, I decided to use memory addresses as the data for write and read in all subsequent tests, because over the entire address space, there are approximately equal quantities of 1s and 0s. Furthermore, memory addresses are common data in real applications. I verified that the average write and read power using addresses for data is approximately the same as the mean of the values for 1s and 0s as data.

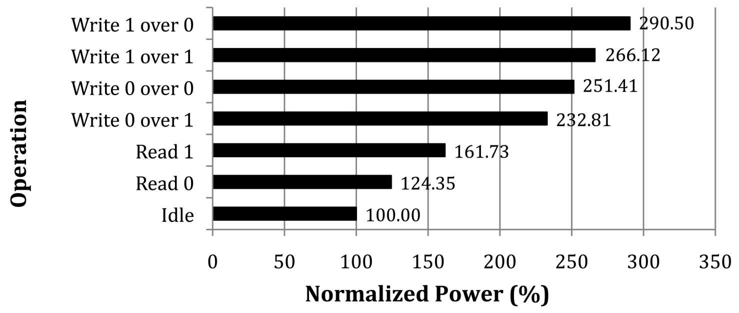


Figure 4.3: Data and operation dependence of DIMM power.

4.2.2 Temperature Effects

To determine if temperature has any effect on memory power consumption, I tested four 1 GB modules, one from each vendor. Each DIMM was tested at ambient temperatures from -50 °C to 50 °C.[3] It is clear from Fig. 4.4 that temperature had a negligible effect on power consumption even across a large range. I speculate that this is partially due to the area and leakage-optimized DRAM architecture [5], but more substantially affected by modern refresh mechanisms. The use of rolling refreshes or conservative timings may consume significant dynamic power, overshadowing the temperature dependent components in the background power consumption. Since no DIMM exhibited more than 3.61% variation across a 100 °C range, all further tests were performed at an ambient temperature of 30 °C.

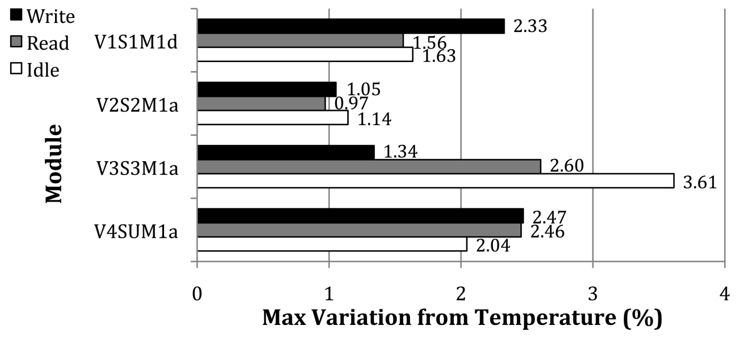


Figure 4.4: Relative temperature effects on write, read, and idle DIMM power, -50 °C to 50 °C range.

4.2.3 DIMM Power Variations

A plot of write, read, and idle power consumption for all 22 DIMMs at 30 °C is depicted in Fig. 4.5. The variability results are summarized in Fig. 4.6.

Variability Within DIMMs of the Same Model (1 GB)

Consider a particular model – V1S1M1 in Fig. 4.5 – of which we had the largest number (five) of specimens. While there was a maximum of 12.29% difference between the five DIMMs, there is a visible gap between the first group of three DIMMs and the second group of two (fourth and fifth in Fig. 4.5). This may be because the DIMMs come from two different production batches, resulting in lot-to-lot variability. The maximum deviations within the first group was only 1.34% for idle, and 1.47% within the second group. This suggests that the majority of the variation in V1S1M1 was between the two batches.

[3]Testing above an ambient temperature of 50 °C was not practical as it caused testbed hardware failure.

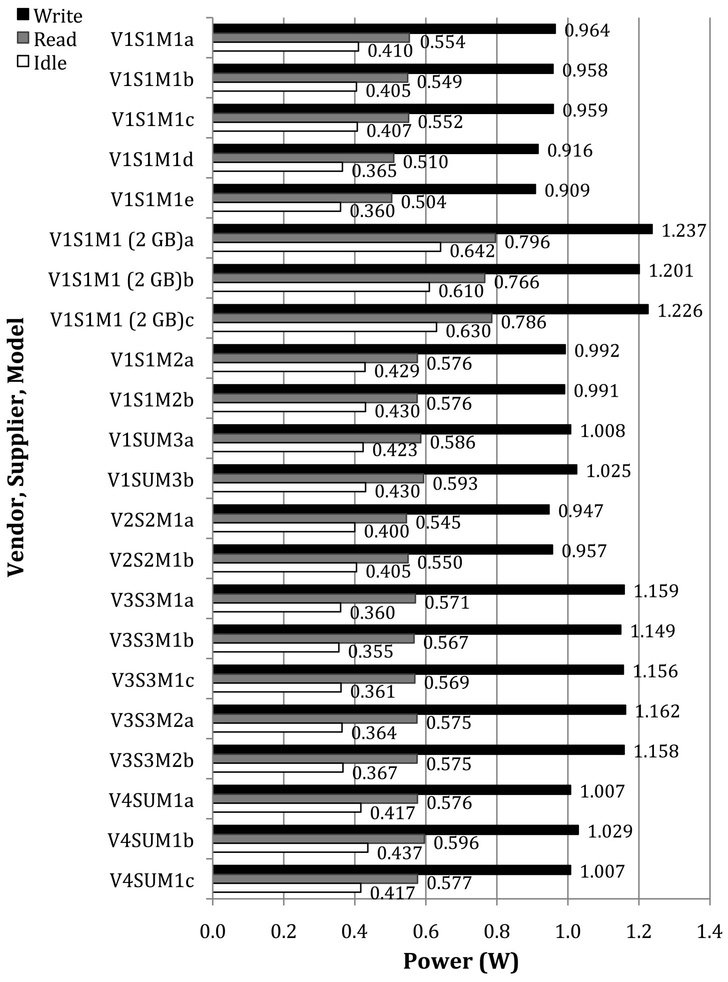


Figure 4.5: Write, read, and idle power by DIMM, 30 °C.

Variability Between Models of the Same Vendor/Supplier (1 GB)

Now consider all DIMMs from Vendor 1. We would expect that there would be more variation in Vendor 1 overall than in V1S1M1 only, and this was confirmed in the data. The maximum variation observed in Vendor 1 (1 GB) was 16.40% for the idle case. This variability may be composed of batch variability or performance differences between models.

Variability Across Vendors (1 GB)

In order to isolate variability as a function of vendors and to mitigate any effects of different sample sizes, we computed the mean powers for each vendor (1 GB). Vendor 3 consumed the most write power at 1.157 W. The variations for write, read, and idle power were 17.73%, 6.04%, and 14.65% respectively.

Overall Variability Amongst 1 GB DIMMs

As one may have expected, the variations across all DIMMs were significantly higher than within a model and among vendors, with the maximum variation occurring for write power at approximately 21.84%.

Effects of Capacity on Power Consumption

It is clear from Fig. 4.5 that the three 2 GB DIMMs of V1S1M1 consumed significantly more power than their 1 GB counterparts. This was expected, as there was bound to be higher idle power with twice as many DRAMs (in two ranks instead of one). Indeed, the maximum variation between the 2 GB and 1 GB versions was 37.91%, which occurred for idle power, whereas write power only differed by half as much. This is because background power is a smaller proportion of overall power when the DIMM is active.

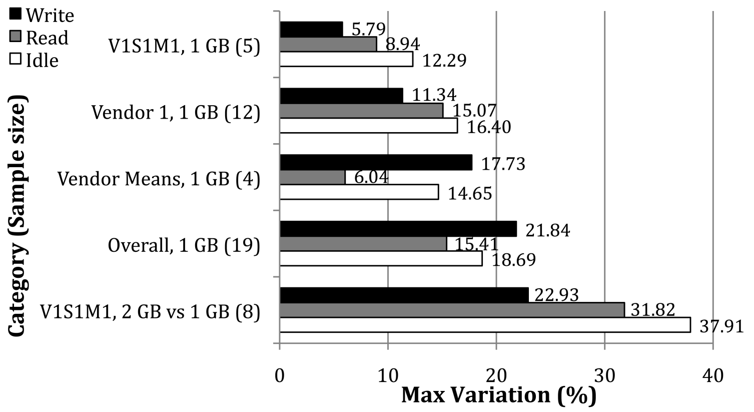


Figure 4.6: Max. variations in write, read, and idle power by DIMM category, 30 °C.

4.3 Summary

I analyzed the power consumption of several mainstream DDR3 DIMMs from different vendors, suppliers, and models, and found several important trends. Firstly, I did not find any significant temperature dependence of power consumption. Manufacturing process induced variation (i.e., variation for the same model) was up to 12.29%. Among models from the same vendor, idle power generally varied the most (up to 16.40% among Vendor 1), followed by read and write power. However, a different trend was evident across vendors, with write power varying the most (up to 17.73%), followed by idle and read power. This pattern was dominant overall amongst all tested 1 GB DIMMs, where I observed up to 21.84% variations in write power. Lastly, I found that a 2 GB model consumed significantly more power than its matching 1 GB version, primarily due to its increased idle power (up to 37.91%). Data-dependence of power consumption was also very pronounced, with about 30% spread within read and 25% spread within write operations. These findings serve as a motivation for variability-aware software optimizations to reduce memory power consumption. In an arbitrary set of DIMMs, there can be considerable variation in power use, and an adaptable system can use this to its advantage. Because I observed negligible temperature dependence, I do not include it in our ViPZonE models of DIMM power.

Chapter 5

ViPZonE: Power Variability-Aware Memory Management

This chapter builds upon the previous three, by describing ViPZonE, a novel DRAM power-variability aware modification for the Linux kernel.

5.1 Exploiting DRAM Power Variation

Before discussing the ViPZonE implementation, I briefly review the nature of DRAM power variability. In this work, I optimized for variability measured at the DIMM level, as it is the smallest piece of memory that is user-replaceable in a typical desktop, server, or laptop. Fig. 5.1 depicts the measured power variation in a set of eight identically specified (voltage, clock frequency, timings, capacity, etc.) off-the-shelf DDR3 DIMMs. Each DIMM's write, read, and idle power was characterized using a memory testing routine (described in Chapter 4). These power deviations arise purely from vendor implementations and manufacturing process variation. Note that using DIMMs from different manufacturers in the same system may be common in a situation where there are many memories, and/or when DIMMs need to be replaced over time due to permanent faults (e.g., in datacenters). Moreover, the variability among DRAMs is expected to increase in the future [10], especially if variation-aware voltage scaling techniques are used, such as those proposed by [57].

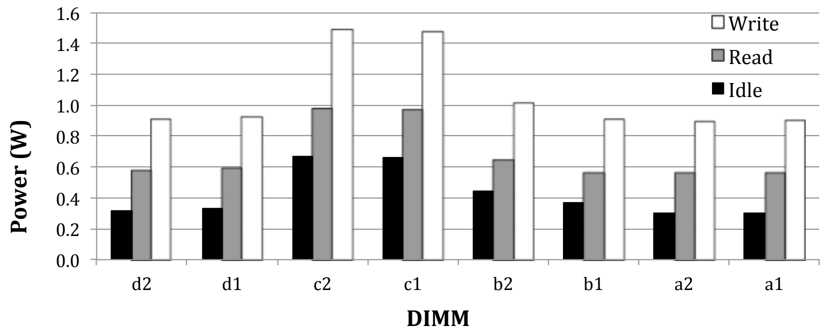


Figure 5.1: Measured power variations in eight identically specified off-the-shelf DDR3 DIMMs, using methods from Chapter 4. Letters denote different DIMM models, and numbers indicate instances of a model.

In a variability-aware memory management scheme, the upper-bound on power savings is determined by the extent of power variation across the memories in the system. For example, if we assume interleaving to be disabled, the worst case for power consumption would be when the DIMM with the highest power contains all the data that is accessed, while the rest are idle. The best case would be where all of this data is on the lowest-power DIMM. In a case where data is spread evenly across all DIMMs, no power variation can be exploited and the result is similar to that if interleaving were used.

In a multi-programmed system where only a portion of the physical memory is occupied, then we can intelligently save energy. If most of the memory is occupied and accessed frequently, it is harder to exploit the power variations, but as long as there is some non-uniformity in the way different pages are accessed, it remains possible. I believe that the former scenario is a good case for our study, as any system where the physical memory is fully occupied will suffer from large performance bottlenecks due to disk swapping. When this happens, memory power and performance are no longer a first-order concern. Thus, I believe the latter scenario to be less interesting from the perspective of a system designer when considering power variability-aware memory management.

In the testbed, interleaving prevents the exploitation of any power or performance variability present in the memory system. When striping accesses across different devices, the system runs all the memories at the speed of the slowest device, thus potentially sacrificing performance of faster modules, and preventing opportunistic use of varied power consumption. Interleaving on the testbed is also inflexible: it is statically enabled/disabled (cannot be changed during runtime), and it could also incur power penalties from the prevention of deeper sleep modes on a per-DIMM basis.

As interleaving and ViPZonE are mutually exclusive on the testbed, I provide an evaluation of power, performance, and energy for different interleaving modes in Sec. 5.3.2. I will discuss possible solutions to allow interleaving alongside variability-aware memory management in Chapter 6.

5.2 ViPZonE Implementation

ViPZonE is composed of several different components in the software stack, depicted in Fig. 5.2, which work together to achieve power savings in the presence of DIMM variability. I refer to the lower OS layer as the "back-end" and the application layer along with the upper OS layer as the "front-end". These are described in Sec. 5.2.2 and Sec. 5.2.3, respectively. ViPZonE uses source code annotations at the application level, which work together with a modified GLIBC library to generate special memory allocation requests which indicate the expected use patterns (write/read dominance, and high/low utilization) to the OS[1]. Inside the back-end Linux memory management system, ViPZonE can make intelligent physical allocation decisions with this information to reduce DRAM power consumption. By choosing this approach, I am able to keep overheads in the OS to a minimum, as I place most of the burden of power-aware memory requests on the application programmer. With this approach, no special hardware support is required beyond software-visible power sensors or pre-determined power data that is accessible to the kernel.

There are alternative approaches to implementing power variation-aware memory management. One method could avoid requiring a modified GLIBC library and application-level source annotations by having the kernel manage all power variation-aware decisions. However, such an approach would place the burden of smarter physical page allocations on the OS, likely resulting in a significant performance and memory overhead. Furthermore, the kernel would be required to continuously monitor applications' memory accesses with hardware support from the memory controller. Nevertheless, ViPZonE's layered architecture means that implementing alternate memory management strategies could be done without significant changes to the existing framework. I leave the study of these alternative methods to future work.

5.2.1 Target Platform and Assumptions

This work targets generic x86-64 PC and server platforms that run Linux and have access to two or more DIMMs exhibiting some amount of power variability (ViPZonE cannot have a benefit with uniform memory power consumption). If device-level power variation is available, then this approach could be adapted to finer granularities, depending on the memory architecture. I 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). It could detect off-chip memory power variation, obtained by one of the

[1]This scheme does not currently support kernel memory allocations (e.g., *kmalloc()*). As the kernel space is generally a small proportion of overall system memory footprint, and invoked by all applications, I statically place the image and all dynamic kernel allocations in the low power zone.

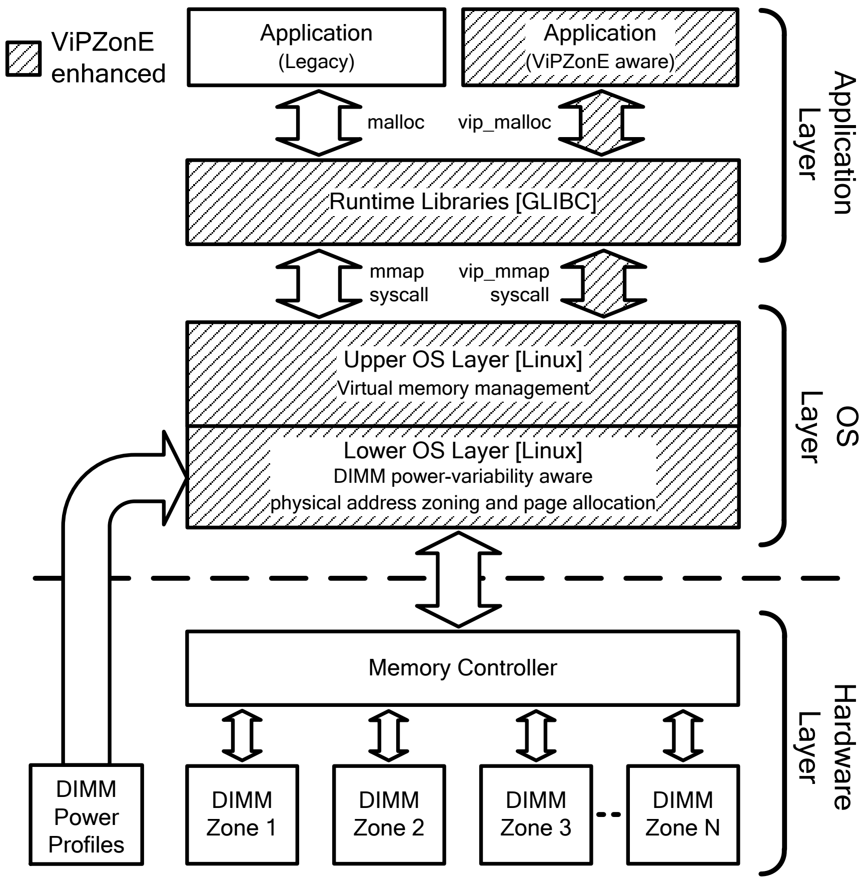


Figure 5.2: Layered architecture of ViPZonE.

following methods: (1) 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 dependent on process variations, and weakly dependent on temperature (see Chapter 4), there is little need for real-time monitoring of memory power use for each module. However, if power variation changes slowly over time (e.g., due to aging and wear-out occuring over time much greater than the uptime of the system after a single boot), I assume these changes can be detected through power sensors in each module.

- We can perform direct mapping of the address space[2] (e.g., select a single DIMM for each read/write request). This is achieved by disabling rank and channel interleaving on the testbed. I verified the direct mapping of addresses to specific DIMMs. Note that the particular division of DIMMs into ranks and channels is not a primary concern to ViPZonE; the only requirement is that only one DIMM is accessed per address.

- Programmers have a good understanding of the memory access patterns of their applications, obtained by some means, e.g., trace-driven simulations, allowing them to decide what dynamic memory allocations are "high utilization", or write-dominated, etc. Of course, in other scenarios, ViPZonE allows for annotation-independent policies.

[2]Note that address space layout randomization (ASLR) is not an issue, as ViPZonE deals with physical page placement only, while ASLR modifies the location of virtual pages.

5.2.2 Back-End: ViPZonE Implementation in the Linux Kernel

I discuss the implementation of ViPZonE in a bottom-up manner, in a similar fashion to the background material presented earlier. The implementation is based on the Linux 3.2 and GLIBC 2.15 source for x86-64.

Enhancing Physical Memory Zoning to Exploit Power Variability

In order to support memory variability-awareness, the ViPZonE kernel must be able to distinguish between physical regions of different power consumption. With knowledge of these power profiles, it constructs 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 Sec. 5.2.3).

In the ViPZonE kernel for x86-64, I explicitly removed the Normal and DMA32 zones, while still allowing for DMA32 allocation support. Regular DMA-able space is retained. 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 (i.e., addresses under 4 GB). Fig. 5.3 depicts the modified memory zoning scheme for the back-end. For example, in a 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 (I assume that each DIMM can have different power consumption):

- *Zone DMA*: 0-16 MB, mapped to *DIMM 1*.
- *Zone 1*: 16-2048 MB, mapped to *DIMM 1*.
- *Zone 2*: 2048-4096 MB, mapped to *DIMM 2*.
- *Zone 3*: 4096-6144 MB, mapped to *DIMM 3*.
- *Zone 4*: 6144-8192 MB, mapped to *DIMM 4*.

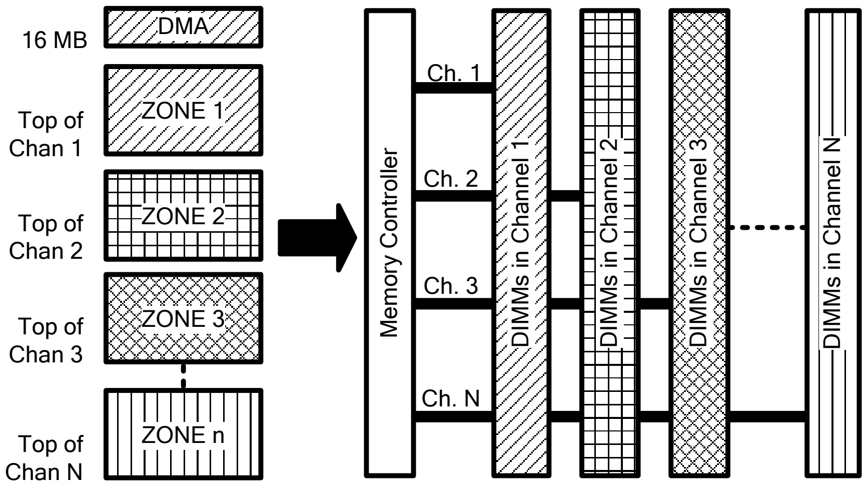


Figure 5.3: ViPZonE modifications to Linux physical address space zoning for x86-64. The kernel maintains separate zones which correspond directly to different physical DIMMs.

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

With zones set up for each DIMM, and knowledge of the relative power consumption of each DIMM, the kernel has the essential tools it needs to make power variability-aware page allocations, whereas the vanilla kernel makes no distinction between modular boundaries. There are many possible physical page allocation policies that could be used in the ViPZonE framework.

The ViPZonE kernel makes a distinction between relative write and read power for each DIMM zone. This is done for the hypothetical case where a module that has the lowest write power may not have the lowest read power, etc. Furthermore, it allows for future applicability to non-volatile memories, such as the MTJ-based family (MRAM, STT-RAM, MeRAM) with large differences in read and write power and energy [58].

The default policy that I implemented is *Low Power First.* The policy is currently configurable at kernel compile-time, but could be changed to work on-the-fly if policies need to be changed at runtime. In this policy, for high-utilization allocations, the kernel tries to get the lowest read or write power zone available. For low-utilization requests, the kernel still tries to fulfill it in a low read or write power zone, as long as some free space is reserved for high-utilization requests. Allocations requiring DMA32-compatible space are restricted to zones less than 4 GB, but otherwise follow the same utilization-priority rules. Finally, legacy DMA requests (less than 16 MB) always are granted in *Zone DMA*. The *Low Power First* policy is depicted in Fig. 5.4.

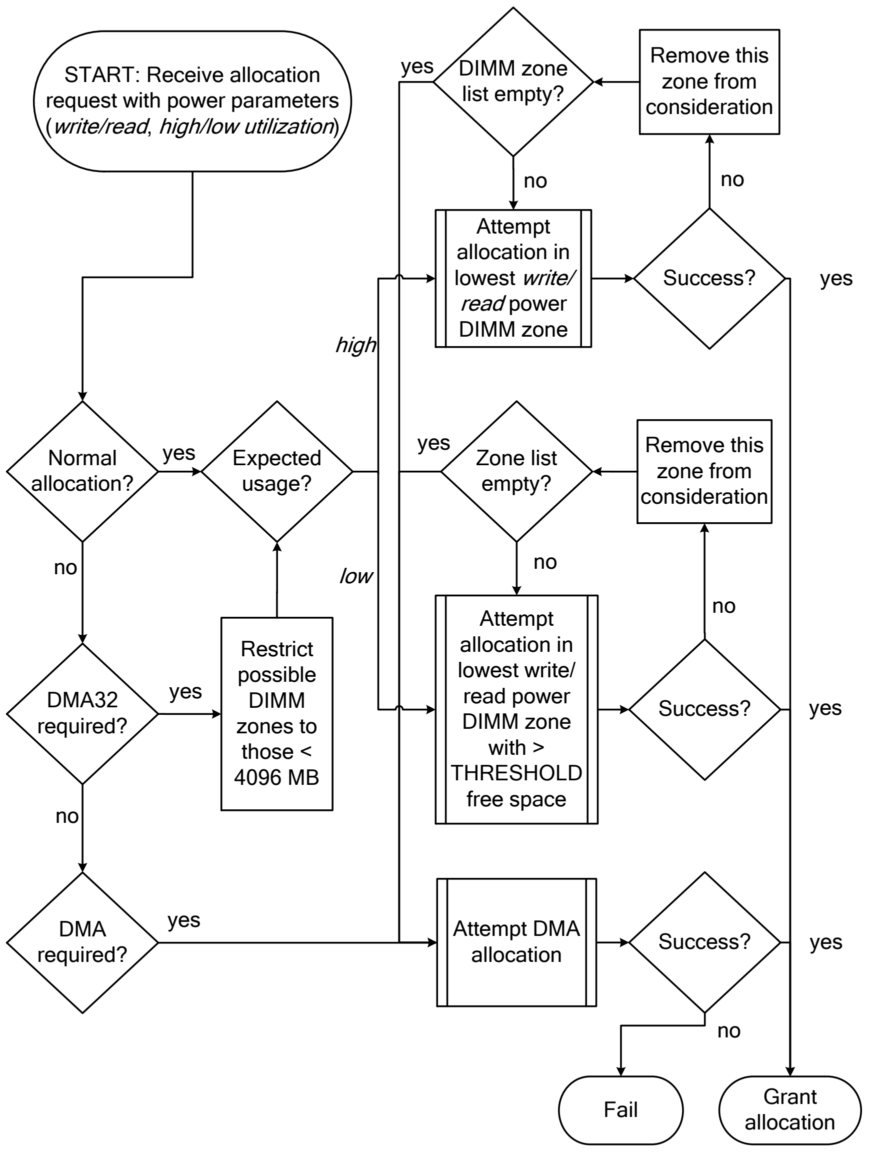

START: Receive allocation request with power parameters (*write/read*, *high/low utilization*)
Normal allocation?
yes
no
Expected usage?
high
low
DMA32 required?
yes
no
Restrict possible DIMM zones to those < 4096 MB
DMA required?
yes
DIMM zone list empty?
yes
no
Remove this zone from consideration
Attempt allocation in lowest *write/ read* power DIMM zone
Success?
yes
no
Zone list empty?
yes
no
Remove this zone from consideration
Attempt allocation in lowest write/ read power DIMM zone with > THRESHOLD free space
Success?
yes
no
Attempt DMA allocation
Success?
yes
no
Fail
Grant allocation

Figure 5.4: *Low Power First* default ViPZonE memory allocation policy in Linux x86-64.

For example, in a system with four DIMMs, each with 2 GB of space, the ViPZonE kernel would make allocation decisions as follows:

- *Request for DMA-capable space*: Grant in *Zone DMA*.
- *Request for DMA32-capable space (superset of DMA)*: Restrict possibilities to *Zone 1* or *Zone 2*. If indicated utilization is *low*, choose the lower *write* (if indicated) or *read* (if indicated, or default) power zone, as long as at least *THRESHOLD* free space is available (generally, we choose *THRESHOLD* to be approximately 20% of DIMM capacity). If indicated utilization is *high*, always choose the lower power (*write/read*) zone if possible. If neither zone has enough free space, the kernel uses the vanilla page reclamation mechanisms or can default to *Zone DMA* as a last resort.
- *Request for Normal space*: Grant in any zone, with the order determined in the same fashion as the above case for DMA32, without the 4 GB address restriction. *Zone DMA* is only used as a last resort.

Alternatively, more sophisticated policies could use a variety of tricks. For example, a kernel which tracks page accesses might employ page migration to promote highly utilized pages to low-power space while demoting infrequently used pages to high power space. However, this would need to be carefully considered, as the performance and power costs of tracking and migrating pages might outweigh the benefits from exploiting power variation. I leave the study of these policies to future work.

5.2.3 Front-End: User API and System Call

The other major component of the ViPZonE implementation is 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 provide hints regarding intended use for memory allocations, so that the kernel can prioritize low power zones for frequently written or read pages. The GNU standard C library (GLIBC) was used to implement the power variation enhanced allocation functions as part of the standard library (source code available at [38]). I briefly describe the methods and their use.

I added two new features to the applications' API, described in Table 5.1, allowing the programmer to indicate to the virtual memory manager the intended usage. I implemented a new GLIBC function, *vip_malloc()*, as a new call to enable backwards compatibility with all non-ViPZonE applications requiring the use of vanilla *malloc()*. *vip_malloc()* is essentially a wrapper for a new syscall, *vip_mmap()*, that serves as the hook into the ViPZonE kernel. While a pure wrapper for a syscall is not ideal due to performance and fragmentation reasons, I found it sufficient to evaluate this scheme. *vip_malloc()* can be improved further to implement advanced allocation algorithms, such as those in various C libraries' *malloc()* routines.

Because low power memory space is likely to be precious, memory should be released to the OS as soon as possible when an application no longer needs it. As a result, I preferred the use of the *mmap()* syscall over *sbrk()*, which has the heap grow contiguously. With *sbrk()*, it is often the case that memory is not really freed (i.e., usable by the OS). For this reason, a ViPZonE version of the *sbrk()* syscall was not implemented. This also keeps the *vip_malloc()* code as simple as possible for evaluation. I do not expect that it would have a major effect on power or performance.

vip_malloc() can be used as a drop-in replacement for *malloc()* in application code, given the ViPZonE GLIBC is available. If the target system is not running a ViPZonE-capable kernel, *vip_malloc()* defaults to calling the vanilla *malloc()*. Custom versions of *free()* and the *munmap()* syscall are not necessary to work with the variability-aware memory manager.

The Linux 3.2 *mmap()* code was used as a template for the development of *vip_mmap()*. Furthermore, the kernel includes ViPZonE helper functions that allow it 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 heavily modified physical page allocator routine (*__alloc_pages_nodemask()*). For this purpose, I reserved two unused bits in *vip_mmap()*'s *prot* field to contain the *vip_flags* passed down from the user.

Table 5.2 shows the sample set of flags supported by *vip_malloc()*, passed down to the ViPZonE back-end kernel to allocate pages from the preferred zone according to the mechanism described earlier in Sec. 5.2.2. The flags (*_VIP_TYP_READ*, *_VIP_TYP_WRITE*) tell the allocator that the expected workload is heavily read or write intensive, respectively[3]. If no flags are specified, the defaults of *_VIP_TYP_READ* and

[3]It is left to the developer to determine what constitutes read or write dominance depending on the semantics of the code. In my implementation, this did not matter, as DIMMs with high write power also had high read power, etc.

Table 5.1: ViPZonE API

Function	Parameter	Type	Description
void vip_malloc*	bytes	size_t	Request size
	vip_flags	size_t	Bitmap flag used by ViPZonE back-end page allocator
(syscall) *void* vip_mmap*	addr	void *	Address to be mapped, typically NULL (best effort)
	len	size_t	Size to be allocated, in bytes
	prot	int	Standard *mmap()* protection flags bitwise ORed with *vip_flags*
	flags	int	Standard *mmap()* flags
	fd	int	Standard *mmap()* file descriptor
	pgoff	off_t	Standard *mmap()* page offset

Table 5.2: ViPZonE supported flags

Parameter	Flag	Description
Dominant Access Type	_VIP_TYP_WRITE	The memory space will have more writes than reads
	_VIP_TYP_READ	The memory space will have more reads than writes
Relative Utilization	_VIP_LOW_UTIL	Low utilization (prefer low power space if plentiful)
	_VIP_HI_UTIL	High utilization (always prefer low power space)

_VIP_UTIL_LOW are used. I 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.

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

```
retval = vip_malloc(arraySize*elementSize,
_VIP_TYP_READ | _VIP_HI_UTIL);
```

Alternatively, the application could use the syscall directly:

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

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

As shown by this example, the necessary programming changes to exploit our variability-aware memory allocation scheme are minimal, provided the application developer has some knowledge about the application's memory access behavior.

5.3 Evaluation

In this section, I demonstrate that ViPZonE is capable of reducing total memory power consumption with negligible performance overheads, thus yielding energy savings for a given benchmark compared to vanilla Linux running on the same hardware. I start with an overview of the testbed and measurement hardware and configurations in Sec. 5.3.1. A comparison of the four combinations of memory interleaving modes is in Sec. 5.3.2 to quantify the advantages and disadvantages of disabling interleaving to allow variability-aware memory management. In Sec. 5.3.3 I compare the power, performance, and energy of ViPZonE software with respect to vanilla Linux as a baseline, using three alternate testbed configurations.

5.3.1 Testbed Setup

I constructed an x86-64 platform that would allow fine control over hardware configuration for my purposes. Table 5.3 lists the important hardware components and configuration parameters used in all subsequent evaluations. The motherboard BIOS allowed me to enable and disable channel and rank interleaving independently, as well as adjust all voltages, clock speeds, memory latencies, etc. if necessary. Memory power was measured on a per-DIMM basis using an external data acquisition (DAQ) unit, as shown by the testbed photo in Fig. 5.5. Data was streamed to a laptop for post-processing.

Table 5.3: Different ViPZonE testbed configurations, varying CPU performance on real hardware.

(a) Common testbed configuration

Parameter	**Value**	**Parameter**	**Value**
CPU	Intel Core i7-3820 (Sandy Bridge-E)	CPU supporting features	All enabled (default)
Motherboard	Gigabyte X79-UD5 (socket LGA2011)	Linux kernel version	3.2.1
BIOS version	F8	GLIBC version (baseline)	2.15
Storage	SanDisk SDSSDX120GG2 120 GB SSD (SATA 6 Gbps)	DAQ	NI USB-6215
No. DIMMs	2	DIMM power sample rate	1 kHz per DIMM
No. memory channels	2	Data logging	Second machine
DIMM capacity	2 GB each	Base core voltage	1.265 V
DDR3 data rate	1333 MT/s	DRAM voltage	1.5 V

(b) *Fast2*

Parameter	**Value**	**Parameter**	**Value**
No. enabled cores	4	TurboBoost	Enabled
Nominal core clock	3.6 GHz	HyperThreading	Enabled
		No. of PARSEC threads	8

(c) *Slow2*

Parameter	**Value**	**Parameter**	**Value**
No. enabled cores	2	TurboBoost	Disabled
Base core clock	1.8 GHz	HyperThreading	Disabled
		No. of PARSEC threads	1

Tables 5.3b and 5.3c list two different CPU and memory configurations that share the same parameters from Table 5.3a. Unless otherwise specified in the tables, all minor BIOS parameters were left at default values. In the *Fast2* configuration, two DIMMs populated the motherboard with up to 50% active total power variation (DIMMs *b1* and *c1* as depicted in Fig. 5.1 and measured in the same way as in [1]). In the *Slow2* configuration, the memory was set identically but the CPU was underclocked to 1.8 GHz, with only two cores enabled, while Intel TurboBoost and HyperThreading were disabled. None of these configurations

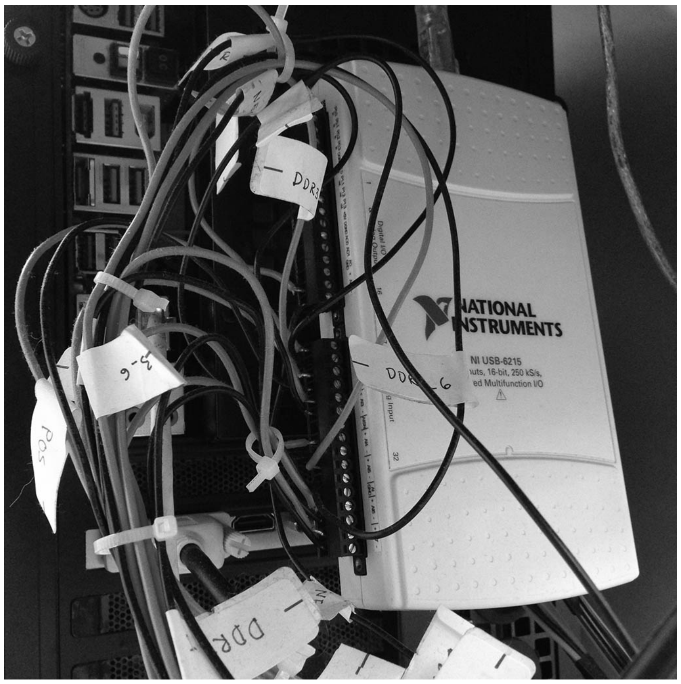


Figure 5.5: Testbed photo showing the DAQ mounted on the reverse of the chassis for DIMM power measurement.

specify anything about the channel and rank interleaving modes, which is evaluated in Sec. 5.3.2. The testbed configurations were chosen to represent two flavors of systems, those which may be CPU-bound and those which may be memory-bound in performance.

I used eight benchmarks from the PARSEC suite [37] that are representative of modern parallel computing: *blackscholes*, *bodytrack*, *canneal*, *facesim*, *fluidanimate*, *freqmine*, *raytrace*, and *swaptions*.

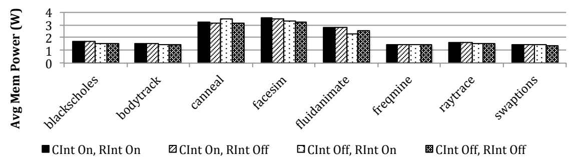
5.3.2 Interleaving Analysis

Since disabling interleaving is required for ViPZonE functionality and exploitation of memory variability in our testbed environment, I measured the average memory power, execution time, and total memory energy for different PARSEC benchmarks under both testbed configurations.

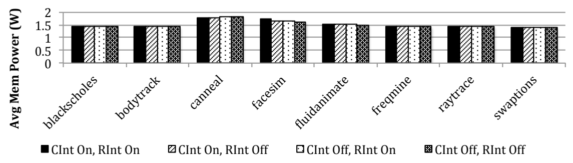
The available combinations of interleaving were:

- *Cint On, Rint On.* Channel interleaving and rank interleaving are enabled.
- *Cint On, Rint Off.* Channel interleaving is enabled, while rank interleaving is disabled.
- *Cint Off, Rint On.* Channel interleaving is disabled, while rank interleaving is enabled.
- *Cint Off, Rint Off.* Channel interleaving and rank interleaving are disabled.

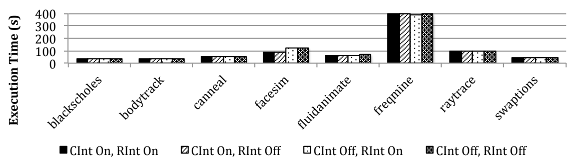
In the high-end CPU configuration, *Fast2* (results in Figs. 5.6a, 5.6c, 5.6e), the CPU was set for maximum performance, with PARSEC running with eight threads to stress the memory system. For benchmarks with high memory utilization, such as *canneal*, *facesim*, and *fluidanimate*, I found that turning off channel interleaving generally reduced power consumption (Fig. 5.6a), while total memory energy increased or decreased depending on the application (Fig. 5.6e) due to degradation in performance from lower main memory throughput (Fig. 5.6c). Rank interleaving had less impact on power or performance, which suggests that the main throughput bottleneck is the effective bus width rather than the devices. Conversely, for workloads with lower main memory utilization, there was negligible difference in power, performance, and energy. This



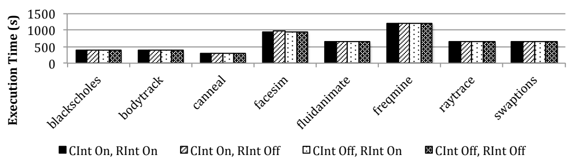
(a) *Fast2* Memory Power



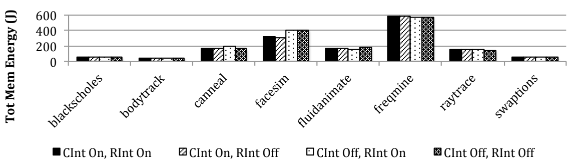
(b) *Slow2* Memory Power



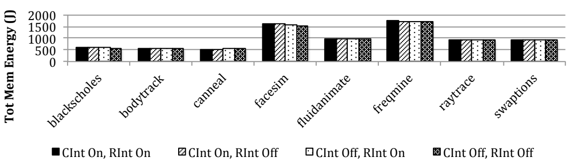
(c) *Fast2* Runtime



(d) *Slow2* Runtime



(e) *Fast2* Memory Energy



(f) *Slow2* Memory Energy

Figure 5.6: Evaluation of channel and rank interleaving for both testbed configurations on vanilla Linux.

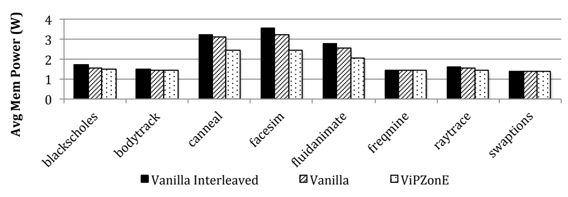
confirms the intuition that the benefits of interleaving are in the improvement of peak memory throughput, which is only a bottleneck for certain workloads where the CPU is sufficiently fast and/or application memory utilization is high.

In the slower CPU setup, *Slow2* (results in Figs. 5.6b, 5.6d, 5.6f), running only a single workload thread, interleaving generally had little effect on memory power, runtime, and memory energy, because the processor/application were unable to stress the memory system. In these cases, interleaving could be disabled with no detrimental effect to performance. Note that power savings of disabling interleaving could be higher if the memory controller used effective power management on individual idle DIMMs, as opposed to global DIMM power management. A related work on power aware page allocation [30] also required interleaving to be disabled in order to employ effective DIMM power management. As interleaving remains an issue for general DIMM-level power management schemes, further investigation into the inherent tradeoffs and potential solutions is an open research direction.

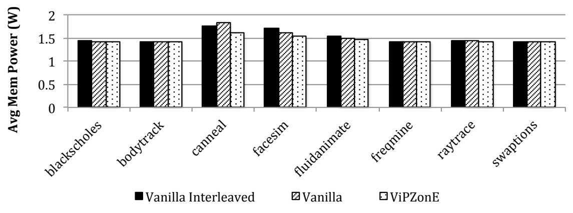
5.3.3 ViPZonE Analysis

In the evaluation of ViPZonE software, channel and rank interleaving were always disabled, as it was a prerequisite for functionality. The PARSEC benchmarks were not explicitly annotated (modified) for use with the ViPZonE front-end, although we have tested the full software stack for correct functionality. Instead, I emulated the effects of using all low-power allocation requests by using the default *Low Power First* physical page allocation policy described in Sec. 5.2.2. ViPZonE was benchmarked with two system configurations, namely *Fast2* (Fig. 5.7a, 5.7c, 5.7e) and *Slow2* (Fig. 5.7b, 5.7d, 5.7f). By using the two-DIMM configurations, I could better exploit memory power variability by including only the highest and lowest power two DIMMs from Fig. 5.1. Because we cannot harness idle power variability in this scheme, additional inactive DIMMs merely act as a “parasitic” effect on total memory power savings; with more DIMMs in the system, a greater proportion of total memory power/energy is consumed by idle DIMMs.

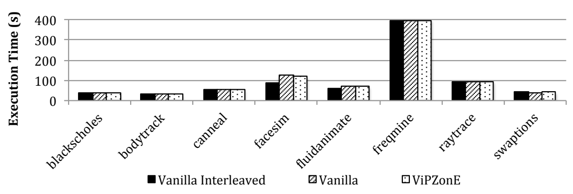
While ViPZonE does not explicitly deal with idle power management like other related works, it could be supplemented with orthogonal access coalescing methods which exploit deep low-power DRAM states. From the address mapping perspective, the nature of ViPZonE’s zone preferences already implicitly allow more DIMMs to enter low-power (or, potentially off) modes by grouping allocations to a subset of the memory.



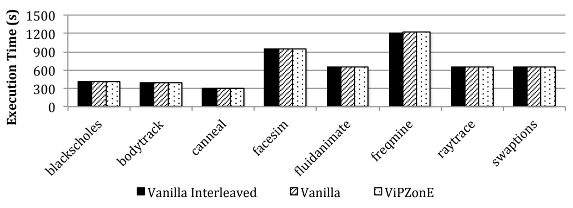
(a) *Fast2* Memory Power



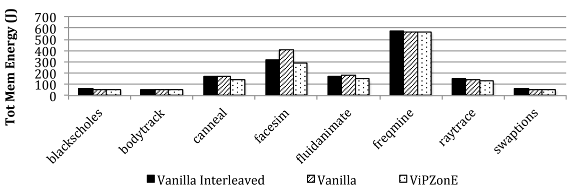
(b) *Slow2* Memory Power



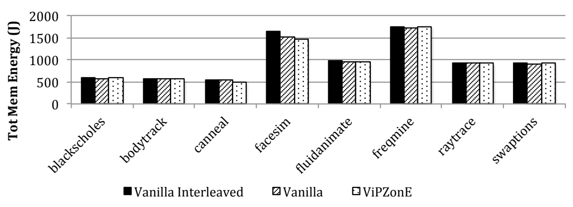
(c) *Fast2* Runtime



(d) *Slow2* Runtime



(e) *Fast2* Memory Energy



(f) *Slow2* Memory Energy

Figure 5.7: ViPZonE vs. vanilla and interleaved (Cint On, Rint On) vanilla Linux.

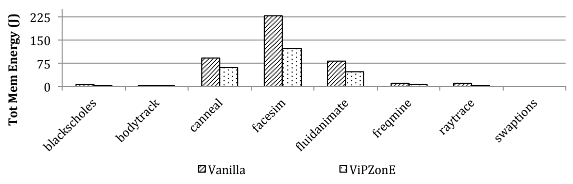
The results from the *Fast2* configuration indicate that ViPZonE can save up to 27.80% total memory energy for the *facesim* benchmark with respect to the vanilla software, also with interleaving disabled. Intuitively, the results make sense: benchmarks with higher memory utilization can better exploit active power variability between DIMMs. For this reason, lower-utilization benchmarks such as *blackscholes* gain no benefit from ViPZonE, just as they see no benefit from interleaving (refer to Sec. 5.3.2). With the slower CPU configuration, Figs. 5.7b, 5.7f indicate that there is a reduced benefit in power and energy from ViPZonE, for the same reasons. The reduced CPU speed, results in less stress being placed on the memory system, resulting in lower average utilization and a higher proportion of total memory power being from idleness.

It may initially surprise the reader to note that in some cases, vanilla interleaved roughly matches ViPZonE on the energy metric. This is due to the performance advantage of interleaving. Because I currently have no way to combine ViPZonE with interleaving (although I propose possible solutions in Chapter 6), I use vanilla without interleaving as the primary baseline for comparison with ViPZonE. In other words, the primary baseline is on equal hardware terms with ViPZonE. The direct comparison with vanilla interleaved was included for fairness, as it merits discussion on whether a variation-aware scheme should be used over a conventional interleaved memory given the current state of DRAM memory organization. Nevertheless, I believe there is significant potential for further work on power variation-aware memory management, especially if there is a solution to allow interleaving simultaneously.

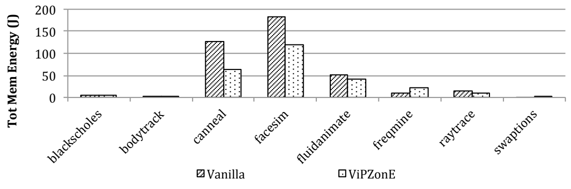
In all cases, ViPZonE running with channel and rank interleaving disabled achieved lower memory energy than the baseline vanilla software, with or without channel and rank interleaving.

5.3.4 What-If: Expected Benefits With Non-Volatile Memories Exhibiting Ultra-Low Idle Power

From the results of the ViPZonE comparison on the testbed, I speculate that the benefits of this scheme could be significantly greater with emerging non-volatile memory (NVM) technologies, such as STT-RAM, PCM, etc. I expect that there are two primary characteristics of NVMs that would make ViPZonE more beneficial: (1) extremely low idle power, thus eliminating its aforementioned parasitic effect on access power variability-aware solutions, and (2) potentially higher process variability with novel devices, leading to higher power variability that can be opportunistically exploited.



(a) *Fast2* Memory Energy, idle energy removed



(b) *Slow2* Memory Energy, idle energy removed

Figure 5.8: ViPZonE vs. Vanilla Linux, "what-if" evaluation for potential benefits with NVMs (CInt Off, Rint Off).

Thus, I present the results with the idle power component removed[4]. While this by no means an accurate representation of the realities of non-volatile memories, such as asymmetric write and read power/performance and potential architectural optimizations, this is meant to illustrate how active power variability can be better exploited without the parasitic idle power. The idle power was approximately 1.41 W for the two DIMMs used in the *Fast2* and *Slow2* configurations, specifically DIMMs *b1* and *c1* from Fig. 5.1. As can been seen in Fig. 5.8a and Fig. 5.8b, the overall memory energy benefits could increase dramatically, up to 50.69% for the *canneal* benchmark. Although these numbers do not realistically represent the results with actual NVMs, as they were derived from the DRAM modules, they present a case for variability-aware solutions for future memories with low idle power and higher power variation.

5.3.5 Summary of Results

Table 5.4 summarizes the results from the evaluation of ViPZonE, as well as the theoretical "what-if" study for potential application to NVM-based systems. I expect that with emerging NVMs, the lack of a significant idle power component will result in ViPZonE getting significant energy savings for workloads with a variety of utilizations, even as the number of modules in the system increase. Thus, using variability-aware memory allocation instead of interleaving would likely be a promising option for future systems.

Table 5.4: Summary of ViPZonE results, with respect to vanilla software with channel and rank interleaving disabled

Metric	**Value (Benchmark)**	**Metric**	**Value (Benchmark)**
Max memory power savings, *Fast2* config	25.13% (*facesim*)	Max memory power savings, *Slow2* config	11.79% (*canneal*)
Max execution time overhead, *Fast2* config	4.80% (*canneal*)	Max execution time overhead, *Slow2* config	1.16% (*canneal*)
Max memory energy savings, *Fast2* config	27.80% (*facesim*)	Max memory energy savings, *Slow2* config	10.77% (*canneal*)
Max memory energy savings, *Fast2* config estimated, "NVM" (no idle power)	46.55% (*facesim*)	Max memory energy savings, *Slow2* config estimated, "NVM" (no idle power)	50.69% (*canneal*)

[4]No performance figures are presented, as I did not actually run the system with real non-volatile memories.

Chapter 6

Conclusion and Future Work

In this work, I implemented and evaluated ViPZonE, a system-level energy-saving scheme that exploits the power variability present in a set of DDR3 DRAM memory modules. ViPZonE is implemented for Linux x86-64 systems, and includes modified physical page allocation routines within the kernel, as well as a new system call. User code can reduce system energy use by using a new variant of *malloc()*, only requiring the ViPZonE kernel and C standard library support. My experimental results, obtained using an actual hardware testbed, demonstrates up to 27.80% energy savings with no more than 4.80% performance degradation for certain PARSEC workloads compared to standard Linux software. A brief "what-if" study suggested that this approach could yield greatly improved benefits using emerging non-volatile memory technology that consume no idle power, notwithstanding potentially higher power variability compared to DRAMs. As my approach requires that no channel or rank interleaving be used, I also included a comparison of four different interleaving configurations on the testbed to evaluate the impact of interleaving on realistic workloads.

The lack of interleaving support in the current implementation of ViPZonE is its primary drawback. It is a general problem facing DIMM-level power management schemes, and I believe finding good tradeoffs remains an open research question. I do not claim that ViPZonE is the best solution for all applications and systems. Rather, it is an interesting demonstration of a novel memory management concept in a realistic setting, and motivates further research in this space.

There are several opportunities for further research with ViPZonE. First, given the ability to co-design hardware and software, it might be possible to combine the benefits of interleaving for performance while exploiting power variation for energy savings. I can imagine a few ways this could be done. One solution would use a modified memory controller that interleaves different groups of DIMMs independently. This compromise would allow for performance and potential energy savings somewhere between the current interleaving vs. ViPZonE scenario, but would still be a static design-time or boot-time decision. This could be useful in systems that already have clustered memory, such as non-uniform memory access (NUMA) architectures.

Alternatively, hypothetical systems with disparate memory device technologies side-by-side (e.g., a hybrid DRAM-PCM memory as in [59]) may discourage interleaving across device types due to different access power, latency, read/write asymmetry, and data volatility. In this case, interleaving could still be used within each cluster of homogeneous memory technology, and each such cluster could be used as a single zone for ViPZonE. The result would be ViPZonE becoming heterogeneity-aware as a generalization of variability-awareness.

A more radical idea which may allow the full benefits of interleaving alongside ViPZonE would likely require a re-design of the DIMM organization to allow individual DIMMs, where each rank is multi-ported, to occupy multiple channels. However, the major issue I forsee with this is a much higher memory cost due to the multiplied pin requirements.

Aside from enabling interleaving alongside variation-aware memory management, ViPZonE could potentially be improved on the software side. Adding compiler support could take some of the burden off the programmer while expanding the scope to include static program data. Variability-aware page migration schemes might yield further improvements in energy efficiency by augmenting our static allocation-time hints. My approach could likely be complemented by several other power-aware methods mentioned in Chapter 1.

A simulation study of ViPZonE with detailed models of non-volatile memories could give a better idea of the benefits in the future, where power and delay variation are likely to be higher and there is negligible idle power.

I believe ViPZonE makes an effective case for further research into the Underdesigned and Opportunistic computing paradigm with the goal of improving energy efficiency of systems, while lowering design cost, improving yield, and recovering lost performance due to conventional guardbanding techniques.

Acknowledgment

I would like to thank my advisor, Dr. Puneet Gupta from UCLA, for his guidance in this work and for his support of my growth as a researcher. I would also like to thank my collaborators Drs. Nikil Dutt, Alex Nicolau, and Luis Bathen from UC Irvine, for their insight, expertise, and numerous contributions to this work. Finally, I would like to thank Dr. Mani Srivastava from UCLA for his valued mentorship.

This work was supported by the NSF Variability Expedition project (www.variability.org).

Bibliography

[1] M. Gottscho, A. A. Kagalwalla, and P. Gupta, "Power Variability in Contemporary DRAMs," *IEEE Embedded Systems Letters*, vol. 4, no. 2, pp. 37–40, 2012.

[2] L. Bathen, M. Gottscho, N. Dutt, P. Gupta, and A. Nicolau, "ViPZonE: OS-Level Memory Variability-Driven Physical Address Zoning for Energy Savings," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. IEEE/ACM/IFIP, 2012, pp. 33–42.

[3] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester, "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 8–23, 2013.

[4] N. Dutt, P. Gupta, A. Nicolau, L. Bathen, and M. Gottscho, "Variability-Aware Memory Management for Nanoscale Computing," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2013.

[5] J. M. Rabaey, A. Chandrakasan, and N. Borivoje, "Designing Memory and Array Structures," in *Digital Integrated Circuits - A Design Perspective*, 2nd ed. Pearson Education, Inc., 2003, pp. 623–717.

[6] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[7] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer, "High-Performance CMOS Variability in the 65-nm Regime and Beyond," *IBM Journal of Research and Development*, vol. 50, no. 4.5, pp. 433–449, 2006.

[8] K. A. Bowman, S. G. Duvall, and J. D. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, pp. 183–190, 2002.

[9] 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. ACM, Apr. 2003, pp. 338–342.

[10] N/A, "The International Technology Roadmap for Semiconductors."

[11] K. Jeong, A. B. Kahng, and K. Samadi, "Impact of Guardband Reduction on Design Outcomes: A Quantitative Approach," *IEEE Transactions on Semiconductor Manufacturing*, vol. 22, no. 4, pp. 552–565, 2009.

[12] F. Wang, C. Nicopoulos, X. Wu, Y. Xie, and N. Vijaykrishnan, "Variation-aware task allocation and scheduling for MPSoC," in *IEEE/ACM International Conference on Computer-Aided Design, 2007. ICCAD 2007*, 2007, pp. 598–603.

[13] J. Sartori, A. Pant, R. Kumar, and P. Gupta, "Variation-aware speed binning of multi-core processors," in *2010 11th International Symposium on Quality Electronic Design (ISQED)*, 2010, pp. 307–314.

[14] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi, "Battery-Driven System Design: A New Frontier in Low Power Design," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE Computer Society, Jan. 2002, p. 261.

[15] L. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, and M. Srivastava, "Variability-Aware Duty Cycle Scheduling in Long Running Embedded Sensing Systems," in *Design, Automation, and Test in Europe (DATE).* IEEE, 2011, pp. 1–6.

[16] 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.* ACM, Jan. 2010, pp. 85–90.

[17] L. Cheng, P. Gupta, C. J. Spanos, K. Qian, and L. He, "Physically Justifiable Die-Level Modeling of Spatial Variation in View of Systematic Across Wafer Variability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 388–401, Mar. 2011.

[18] L.-T. Pang, K. Qian, C. J. Spanos, and B. Nikolic, "Measurement and Analysis of Variability in 45 nm Strained-Si CMOS Technology," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 8, pp. 2233–2243, Aug. 2009.

[19] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, and N. Borkar, "Within-Die Variation-Aware Dynamic-Voltage-Frequency Scaling Core Mapping and Thread Hopping for an 80-Core Processor," in *IEEE International Solid-State Circuits Conference (ISSCC).* IEEE, 2010, pp. 174–175.

[20] H. Hanson, K. Rajamani, J. Rubio, S. Ghiasi, and F. Rawson, "Benchmarking for Power and Performance," in *SPEC Benchmark Workshop*, 2007.

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

[22] 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. IEEE Computer Society, Apr. 2007, pp. 15–26.

[23] M. Mutyam, F. Wang, R. Krishnan, V. Narayanan, M. Kandemir, Y. Xie, and M. J. Irwin, "Process-Variation-Aware Adaptive Cache Architecture and Management," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 865–877, 2009.

[24] 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 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '09. ACM, Feb. 2009, pp. 251–260.

[25] L. A. D. Bathen and N. D. Dutt, "E-RoC: Embedded RAIDs-on-Chip for low power distributed dynamically managed reliable memories," in *Design, Automation, and Test in Europe (DATE)*, 2011, pp. 1–6.

[26] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller, "Energy Management for Commercial Servers," *IEEE Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[27] K. Rajamani, C. Lefurgy, S. Ghiasi, J. C. Rubio, H. Hanson, and T. Keller, "Power Management for Computer Systems and Datacenters," 2008.

[28] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu, "Compiler-directed array interleaving for reducing energy in multi-bank memories," in *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, 2002, pp. 288–293.

[29] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller, "Improving Energy Efficiency by Making DRAM Less Randomly Accessed," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. ACM, 2005, pp. 393–398.

[30] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 35, no. 11. ACM, Nov. 2000, pp. 105–116.

[31] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Scheduler-Based DRAM Energy Management," in *Proceedings of the 39th annual Design Automation Conference*. IEEE, 2002, pp. 697–702.

[32] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XI. ACM, Apr. 2004, pp. 177–188.

[33] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 2008, pp. 210–221.

[34] J. H. Ahn, J. Leverich, R. Schreiber, and N. Jouppi, "Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 5–8, Jan. 2008.

[35] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "VaMV: Variability-Aware Memory Virtualization," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 284–287.

[36] J. Hezavei, N. Vijaykrishnan, and M. J. Irwin, "A Comparative Study of Power Efficient SRAM Designs," in *Proceedings of the 10th Great Lakes Symposium on VLSI*. ACM, 2000, pp. 117–122.

[37] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. ACM, Apr. 2008, pp. 72–81.

[38] M. Gottscho, "ViPZonE Source Code," 2013.

[39] K. Itoh, *VLSI Memory Chip Design*, 1st ed. Springer, 2001.

[40] D. T. Wang and B. L. Jacob, "Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm," Ph.D. dissertation, University of Maryland, 2005.

[41] H. S. Stone, *High-Performance Computer Architecture*, 3rd ed. Addison-Wesley, 1993.

[42] G. J. Burnett and E. G. Coffman Jr, "A study of interleaved memory systems," in *AFIPS '70 (Spring) Proceedings of the May 5-7, 1970, spring joint computer conference*, May 1970, pp. 467–474.

[43] B. R. Rau, "Program Behavior and the Performance of Interleaved Memories," *IEEE Transactions on Computers*, vol. C-28, no. 3, pp. 191–199, 1979.

[44] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[45] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Transactions on Computers*, vol. C-20, no. 12, pp. 1566–1569, 1971.

[46] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Transactions on Computers*, vol. C-31, no. 5, pp. 363–376, 1982.

[47] G. S. Sohi, "High-bandwidth interleaved memories for vector processors-a simulation study," *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 34–44, 1993.

[48] J. Breternitz M. and J. P. Shen, "Organization of array data for concurrent memory access," in *Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, ser. MICRO 21. IEEE Computer Society Press, Apr. 1988, pp. 97–99.

[49] N/A, "The Linux Kernel 3.2," 2012.

[50] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.

[51] D. A. Patterson and J. L. Hennessy, "Large and Fast: Exploiting Memory Hierarchy," in *Computer Organization and Design - The Hardware/Software Interface*, 4th ed. Elsevier Inc., 2009, pp. 452–547.

[52] G. Duarte, "Anatomy of a Program in Memory," 2009.

[53] N/A, "GLIBC, The GNU C Library," 2012.

[54] ——, "TN-41-01: Calculating Memory System Power for DDR3," 2007.

[55] ——, "Memtest86+."

[56] ——, "34410A/11A 6-1/2 Digit Multimeter Users Guide," 2012.

[57] H.-W. Lee, K.-H. Kim, Y.-K. Choi, J.-H. Sohn, N.-K. Park, K.-W. Kim, C. Kim, Y.-J. Choi, and B.-T. Chung, "A 1.6 V 1.4 Gbp/s/pin Consumer DRAM With Self-Dynamic Voltage Scaling Technique in 44 nm CMOS Technology," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 131–140, Jan. 2012.

[58] K. L. Wang, J. G. Alzate, and P. Khalili Amiri, "Low-power non-volatile spintronic memory: STT-RAM and beyond," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, Mar. 2013.

[59] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *46th ACM/IEEE Design Automation Conference, 2009. DAC '09*, 2009, pp. 664–669.