Cross-Layer Memory Management for Managed Language Applications

Michael R. Jantz
University of Tennessee
mrjantz@utk.edu

Forrest J. Robinson
Prasad A. Kulkarni
University of Kansas
{fjrobinson,kulkarni}@ku.edu

Kshitij A. Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Abstract

Performance and energy efficiency in memory have become critically important for a wide range of computing domains. However, it is difficult to control and optimize memory power and performance because these effects depend upon activity across multiple layers of the vertical execution stack. To address this challenge, we construct a novel and collaborative framework that employs object placement, cross-layer communication, and page-level management to effectively distribute application objects in the DRAM hardware to achieve desired power/performance goals.

In this work, we describe the design and implementation of our framework, which is the first to integrate automatic object profiling and analysis at the application layer with fine-grained management of memory hardware resources in the operating system. We demonstrate the utility of our framework by employing it to more effectively control memory power consumption. We design a custom memory-intensive workload to show the potential of our approach. Next, we develop sampling and profiling-based analyses and modify the code generator in the HotSpot VM to understand object usage patterns and *automatically* determine and control the placement of hot and cold objects in a partitioned VM heap. This information is communicated to the operating system, which uses it to map the logical application pages to the appropriate DRAM ranks according to user-defined provisioning goals. We evaluate our framework and find that it achieves our test goal of significant DRAM energy savings across a variety of workloads, without any source code modifications or recompilations.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Memory Management, Optimization; D.4.2 [*Operating Systems*]: Storage Management—Allocation/deallocation strategies

General Terms Design, Experimentation, Performance

Keywords Cross-layer, Power, Efficiency, DRAM, HotSpot

1. Introduction

Performance and energy efficiency in the memory subsystem is a critical factor for a wide range of computing applications. Trends such as the increasing demand for data analytics (i.e. Big Data) and the desire to multiplex physical resources to improve efficiency have driven the adoption of memory systems with larger power, bandwidth, and capacity requirements. While the importance of effective object placement in memory to improve program performance is well established [13, 27], recent research suggests that power consumption in memory is also a dominant factor for many data-center and enterprise server systems [15, 29–31].

Not surprisingly, it is very challenging to obtain precise control over the distribution and usage of memory power or bandwidth when virtualizing system memory. These effects depend upon the assignment of virtual addresses to the application's data objects, the OS binding of virtual to physical addresses, and the mapping of physical pages to hardware DRAM devices. Furthermore, due to the use of virtual memory, most systems abstract away information during memory management that could be used to bring about a more efficient distribution of memory resources.

We contend that, to overcome these challenges, we need a cross-layer framework, where

- the **application-layer** determines data object usage patterns, assigns objects to appropriate locations/pages in the virtual address space, and conveys corresponding page-level memory usage information to the OS, and
- the **OS-layer** incorporates this guidance when deciding which physical page to use to back a particular vir-

tual page, according to user-specified power/performance goals.

Recently, OS-level system calls, such as *madvise*, *mbind* and *vadvise*, and frameworks have been developed to facilitate communication across multiple layers of the vertical execution stack during memory management [21, 26]. In particular, the framework proposed by Jantz et al. [21] allows applications to communicate to the OS how they intend to use portions of the virtual address space, and then uses this information to guide low-level memory management decisions.

However, all such frameworks require the application to manually determine the set of memory usage guidance to provide to the OS, which may be infeasible for many complex applications and workloads, Furthermore, where relevant, all memory usage guidance has to be manually inserted into the source code, and modified applications must be recompiled.

Our work aims to address the limitations of such OS-level frameworks by developing a corresponding *automated application-layer* mechanism to appropriately empower and guide the OS-layer actions without any additional program recompilations. Our work, implemented in the standard HotSpot Java Virtual Machine (JVM), divides the application's *heap* into separate regions for objects with different (expected) usage patterns. At application run-time, our custom JVM automatically partitions and allocates heap data into separate regions using an integrated object partitioning strategy. Our JVM then transfers this information to the OS, where it is used to guide physical memory management. For this work, we extend HotSpot with two new object partitioning strategies: an *offline* profiling-based approach that classifies program allocation points statically, as well as an *online* sampling-based approach that segregates individual objects at run-time. To communicate and interpret the JVM's suggestions regarding physical memory management, we adopt a recent Linux OS extension and its associated API [21].

Our design supports such usage scenarios as prioritization of memory capacity and bandwidth to improve performance, and saving power by transitioning more memory ranks into self-refresh states. It is flexible enough to adapt to different provisioning goals and shifting demands at run-time. In this paper, we describe the various components of our cross-layer framework, and then demonstrate its utility to explore configurations designed to reduce DRAM energy consumption. We construct and use a custom benchmark to conduct experiments made possible by our framework that reveal interesting aspects of the power-performance tradeoff in the memory subsystem, and the potential of our framework to achieve DRAM energy savings. We integrate our approach with two object partitioning algorithms, and evaluate their effectiveness to control the placement of program objects to reduce DRAM energy consumption with our cross-layer framework.

This work makes the following important contributions:

1. We develop, implement, and evaluate a framework to automatically partition application data objects into distinct classes based on their expected usage patterns,
2. We analyze the potential of memory power savings, and the interplay between memory power, program speed, and program bandwidth requirements by designing and employing a custom benchmark program,
3. We build profiling and sampling routines into the HotSpot JVM to automatically predict memory usage behavior, and
4. We provide detailed experimental results, including analysis of DRAM power and performance, to evaluate our cross-layer memory management framework using a standard set of Java applications.

The rest of this paper is organized as follows. We present related work and background information for the various topics relevant to this work in Sections 2 and 3 respectively. We provide an overview of our complete cross-layer memory management framework in Section 4. We describe our experimental setup in Section 5. We discuss the results of experiments to study the impact of our framework on DRAM energy consumption in Section 6. We describe our object partitioning strategies in Section 7. We present our detailed experimental results and observations in Section 8. We discuss some directions for future research in Section 9. Finally, we present our conclusions in Section 10.

2. Related Work

Several researchers have explored the effect of object layouts on program performance. Hirzel [13] conducted an extensive evaluation to demonstrate the importance of data layout for program performance. Zhang and Hirzel [41] employ layout auditing to automatically select data layouts that perform well. Huang et. al. [20] use online class analysis to produce better data layouts during garbage collection. Jula and Rauchwerger [23] propose two memory allocators that use automatically provided allocation hints to improve spatial locality. Hirzel et. al. [14] and Guyer and Mckinley [11] manipulate object layout to implement optimizations in the garbage collector. In contrast to all of these works, which only affect object layout in the virtual address space, our framework controls the physical location of objects in memory as well. Another approach, by Sudan et. al. [37], attempts to co-locate small, contiguous "chunks" of cache blocks (called micro-pages) with similar access patterns into the same row-buffer. This approach is similar to ours in that it actually re-locates data in physical memory to increase efficiency. However, it requires hardware changes (and optional changes to the OS), while our approach is entirely software-based.

Other works have explored integrating information at the application-level with the OS and hardware to aid resource management. Projects, such as Exokernel [9] and Dune [3], attempt to give applications direct access to physical resources. In contrast, our framework does not use or expose any physical structures or privileged instructions directly to applications. Banga, et. al. [2] propose a model and API that allows applications to communicate their resource requirements to the OS through the use of resource containers. Brown and Mowry [5] integrate a modified SUIF compiler, which inserts *release* and *prefetch* hints using an extra analysis pass, with a runtime layer and simple OS support to improve response time of interactive applications in the presence of memory-intensive applications.

Researchers have also explored the possibility of cooperation between the JVM and the OS to improve run-time performance. For example, cooperation between the JVM heap management and the OS virtual memory systems was proposed to reduce paging or thrashing behavior during garbage collections [12, 40]. However, our work has a different goal and approach from such previous studies.

Also related is the concept of cache coloring [24], where the OS groups pages of physical memory (as the same *color*) if they map to the same location in a physically indexed cache. Despite their similar names, coloring in our framework is different than coloring in these systems. Cache coloring aims to reduce cache conflicts by exploiting spatial or temporal locality when mapping virtual pages to physical pages of different colors, while colors in our framework primarily serve to signal usage intents from the JVM to the OS.

Prior work has also explored virtual memory techniques for energy efficiency. Lebeck et. al. [28] propose several policies for making page allocation power aware. Zhou et. al. [42] track the page miss ratio curve, i.e. page miss rate vs. memory size curve, as a performance-directed metric to determine the dynamic memory demands of applications. Petrov et. al. [35] propose virtual page tag reduction for low-power translation look-aside buffers (TLBs). Huang et. al. [19] propose the concept of power-aware virtual memory, which uses the power management features in RAMBUS memory devices to put individual modules into low power modes dynamically under software control. All of these works highlight the importance and advantages of power-awareness in the virtual memory system – and explore the potential energy savings. In contrast to our work, these systems do not employ integration between upper- and lower-level memory management, and thus, are vulnerable to learning inefficiencies as well as those resulting from the operating system and application software working at cross purposes.

3. Background

Although our cross-layer framework has the potential to benefit numerous memory architectures, technologies, and variations, we focus our current study on today's pervasive JEDEC-style DDR* memory subsystems [22]. In this section, we present an overview of how memory hardware is organized in a typical DDR* memory subsystem and discuss factors contributing to DRAM power consumption. Later, we describe how memory is represented and managed by the operating system and applications.

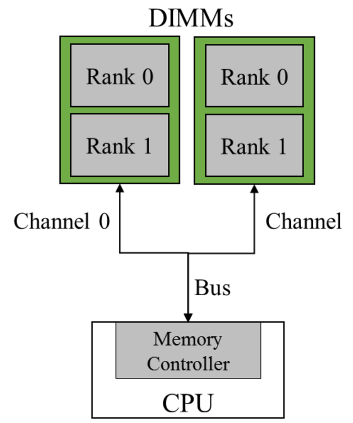


Figure 1. Organization of memory devices in a DDR* memory subsystem.

3.1 Organization of the Memory Subsystem

Figure 1 shows the organization of memory devices in a DDR*-style memory system. Each processor employs a *memory controller* that sends commands to its associated DIMMs across a *memory bus*. To enable greater parallelism, the width of the bus is split into multiple independent *channels*, which are each connected to one or more *DIMMs*. Each DIMM comprises a printed circuit board with various devices to manage frequency and control signals as well as a set of *ranks*, which constitute the actual memory storage. A rank is simply a set or block of memory that participates in a memory access. Memory on different ranks is accessed independently. A typical DIMM contains one, two, four, or eight ranks, each ranging from 2GB to 8GB in capacity. Each rank can be further subdivided into a set of *banks*, which contain independent DRAM storage arrays and share circuitry to interface with the memory bus.

3.2 DRAM Power Consumption

DRAM power consumption can be divided into two main components: *operation power*, which is the power required for active memory operations, and *background power*, which accounts for all other power in the memory system.[1] Operation power is primarily driven by the number of memory accesses to each device, while background power depends solely on the operating frequency and the current

[1] For a full accounting of the various factors that contribute to power consumption in a DDR3 memory subsystem, see [38].

power-down state of the memory devices. Modern memory devices perform aggressive power management to automatically transition from high power to low power states when either all or some portion of the memory is not active. Ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level. As with any power-management technique, turning off a portion of the device in a power-down state incurs a wakeup penalty the next time that portion of the device is accessed. Current DDR3 technology supports multiple power-down states, each of which poses a different tradeoff between power savings and wakeup latency. Table 3 in [6] provides a summary of power requirements and wakeup latencies for the various power-down states for a typical DDR3 device. Thus, controlling memory power consumption requires understanding how memory accesses are distributed across memory devices. Configurations tuned to maximize performance attempt to distribute accesses evenly across the ranks and channels to improve bandwidth and reduce wakeup latencies, while low-power configurations attempt to minimize the number of active memory devices.

3.3 Operating System View of Physical Memory

During system initialization, the BIOS reads the memory hardware configuration and converts it into physical address ranges provided to the operating system. Many vendors provide options to control (prior to boot) how physical addresses are distributed among the underlying memory hardware units. For example, options to enable channel and rank interleaving ensure that consecutive physical addresses are distributed across the system's memory devices. At boot time, the operating system reads the physical address ranges from the BIOS and creates data structures to represent and manage physical memory. *Nodes* correspond to the physical NUMA nodes in the hardware. Each node is divided into a number of blocks called *zones*, which represent distinct physical address ranges. Next, the operating system creates physical page frames (or simply, *pages*) from the address ranges characterized by each zone. Page size varies depending on the system architecture and configuration. On the (x86-based) platform we use for this work, each page addresses 4KB of memory. The kernel's physical memory management (allocation and recycling) operates on these pages, which are stored and kept track of on various lists in each zone. For example, a set of lists of pages in each zone called the *free lists* describes all of the physical memory available for allocation.[2] Most current operating systems do not keep track of how physical pages map to the underlying memory topology (i.e. the actual layout of DRAM ranks and channels in hardware) during memory management.

[2] For a more detailed description of memory management in the Linux kernel we use as base for this work, see Chapter 3 in [33].

3.4 Automatic Heap Management in Managed Language Runtimes

The operating system provides each process with its own virtual address space for allocating and using memory at the application level. Native applications use system calls, such as *brk* and *mmap*, to request and manage virtual memory resources. Programs written in managed languages, such as Java, do not interact directly with the operating system, but rather, execute inside a process virtual machine (VM) (also called runtime system). On initialization, the VM allocates a large virtual address space for use as the application's heap. When the application allocates an object (e.g., using Java's *new* instruction), the VM updates a pointer in the application's heap area to reserve space for the new object. Periodically, the heap area will fill with objects created by the application, triggering a garbage collection (GC). During GC, the VM frees up space associated with the dead (unreachable) objects, and possibly shrinks or expands the virtual memory space depending on the current need. In this way, the VM automatically manages several aspects of the application's memory resources, including the size of the application's heap and where program objects are located in the virtual memory space.

4. Cross-Layer Memory Management

Controlling the placement of program objects/data in memory to achieve power or performance efficiency requires collaboration between multiple layers of the vertical execution stack, including the application, operating system, and DRAM hardware. For this work, we modify the HotSpot JVM to divide its application heap into separate regions for objects with different expected usage patterns. We integrate our modified JVM with the OS-based framework proposed by Jantz et al. [21], which allows application-level software to communicate usage intents and provisioning goals to the OS to guide management of memory hardware resources. In this section, we briefly describe our adopted OS framework before explaining our VM-based approach.

4.1 Guiding Memory Management with Colors

Our adopted OS framework provides two major components for this work:

1. An application programming interface (API) for communicating to the OS information about how applications intend to use memory resources (usage patterns), and
2. An operating system with the ability to keep track of which memory hardware units (DIMMs, ranks) host which physical pages, and to use this detail in tailoring memory allocation to usage patterns.

To support the first component, the framework implements a *memory coloring* interface to facilitate communication of memory usage guidance from applications to the operating system. In this approach, a color is an abstraction that enables applications to indicate to the operating system that some common behavior or intention spans a set of virtual pages. Applications associate *attributes* (or combinations of attributes) with each color, to provide information to the operating system about how to manage the colored range. More generally, an application can use colors to divide its virtual pages into groups. Each color can be used to convey one or more characteristics that the pages with that color share. Contiguous virtual pages may or may not have the same color. In this way an application provides a usage map to the OS, and the OS consults this usage map in selecting an appropriate physical memory scheduling strategy for those virtual pages. Virtual pages that have not been colored, and therefore do not provide any associated guidance, are managed using a default strategy.

The memory coloring scheme is implemented using a recent Linux kernel (v. 2.6.32-431.el6). The custom kernel provides a system call API that enables applications to apply colors to virtual ranges and bind colors to attributes that describe memory usage intents. Using this API, applications can apply and change colors and attributes at any time, and the OS can efficiently interpret this information when performing memory allocation, recycling, or page migration decisions.

To provide the second major component, the OS framework organizes and facilitates memory management over power-manageable domains that are closely related to the underlying hardware. These management units are called “trays”. A *tray* is a software structure which contains sets of pages that reside on the same power-manageable memory unit (i.e. DRAM rank). Each physical memory zone in the adopted Linux kernel contains a set of trays, and the lists of pages on each zone, which are used to manage physical memory, are replaced with corresponding lists in each tray. Figure 2 (taken from [21]) shows how the adopted kernel organizes its representation of physical memory with trays in relation to the actual memory hardware. The kernel’s page management routines, which operate on lists of pages at the zone level are modified to operate over the same lists, but at a subsidiary level of trays. That is, zones are subdivided into trays, and page allocation, scanning, recycling are all performed at the tray level.

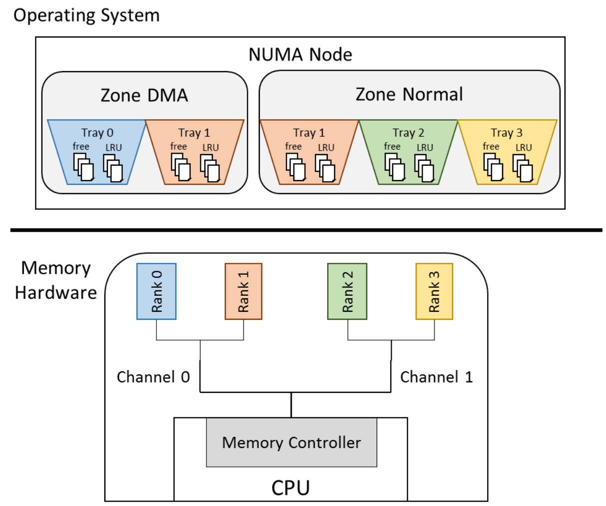


Figure 2. Organization of tray structures in relation to memory hardware in our adopted Linux kernel [21].

4.2 Organizing Memory Objects in HotSpot

Incorporating application guidance during memory management can enable the OS to achieve more efficient memory bandwidth, power, and/or capacity distributions. While the existing OS framework described in the previous subsection facilitates communication between the applications and operating system, it:

1. requires developers to manually determine the set of colors or coloring hints that will be most useful, which may be tedious or difficult for many applications, and
2. requires all colors and coloring hints to be manually inserted into source code, and all colored applications must be recompiled.

This work complements the OS framework by integrating application guidance with a custom Java virtual machine that *automatically* partitions program objects into separately colored regions. Our custom JVM divides the application’s heap into multiple distinct regions, each of which is colored to indicate how the application intends to use objects within the region. For example, in our current configuration, we divide the application’s heap into two regions: one for objects that are accessed relatively frequently (i.e. hot objects), and another for objects that are relatively cold. Colors are applied to each space during initialization and whenever region boundaries change due to heap resizing. As the application runs, the OS interprets colors supplied by the VM and attempts to select physical memory scheduling strategies tailored to the expected usage behavior of each region. The VM is free to allocate and move objects among the colored spaces, but it makes every effort possible to ensure that the objects reside in the appropriate space. Depending on the chosen coloring, this might require additional profiling and/or analysis of object usage patterns or program instructions in order to be effective. Fortunately, the VM provides a convenient environment for monitoring application behavior. Several VM tasks (such as selective compilation, code optimization, and heap resizing) already use software counters and other feedback to help guide optimization decisions [1].

We employ Oracle’s HotSpot JVM to implement our framework. Details about our specific HotSpot version and how it is configured are provided in Section 5.3. The default garbage collector in HotSpot is *generational*, meaning that the GC divides the heap into different areas according

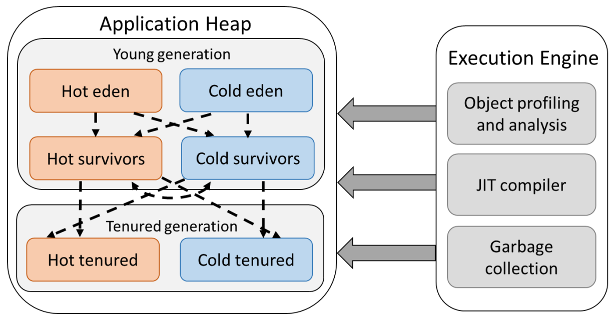


Figure 3. Colored spaces in our JVM framework. Dotted lines indicate possible paths for objects to move from one colored space to another.

to the age of the data. Newly created objects are placed in an area called the *eden space*, which is part of the younger generation. The survivor space holds objects that have survived at least one young generation collection, while objects that have survived a number of young generation GC's are eventually promoted, or tenured, to the older generation.

Figure 3 shows how our custom JVM framework organizes the application heap into separate spaces for hot and cold objects. Each space within each generation is divided into two equal-sized regions: one colored orange for hot (frequently accessed) objects and the other colored blue for cold objects. As the application executes, the VM attempts to allocate and move program objects into the appropriate colored space. For this work, we implement two distinct strategies for ensuring objects reside in the correct space: 1) an *offline* strategy (described in Section 7.1) that uses profiling information from a previous run to guide object coloring decisions in later runs, and 2) an *online* strategy (described in Section 7.2) that collects samples of program activity to predict object usage in the same run. The VM consults the available usage information at each object allocation point to determine which color to assign the new object. Periodically, perhaps at every garbage collection cycle, the VM may also re-assign colors for live objects depending on the predicted access patterns and / or program phase behavior.

5. Experimental Framework

In this section, we describe our experimental platform as well as the tools and methodology we use to conduct our experiments, including how we measure power and performance.

5.1 Platform

All of the experiments in this paper were run on a single socket HP ProLiant DL380p Gen8 server machine with an Intel E5-2620 v2 (Ivy Bridge) processor. This machine has 6 2.1GHz cores with hyperthreading enabled (for a total of 12 hardware threads) and 4 8GB DIMMs of HP DDR3 SDRAM with a maximum clock speed of 1600MHz (product #: 713979-B21). Each DIMM is connected to its own channel and is comprised of two 4GB ranks. We install 64-bit CentOS 6.5 and select a recent Linux kernel (version 2.6.32-431.el6) as our default operating system. In order to evaluate our approach against a realistic low-power baseline, we configure the system's BIOS to use the "Minimum Power Usage" HP Power Profile [16]. This configuration selects a number of options typical in a low-power machine, including reduced operating frequencies for compute devices, and no interleaving of physical memory addresses.

5.2 Power and Bandwidth Measurement

We adopt the analytical model described by David et. al. [6] to estimate memory power consumption. This model is based on *background power*, or power consumed regardless of the operational activity on the device and determined only by the background power state, and *operation power*, which depends on the commands executed by the device. It uses previously collected power measurements as well as the observed read/write bandwidth and the length of time spent in each background power state to estimate power consumption for each DIMM. For this work, we configure our system's memory devices to operate at a maximum speed of 1333MHz and use the model's corresponding power measurements listed in Tables 2 and 3 in [6]. To collect the other parameters required by the DRAM energy model as well as to estimate CPU energy consumption, we employ Intel's Performance Counter Monitor (PCM) tool [17]. PCM is a sampling-based tool that reads various activity counters (registers) to provide details about how internal resources are used by Intel devices. At every sample, PCM reports a variety of counter-based statistics, including: the number of cycles each DIMM spent in each background power state, the average read/write bandwidth on each memory channel (in MB/s), and an estimate (in Joules) of energy consumed by the CPU (cores and caches).[3] We run the PCM tool in the background during each experiment to collect samples every two seconds. We then aggregate the collected data in post-processing to compute CPU/memory energy and bandwidth estimates for each run.

5.3 HotSpot Java Virtual Machine

Oracle's HotSpot Java Virtual Machine (build 1.6.0_32) [34] provides the base for our JVM implementation. For this work, we used the default HotSpot configuration for server-class applications [18]. This configuration uses the "parallel scavenge" garbage collector (PS-GC), which is a "stop-the-world" collector (i.e. application threads do not run during GC) that is capable of spawning multiple concurrent scavenging threads during GC. For all of our experiments, we configure HotSpot to use a static heap size to avoid re-computing colored space boundaries during heap resizing.

[3] PCM employs the RAPL counter interface to estimate CPU energy consumption [7]. The RAPL interface also supports DRAM energy estimates, which we would have used for this work, but this feature was disabled on our experimental platform by the system vendor (HP).

6. Impact of Cross-Layer Framework on Program Performance and DRAM Energy Consumption

Current systems employ a hardware-based approach to manage memory power. During periods of low activity, the memory controller transitions individual ranks into low power states (such as "self refresh"). To amplify the effectiveness of this technique, it is desirable that pages that are very lightly accessed are not located on the same memory ranks as pages that are accessed frequently. In this section, we employ our framework to control the placement of hot and cold objects across memory ranks and demonstrate the effect of different layouts on program performance and DRAM energy consumption.

6.1 The MemBench Benchmark

In order to clearly isolate and measure the effect of controlling object placement, we require a simple, memory-intensive workload with well-defined, and easy to distinguish, sets of hot and cold objects. Hence, we construct a custom benchmark (called MemBench) that differentiates hot and cold objects *statically* by declaring objects with distinctive program types in its source code. The MemBench benchmark creates two main types of objects: *HotObject* and *ColdObject*. Each object contains an array of integers (comprising about 1MB of space per object) which represent the objects' memory resources. On initialization, MemBench allocates a large number of such objects and stores them in a single, large, in-memory array. A certain portion of the objects (about 15% for these experiments) are allocated as the HotObject type, while the remaining objects are allocated as the ColdObject type. The order of object allocations is randomized so that HotObjects and ColdObjects are stored intermittently throughout the object array. For each of our experiments, we configure MemBench to allocate 27GB of program objects (or approximately 90% of the free memory on our server node).

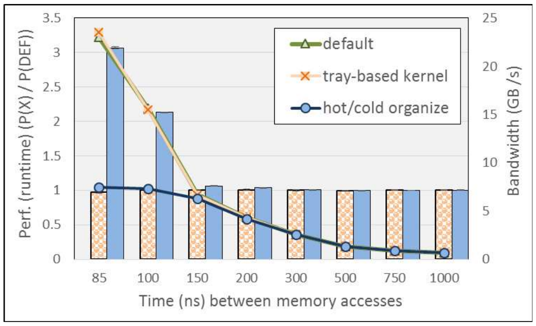
After MemBench is done allocating the hot and cold objects, the object array is divided into 12 roughly equal parts (one for each hardware thread on our server node) that are each passed to their own software thread. The threads (simultaneously) iterate over their own portion of the object array for some specified number of iterations, passing over the cold objects, and only accessing memory associated with the hot objects. When a hot object is reached, the thread performs a linear scan over the object's associated integer array, storing a new value into each cell as it walks the array. The benchmark completes when all threads have finished iterating over their portion of the object array. For these experiments, we select a large enough number of iterations so that each scanning thread runs for at least four minutes. Additionally, in order to allow experiments with different bandwidth requirements, we provide an optional "delay" parameter for MemBench that can be configured to slow down the rate at which memory accesses occur. The delay is implemented by inserting additional computation between stores to the integer array associated with each hot object. Specifically, the delay mechanism iteratively computes a Fibonacci number between integer stores. Thus, computing a larger Fibonacci number imposes a longer delay between memory accesses, which reduces the average bandwidth requirements for the workload. For our experiments, we use the system timer (i.e. *System.nanoTime()*) to measure the delay with varying Fibonacci numbers, and select numbers that induce delays at regular intervals. Note that, even if we do not insert additional computation between memory accesses, there is still a short delay (85ns) between accesses according to *System.nanoTime()*.

6.2 Experimental Evaluation

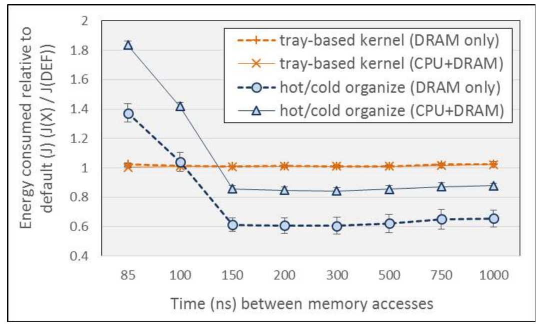
We constructed three configurations to evaluate our cross-layer framework with the MemBench benchmark. The *default* configuration employs the unmodified Linux kernel with the default HotSpot JVM. The *tray-based kernel* configuration runs MemBench with the unmodified HotSpot JVM on the custom kernel framework described in Section 4.1. Since the JVM in the tray-based kernel configuration does not provide any memory usage guidance, the kernel applies its default strategy of selecting pages from trays in a round-robin fashion. This strategy effectively interleaves contiguous virtual pages across memory devices. Our final configuration, called *hot/cold organize*, uses the custom kernel with our custom JVM that divides the application's heap into two separate colored spaces: one space colored orange for hot objects and the other colored blue for cold objects. Using the memory coloring API, we configure our framework so that, whenever possible, demands from different colored spaces are filled with physical pages from trays corresponding to different DIMMs. We also modify HotSpot's object allocator to recognize objects of the ColdObject type and assign them to the space colored blue. Alternatively, HotObjects, as well as any other objects created by the MemBench workload, are allocated to the orange space. Thus, the effect of the hot/cold organize configuration is to ensure that, of the four 8GB DIMMs, three of the DIMMs are occupied with pages that back ColdObjects, while the other DIMM is used for the rest of the MemBench memory, including the HotObjects.

We ran the MemBench benchmark with each of our three configurations and with different delay parameters to vary the bandwidth requirements for the workload. For these experiments, we intend to isolate the power and performance effects of object placement. Therefore, the following measurements are collected only during the portion of the run when threads are scanning and accessing the object array, that is, after all allocation has completed. Our results report the average over five runs for each configuration–delay pair.

The bars in Figure 4(a) (plotted on the left y-axis) compare the performance (execution time) of the tray-based ker-



(a)



(b)

Figure 4. MemBench evaluation. (a) plots performance relative to the default configuration as bars on the left y-axis and average bandwidth as lines on the right y-axis. (b) shows energy consumed relative to default.

nel and hot/cold organize configurations to the default performance with the same delay. To estimate the degree of variability in our performance and energy results, we compute 95% confidence intervals for the difference between the means [10], and plot these intervals as error bars in Figures 4(a) and 4(b). (The error bars in Figure 4(a) are difficult to see because there is little performance variability for these experiments). Thus, we can see that the tray-based kernel configuration performs about the same as the default configuration regardless of delay factor. The hot/cold organize configuration exhibits relatively poor performance when there is little delay between memory accesses, but performs more like the default configuration as the delay is increased. Interestingly, we find this observed performance difference is directly related to bandwidth on the memory controller. The lines in Figure 4(a) show the average read/write bandwidth (in GB/s) for MemBench with each configuration at varying delays. Notice that the default and tray-based kernel configurations actually produce much higher memory bandwidth than hot/cold organize when the delay factor is low. With very little delay between memory accesses, MemBench requires very high memory bandwidth to achieve good performance. The default and tray-based kernel configurations both distribute hot objects across the system's memory devices, allowing the system to exploit rank and channel parallelism to achieve higher bandwidths. Alternatively, the hot/cold organize configuration co-locates all the hot objects onto a single DIMM that is connected to a single channel. Consequently, this configuration cannot attain the bandwidth required to achieve good performance when the delay is low. Increasing the delay between memory accesses reduces the bandwidth requirements for the workload, and thus, enables hot/cold organize to achieve performance similar to the other configurations.

While co-locating the hot objects onto a single DIMM restricts bandwidth, this organization consumes much less power, on average, than distributing accesses across all the memory devices. We find that this power differential persists even when the bandwidth requirements are reduced and the performance of the different configurations is similar. Thus, there is a significant potential for DRAM *energy* savings with the hot/cold organize configuration. Figure 4(b) shows the energy consumed with the tray-based kernel and hot/cold organize configurations compared to the default configuration with different delay factors. We plot two lines for each configuration: one that compares only DRAM energy consumption, and another to show the relative energy consumption of the CPU and DRAM together. Observe that, in experiments with low delay, the hot-cold organize configuration actually requires more DRAM energy than the default configuration due to its relatively poor performance. However, our prototype framework enables significant energy savings for workloads that do not require as much memory bandwidth. At a maximum, the hot/cold organize configuration achieves 39% DRAM energy savings (with delay=300).

As expected, the CPU energy consumed during these experiments is closely related to the performance of the workload. When the delay between memory accesses is low, the hot/cold organize configuration consumes more CPU energy than the default configuration (about 2.8x in the worst case, when delay=85). Similarly, when the delay is increased past a certain point, the CPU consumes about the same amount of energy regardless of the object layout configuration. As a result, the hot/cold organize configuration significantly reduces the node's *combined* (CPU+DRAM) energy when the delay between memory accesses is greater than 150ns. In the best case (delay=300), hot/cold organize reduces the (CPU+DRAM) energy consumption by 15.8% compared to the default configuration.

7. Object Partitioning Strategies

The experiments in the previous section demonstrate that there is significant potential to reduce DRAM energy consumption using our JVM framework. However, these experiments use a custom benchmark that was written to explicitly assign different program types to distinguish objects that are accessed frequently from objects that are relatively cold. In order to maximize the effectiveness of our framework, we desire an approach that is able to automatically, efficiently, and accurately distinguish and partition hot and cold objects for arbitrary applications.

Such characterization of dynamic object usage patterns can be achieved by observing earlier program behavior in the same run (online) or by incorporating profiles collected during a separate run (offline). Online partitioning strategies are attractive because they can automatically adapt to different program inputs and changes in program behavior during execution. However, profiling memory references at run-time to gather information about each object's usage patterns entails significant overheads. On the other hand, offline partitioning schemes can collect and analyze very detailed profiles of application behavior and have very little run-time overhead. Of course, such techniques depend upon the representativeness of the profile run and can only provide static guidance.

Considering these tradeoffs, we construct two distinct object partitioning strategies to integrate with our framework: 1) an offline approach that uses information collected during a separate profile run to characterize program allocation points as either hot or cold, and 2) an online approach that monitors the run-time call stack to determine methods and classes that are likely to correspond to hot objects. In this section, we describe our offline- and online-based partitioning schemes.

7.1 Profiling for Hot and Cold Allocation Points

Our offline partitioning strategy profiles memory usage activity related to *program allocation points*. For this approach, we instrument the HotSpot bytecode interpreter to construct a hash table relating objects and their usage activity to their allocation points in the interpreted code. The hash table is indexed by program allocation points (identified by method and bytecode index) and values are a list of records describing the size and access count of each object created at that point. During the run, whenever the VM interprets a `_new` instruction (or one of its variants), it creates a record in the global hash table to indicate that a new object was created at that point. Then, whenever an object is accessed using one of the object reference bytecodes (`getfield`, `putfield`, etc.), the VM increments its access count.

After collecting the application's memory profile, the next step is to assign colors to the allocation points according to the memory usage activity of the objects created at each point. For this analysis, we are interested in maximizing the size of the cold space because a larger space of cold objects ensures that a larger portion of DRAM devices will be able to reside in a low power state. We can express this problem as an instance of the classical 0/1 knapsack optimization problem. In this formulation, each allocation point is an item, which has a value equal to the total size of the objects created at that point, and a weight equal to the sum of the access counts for those objects. We assume some maximum number of object accesses is the capacity of the knapsack. The optimization problem then is to select items (allocation points) such that the combined value (object size) is maximized without the combined weight (access counts) exceeding the capacity of the knapsack. Although the knapsack problem is NP-complete, there exist well-known polynomial-time algorithms that can approximate the solution to any specified degree. We employ a popular dynamic programming algorithm to find a partitioning that (nearly) maximizes the size of the cold space [32].

To compare colorings across different workloads, we select knapsack capacities as a percentage of the total number of accesses in the profile run. For example, with a knapsack capacity of 5%, our technique selects allocation points such that the objects created at these points together account for no more than 5% of the total number of object accesses in the profile run. For the chosen knapsack capacity, we compute a partitioning of the program's allocation points using the approach described above and store the result to a file on disk. Then, at the start of any guided/measured program run, the VM loads the partitioning information into memory. Specifically, the hot/cold classification of each allocation point is stored in a list on the point's associated method object. Since each node in this list requires only a few bytes of space (16 in our current implementation), and since each method has only a few allocation points (Table 1 in Section 8 lists the number of static allocation points per method in each benchmark), the space overhead required for this approach is small and could be further optimized by recording information for only the hot or cold allocation points. Whenever the application allocates an object, the VM checks the associated method's list to determine whether the allocation point has a known color. If so, the new object is assigned to the corresponding colored space. Otherwise, the object is simply assigned to some default colored space. In our experiments, we opt to assign objects created at unknown allocation points to the orange, or hot, space. Thus, if an unknown object is actually hot, it will not thwart our power-saving strategy of keeping hot objects off the cold DIMMs.

7.2 Sampling Application Threads' Call Stacks for Hot Methods and Classes

Our online approach is inspired by a technique proposed by Huang et. al. [20]. This strategy combines simple static analysis with light-weight sampling to automatically predict methods and classes associated with hot program objects. To implement this approach, the HotSpot compiler records the list of classes with instances that might be accessed by the compiled method. Specifically, any class whose instance would be the source/target of a load/store instruction in the compiled method is added to a list associated with the compiled method, which we call the ClassAccessList. At the same time, a separate thread, invoked by a timer-driven signal, periodically examines the call stacks associated with each application thread. For each method at the top of a call stack, a counter associated with the method is set to indicate

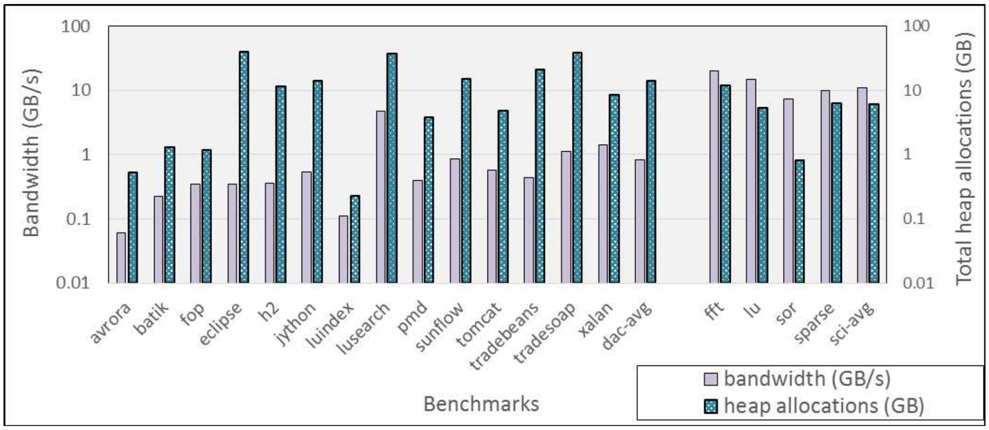


Figure 5. Bandwidth (GB/s) and total heap allocations (GB) for each benchmark with the default configuration.

Table 1. Benchmark descriptions (AP=distinct static allocation points, AP/M=static allocation points per method).

Benchmark	Description	APs	AP/M
DaCapo			
avrora	program simulator for AVR microcontroller	2402	2.54
batik	produces SVG images from batik unit tests	5132	2.69
fop	generates a PDF file from XSL-FO file	9061	5.96
eclipse	performance tests for Eclipse IDE	9171	2.32
h2	transactions on an in-memory database	2132	2.21
jython	interprets the pybench Python benchmark	10296	4.86
luindex	uses lucene to index a set of documents	1826	2.27
lusearch	uses lucene to search for keywords	1565	2.29
pmd	analyzes set of Java classes for problems	2385	2.13
sunflow	renders images using ray tracing	2717	2.51
tomcat	runs queries against a Tomcat server	7237	2.54
tradebeans	daytrader benchmark via Java Beans	14215	2.30
tradesoap	daytrader benchmark via SOAP	14339	2.28
xalan	transforms XML documents into HTML	2516	2.56
SciMark			
fft	Fast Fourier Transform	5579	3.13
lu	LU matrix factorization	5582	3.13
sor	Jacobi Successive Over-relaxation	5569	3.13
sparse	sparse matrix multiply	5571	3.13

that the method is hot. Additionally, counters associated with all of the classes on the method's ClassAccessList are set to indicate that instances of this class are likely to be hot. In order to "cool down" methods and classes that are no longer hot, we also periodically decrement the counters associated with the hot methods and classes. In this way, methods and classes that are not found on any thread's call stack for some number of sampling intervals will eventually become cold.

Whenever a new object is allocated, the allocator examines the counters associated with the method performing the allocation and the class of the object that is to be allocated. If either counter indicates that the new object is likely to be hot, we allocate the object to the hot space. Otherwise, the object is allocated to the cold space. It is important to note that, in contrast to the offline approach, some objects might change color as the application executes. At each garbage collection cycle, the surviving objects' colors are reassigned according to the hotness/coldness of their classes.

8. Evaluation

8.1 Benchmarks and Experimental Configuration

We employ the entire suite of DaCapo benchmarks [4] as well as four of the SciMark benchmarks from SPECjvm2008 [36] to evaluate our cross-layer framework. Table 1 shows the benchmarks we use to evaluate our framework along with a short description, the number of allocation points reached during execution, and the average number of allocation points for each method that creates at least one object. Some of our experiments use information collected during a profile run with a small input size to guide coloring decisions in a run with a default or large input size. Since most of the workloads in SPECjvm2008 only include one input size, we omit these from our study. All of our experiments measure *steady-state* performance and energy consumption. For each run of a DaCapo benchmark, we iterate the workload a total of seven times. We use two initial iterations to put the benchmark into steady-state, and record the median execution time of the final five iterations as the benchmark's performance. For the SciMark benchmarks, the harness continuously iterates the workload for each run, starting a new operation as soon as a previous operation completes. Each run includes a warmup period of one minute and an iteration period of at least four minutes. The score for each SciMark run is the average time it takes to complete one operation during the iteration period. For all of our performance and energy measurements, we report the average result of five runs as the final score for the benchmark.

The experimental configurations we evaluate in this section use a custom Linux kernel as well as a custom version of HotSpot. Since we wish to evaluate the performance and energy impact of only our JVM framework, we first conducted experiments to determine the effect, if any, the custom kernel has on performance and energy consumption. For these experiments, we run all of our selected benchmarks using the same unmodified HotSpot JVM with both the unmodified Linux kernel and the custom Linux kernel with tray-based

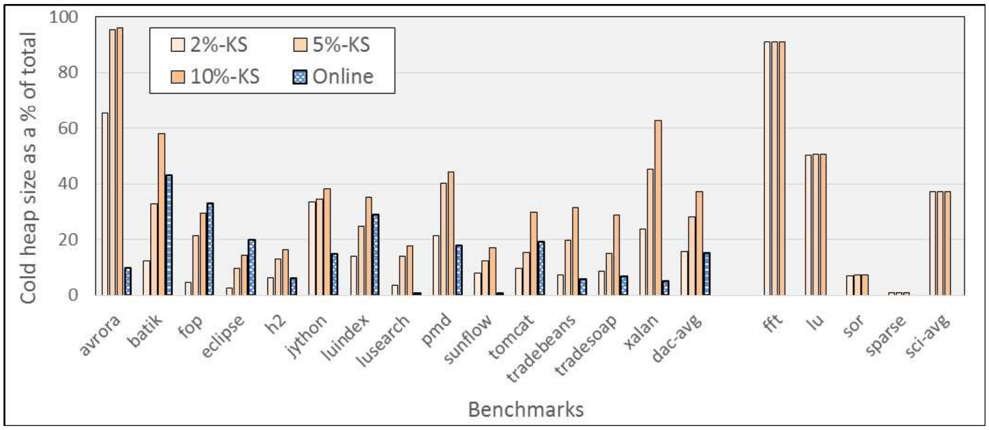


Figure 6. Cold heap size as a percentage of total size with different object partitioning schemes.

allocation. For all but one of the benchmarks, we found there is no significant performance difference between the two kernel versions. The *fft* benchmark from SciMark runs 8% faster on the custom kernel, which we suspect is because the custom kernel's page allocation strategy results in a more efficient distribution of the application's data objects. The custom kernel consumes, on average, about 2% more DRAM energy than the default Linux kernel, but 14 out of 18 benchmarks show no significant difference in DRAM energy consumption. For all of the remaining experiments in this section, we use the default (unmodified) Linux kernel with the default version of HotSpot as the baseline configuration.

Figure 5 shows the bandwidth (in GB/s, plotted on the left y-axis) and the aggregate size of allocations to the heap (in GB, plotted on the right y-axis) for each of our selected benchmarks with the default configuration. Thus, these workloads display a wide range of bandwidth and capacity requirements. On average, the DaCapo benchmarks require much less bandwidth (less than 1 GB/s) than the SciMark benchmarks (more than 11 GB/s), but often allocate more application data on the heap.

8.2 Experimental Evaluation

We now present a series of experiments to evaluate the performance and effectiveness of our JVM framework with the different object coloring strategies described in Section 7.

To evaluate our framework with the offline partitioning strategy, we collect a profile of each benchmark's memory activity using its small input size. Next, we partition the benchmark's profiled allocation points into different colored sets as described in Section 7.1 using knapsack capacities of 2%, 5%, and 10%. We then employ the allocation point partitionings to guide object coloring in a set of experiments with the larger benchmark inputs.

We also perform a set of experiments to evaluate our framework with the online partitioning strategy.[4] For these experiments, we configure our framework to sample the application call stacks every 50ms. In preliminary experiments, we found that shorter sampling periods significantly degrade the performance of our benchmarks, while using longer sampling periods did not affect performance. We also configure the "cool down" mechanism to remove methods or classes from the hot set within 500ms to 1000ms of the last time sample that they were identified as hot.

8.2.1 Cold Heap Size

Figure 6 presents the aggregated size of objects assigned to the cold space as a percentage of the total size with each of the object partitioning strategies. For example, if the profile run for the *pmd* benchmark is accurate, the 2% knapsack partitioning scheme finds a set of allocation points that account for about 22% of total heap size, but only about 2% of object accesses. We can make a few observations from these results:

1. With the DaCapo benchmarks, increasing the knapsack capacity increases the cold space size for the offline partitioning strategy as expected.
2. However, the knapsack capacity has very little effect on the SciMark workloads. This is because the vast majority of heap space is allocated at only a few allocation sites in these benchmarks.
3. On average, the online profiling scheme assigns about as many objects to the cold space as the 2% knapsack partitioning. However, for several individual benchmarks (e.g. *avrora*, *batik*, and *eclipse*) the proportion of cold space objects for the offline and online approaches is different.

[4] Our current implementation of the online partitioning strategy requires that all the VM threads must be brought to a safepoint before the sampling thread is allowed to examine the call stacks. Unfortunately, this restriction severely impacts the operation of the SciMark workloads, often causing them to hang or crash. We are currently working to implement a sampling strategy that does not require other threads to reach a safepoint before each sample. For this work, however, we evaluate our online partitioning scheme using only the benchmarks from the DaCapo benchmark suite.

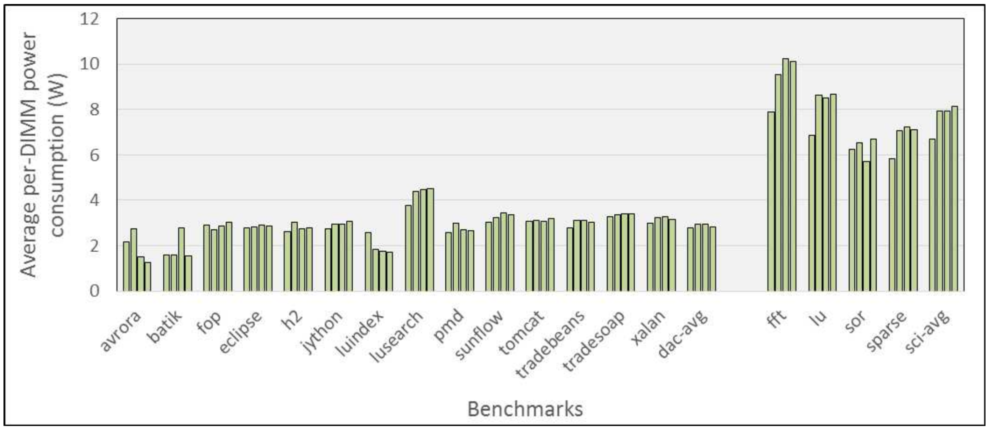


Figure 7. Average power consumed (in W) for each DIMM with the default configuration.

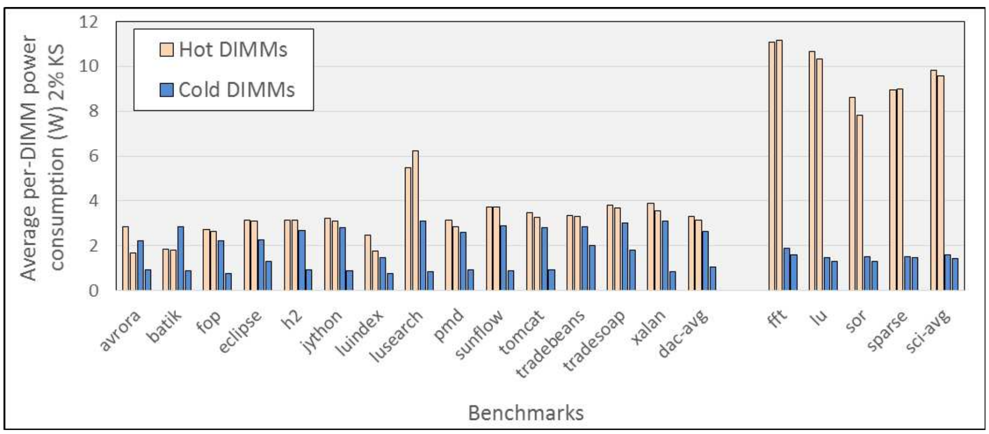


Figure 8. Average power consumed (in W) for each DIMM with 2% knapsack coloring.

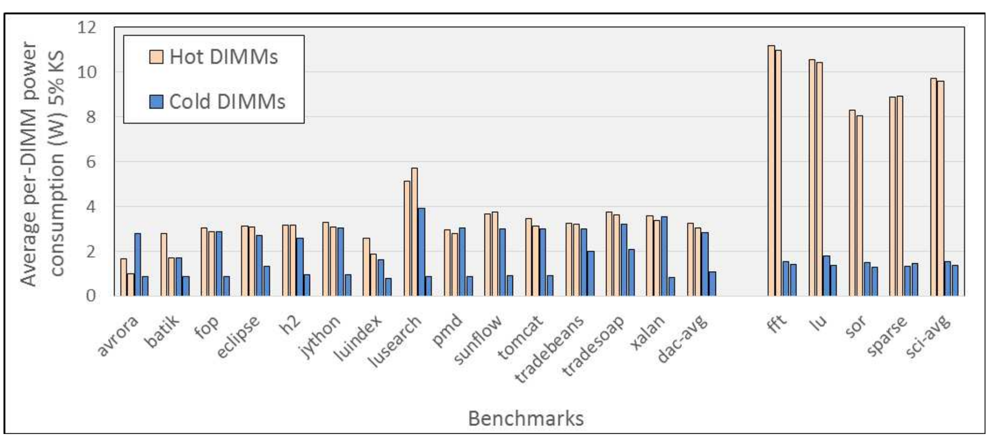


Figure 9. Average power consumed (in W) during benchmark execution for each DIMM with 5% knapsack coloring.

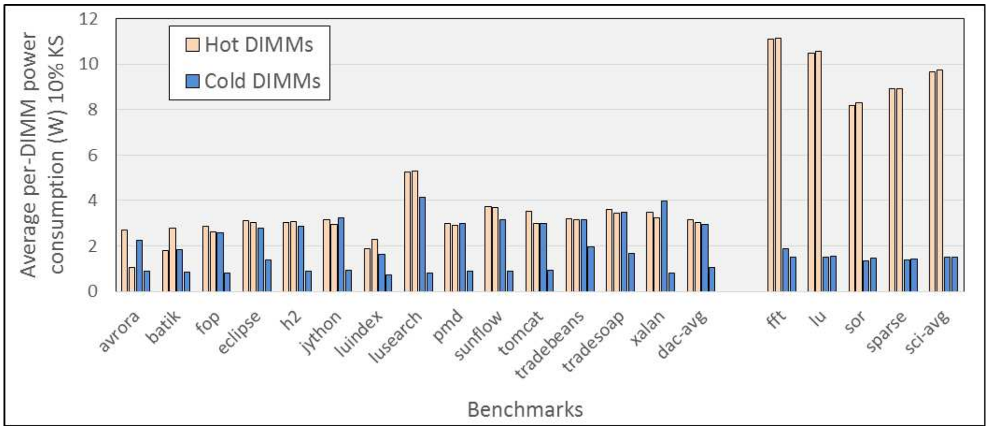


Figure 10. Average power consumed (in W) during benchmark execution with 10% knapsack coloring.

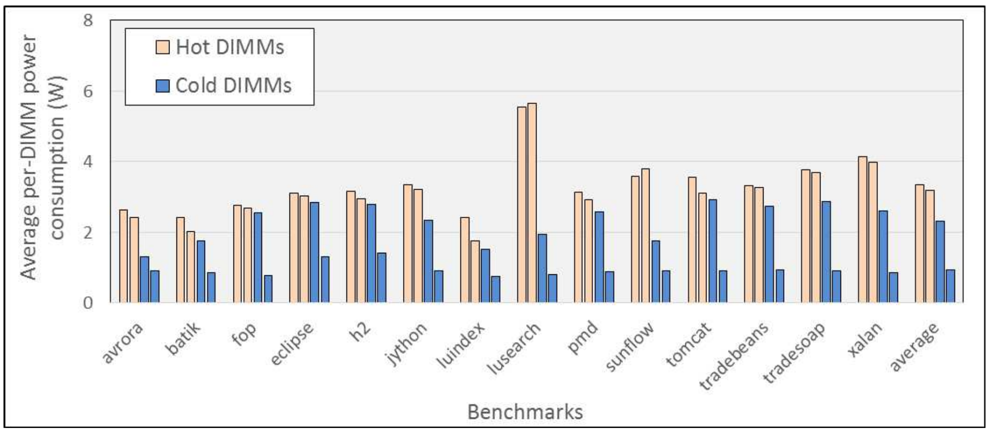


Figure 11. Average power consumed (in W) during benchmark execution with online object partitioning scheme.

8.3 Controlling Per-DIMM Power Consumption

Figure 7 shows the average power consumption (in W) for each DIMM over the entire program run time for our baseline configuration. For each benchmark, we plot four bars corresponding to the four DIMMs in our hardware platform. A higher bar indicates that the corresponding memory device consumed energy at a faster rate during the benchmark run, while lower bars show that the device did not consume much power. Thus, we find that all of the DaCapo benchmarks consume less power than the SciMark workloads. Benchmarks such as *avrora*, *batik*, and *luindex* consume less than 2 watts per DIMM, on average, over the application's run time. On the other hand, the SciMark workloads consume a great deal of DRAM energy. For instance, *fft* requires almost 38 watts across all four DIMMs according to our energy model. Also note that memory usage activity is distributed almost evenly across the ranks. This is an effect of the baseline configuration's physical page allocation policy, which does not attempt to control how physical pages are distributed across memory devices.

Figures 8, 9, and 10 show the same information as Figure 7, but for our runs that employ the offline partitioning strategy with knapsack capacities of 2%, 5%, and 10%, respectively, and Figure 11 shows the results with our online partitioning strategy. For each of our colored runs, we configure our framework to co-locate pages from the hot and cold spaces onto different sets of DIMMs. Thus, the bars in Figures 8–11, are colored differently to distinguish DIMMs that are used for hot objects (orange bars) from ranks that are used for cold objects (blue bars).

In addition to controlling object placement, our framework enables each colored space to select its own allocation strategy to maximize efficiency. The cold space strategy attempts to only select pages from trays that have already furnished another page for similar use. In this way, the strategy for the cold space attempts to minimize the number of memory devices that need to stay powered up. The hot space attempts to select pages from alternating DIMMs in order to increase parallelism.

These results allow us to make several observations:

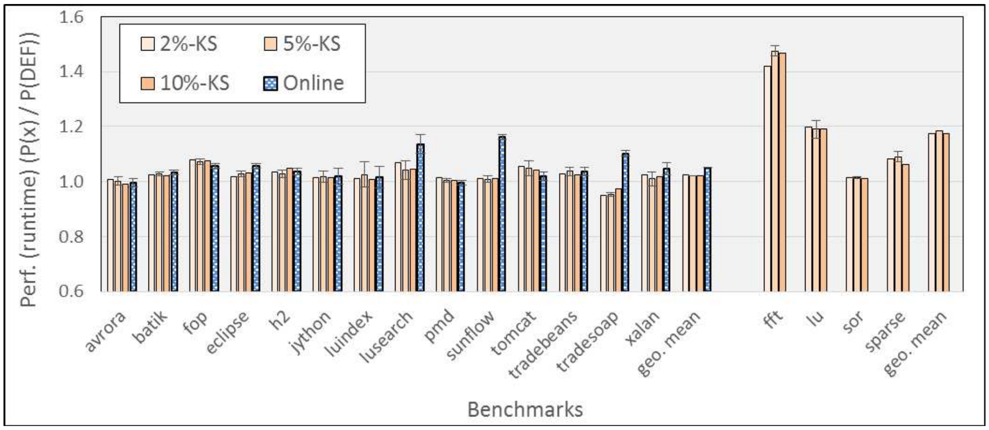


Figure 12. Performance with each colored configuration compared to default.

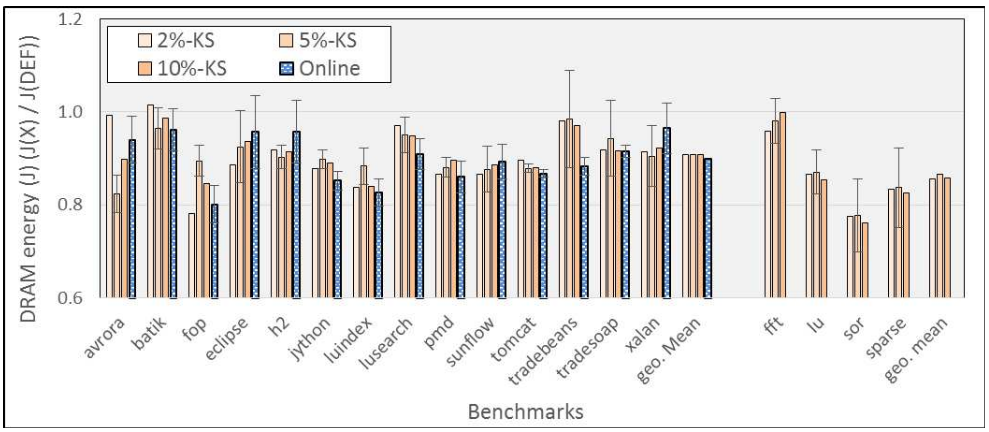


Figure 13. DRAM energy consumed with each colored configuration compared to default.

1. We find that the cold space allocation strategy is very effective at keeping most of the cold ranks in a low power state. This result is expected because most of the guided runs do not allocate enough memory in the cold space to affect more than one DIMM.
2. The figures illustrate the effect of different allocation strategies employed by the hot and cold spaces on the energy profile of individual DIMMs. Thus, while all the hot DIMMs show uniform energy usage, the biased distribution of objects in the cold space produces a skewed energy usage profile for the individual cold DIMMs.
3. We also see that, in some cases, the offline partitioning strategy clearly and effectively distinguishes hot and cold objects. For instance, benchmarks such as *fft* and *lu* contain a majority of cold objects, and the offline strategy accurately separates these objects from the rest of memory. However, in some cases with very little memory activity, such as *avrora* with the 5% knapsack coloring, the memory usage patterns in the profile run do not match the guided run, and the cold ranks reside in a low-power state less often than the hot ranks.
4. In the DaCapo benchmarks, we find a trend of increasing cold memory activity as the capacity for the knapsack coloring used to guide the run is increased, but, on average, this results in only a slight difference in power consumed by the cold DIMMs.
5. The online strategy also consumes slightly less cold memory energy than any of the offline approaches, but, on average, this difference is small.

8.4 Performance and Energy Evaluation

Figures 12, 13, and 14 respectively compare the performance, DRAM energy, and combined (CPU+DRAM) energy consumption of our JVM framework with offline and online partitioning strategies to our baseline for all benchmarks. Similar to our approach for Figures 4(a) and 4(b), we plot 95% confidence intervals for the difference between the means to estimate the degree of variability in these results. To reduce clutter, we omit the error bars for the 2% and 10% knapsack configurations, but note these are similar to the 5% knapsack configuration.

As expected, the performance of some of the workloads with custom JVM framework, such as the SciMark bench-

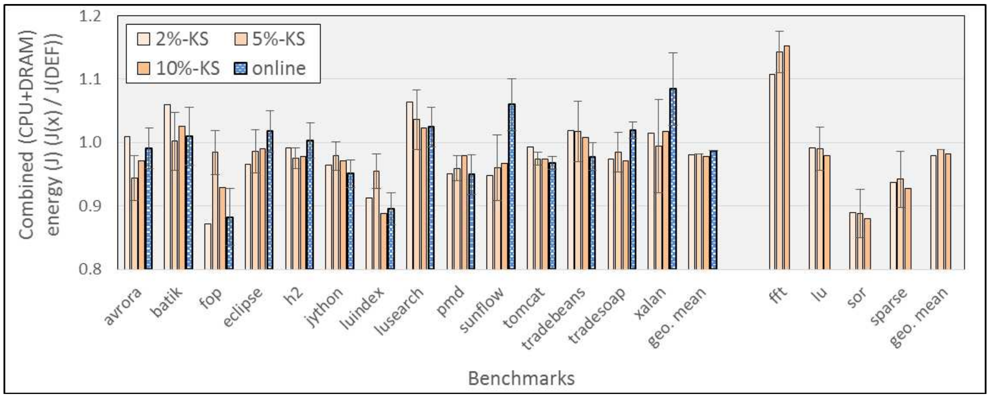


Figure 14. Combined (CPU+DRAM) energy consumed with each colored configuration compared to default.

marks, degrades when compared to the default configuration. The baseline configuration enables more rank and channel parallelism than the custom JVM configurations, and thus, workloads with high bandwidth requirements do not perform as well with the custom JVM. In the worst case (*fft*), this effect causes a performance loss of over 45%. However, with the offline partitioning strategy, benchmarks with low bandwidth requirements often achieve similar performance or only slightly degrade when compared to the baseline configuration. On average, the offline partitioning strategy degrades performance a little over 2% for all three knapsack configurations. With the online strategy, a few benchmarks incur more substantial performance losses (14% for *lusearch* and 16% for *sunflow*), and on average, this technique incurs a 5% performance penalty.

Results plotted in Figure 13 show that our custom JVM is able to achieve our goal of substantially reducing DRAM energy savings over the baseline, regardless of the object partitioning strategy. On average, the offline partitioning strategies reduce DRAM energy consumption by a little over 9% for the DaCapo benchmarks, and by 13% to 14% for the SciMark workloads depending on the knapsack size. The online strategy consumes slightly less DRAM energy than the offline strategies with a 10% average reduction compared to the baseline.

Figure 14 compares the combined (CPU+DRAM) energy consumption of the JVM with our memory-power saving strategies to the baseline approach. We find that most benchmarks and configurations see either a net reduction or no significant impact on combined energy consumption. However, similar to the observation with our MemBench benchmark (Figure 4b), we can see that programs that require higher bandwidth and suffer a performance loss (for example, *fft* and, with the online approach, *sunflow* and *xalan*) consume higher overall combined (CPU+DRAM) energy.[5] One reason for this result is that with our machine configuration many of the benchmarks expend relatively more energy in the CPU than in DRAM. For such workloads, even small performance overheads have a disproportionate effect on combined energy consumption. This effect is especially pronounced for the DaCapo benchmarks, each of which consumes more power in the CPU than in DRAM, and where, on average, CPU energy comprises 60% of the combined energy consumption. Overall, we find that the offline approaches are slightly more efficient than the online approach. The most efficient offline approach (10%-KS) achieves an average combined energy reduction of 2.2% with the DaCapo benchmarks, while the online approach saves 1.2% of combined energy compared to the default configuration.

9. Future Work

Although cross-layer memory management can significantly reduce DRAM energy consumption, our current implementation degrades performance in some cases, which negates some of the (combined) energy savings. Our immediate future work is to address these performance losses in order to realize the full potential of this approach. For instance, we will develop an adaptive online approach to enable the VM to estimate the application's bandwidth requirements and modify its memory management policy to conserve energy without slowing down high-bandwidth applications. Additionally, we have found that our online partitioning strategy incurs a small profiling overhead, which grows as the sampling frequency is increased. We plan to explore the potential of a more asynchronous online profiling approach to overcome this cost and to allow the collection of more accurate profile information. In addition to saving power/energy, we also plan to explore memory management algorithms that take advantage of cross-layer guidance to automatically improve performance. Our planned experiments include: 1) bi-

[5] Note that all of our measurements only consider energy consumed while the benchmark is running, and are not adjusted to account for energy consumed while the machine is *idle*. Thus, when comparing two configurations, the energy consumption of the configuration that finishes early may be under-reported compared to a more realistic scenario where the machine continues to run after the benchmark has completed. Therefore, these results report a lower-bound on the potential benefits to combined energy consumption.

asing the placement of high value data so that performance critical data is distributed widely across memory channels, and 2) tailoring page reclamation policies to access patterns discovered during profiling.

While this work focuses on conventional memory systems in which all of the devices are homogeneous, there is also significant potential for this approach on systems containing different types of memory devices. For example, some systems combine DRAM with non-volatile DIMMs [25, 39] or slower, but higher capacity DIMMs [8] to provide varying capabilities and performance. Implementing cross-layer management on such heterogeneous memory systems will allow applications to automatically adapt to the underlying hardware, potentially exposing powerful new efficiencies. Finally, this work highlights the importance of data object location on a page, and placement of a page in physical memory. We plan to explore and effectively exploit this issue in the future.

10. Conclusion

Memory efficiency has become a very important factor for a wide range of computing domains. We contend that power and performance efficiency cannot be achieved by either the operating system (OS) or the architecture acting alone, but needs guidance from the application and communication with the OS and hardware to ideally allocate and recycle physical memory. In this work, we design the first application-level framework that collects information about data object usage patterns to steer low-level memory management. Our cross-layer approach integrates the HotSpot JVM and a custom Linux kernel to automatically provide efficient distribution of memory resources for a wide range of managed language applications.

We use our framework to show that memory power saving is correlated with the program's bandwidth requirements and performance. Our results indicate that a significant portion of the memory allocated by many programs is *cold*. We find that co-allocating such cold objects on a single memory module can result in significant energy savings in the memory subsystem. Overall, our cross-layer framework shows good potential for enabling existing applications to use memory more efficiently, without requiring any source code modifications or recompilations.

Acknowledgments

We thank the anonymous reviewers for their thoughtful and constructive feedback. This research is supported in part by the National Science Foundation under CNS-1464288 and CAREER award CNS-0953268. Intel Corporation provided computing equipment for experiments in earlier versions of this work.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 45–58. USENIX Association, 1999.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348. USENIX Association, 2012.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, 2006.

[5] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00. USENIX Association, 2000.

[6] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 31–40, 2011.

[7] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[8] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. Memscale: Active low-power modes for main memory. *SIGPLAN Not.*, 47(4):225–238, March 2011.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, December 1995.

[10] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, 2007.

[11] Samuel Z. Guyer and Kathryn S. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 237–250, 2004.

[12] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM.

[13] Martin Hirzel. Data layouts for object-oriented programs. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 265–276, 2007.

[14] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 359–373, 2003.

[15] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[16] HP. Power regulator for proliant servers, 2011.

[17] http://www.intel.com/software/pcm. Intel Performance Counter Monitor, 2012.

[18] http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper 150215.pdf. Memory Management in the Java HotSpot Virtual Machine, April 2006.

[19] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[20] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, New York, NY, USA, 2004. ACM.

[21] Michael R. Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A. Doshi. A framework for application guidance in virtual memory systems. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 155–166, 2013.

[22] JEDEC. Ddr3 sdram standard, 2009.

[23] Alin Jula and Lawrence Rauchwerger. Two memory allocators that use hints to improve locality. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 109–118, 2009.

[24] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, November 1992.

[25] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Phase change memory in enterprise storage systems: Silver bullet or snake oil? *SIGOPS Oper. Syst. Rev.*, 48(1):82–89, May 2014.

[26] A. Kleen. A numa api for linux. *SUSE Labs white paper*, August 2004.

[27] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[28] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 105–116, New York, NY, USA, 2000. ACM.

[29] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, December 2003.

[30] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.

[31] Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 37–48, Washington, DC, USA, 2012. IEEE Computer Society.

[32] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[33] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., Indianapolis, IN, 2008.

[34] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[35] P. Petrov and A. Orailoglu. Virtual page tag reduction for low-power tlbs. In *IEEE International Conference on Computer Design*, pages 371–374, 2003.

[36] SPEC2008. Specjvm2008 benchmarks. http://www.spec.org/jvm2008/, 2008.

[37] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38(1):219–230, March 2010.

[38] Micron Technology Inc. TN-41-01. Calculating Memory System Power for DDR3. Technical report, 2007.

[39] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 325–334, New York, NY, USA, 2011. ACM.

[40] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 103–116, Berkeley, CA, USA, 2006. USENIX Association.

[41] Chengliang Zhang and Martin Hirzel. Online phase-adaptive data layout selection. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 309–334, 2008.

[42] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev 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*, ASPLOS XI, pages 177–188. ACM, 2004.