A Framework for Application Guidance in Virtual Memory Systems

Michael R. Jantz
University of Kansas
mjantz@ittc.ku.edu

Carl Strickland
Arizona State University
cdstrick@asu.edu

Karthik Kumar
Intel Corporation
karthik.kumar@intel.com

Martin Dimitrov
Intel Corporation
martin.p.dimitrov@intel.com

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

Abstract

This paper proposes a collaborative approach in which applications can provide guidance to the operating system regarding allocation and recycling of physical memory. The operating system incorporates this guidance to decide which physical page should be used to back a particular virtual page. The key intuition behind this approach is that application software, as a generator of memory accesses, is best equipped to inform the operating system about the relative access rates and overlapping patterns of usage of its own address space. It is also capable of steering its own algorithms in order to keep its dynamic memory footprint under check when there is a need to reduce power or to contain the spillover effects from bursts in demand. Application software, working cooperatively with the operating system, can therefore help the latter schedule memory more effectively and efficiently than when the operating system is forced to act alone without such guidance. It is particularly difficult to achieve power efficiency without application guidance since power expended in memory is a function not merely of the intensity with which memory is accessed in time but also how many physical ranks are affected by an application's memory usage.

Our framework introduces an abstraction called "colors" for the application to communicate its intent to the operating system. We modify the operating system to receive this communication in an efficient way, and to organize physical memory pages into intermediate level grouping structures called "trays" which capture the physically independent access channels and self-refresh domains, so that it can apply this guidance without entangling the application in lower level details of power or bandwidth management. This paper describes how we re-architect the memory management of a recent Linux kernel to realize a three way collaboration between hardware, supervisory software, and application tasks.

VEE'13, March 16–17, 2013, Houston, Texas, USA.

Categories and Subject Descriptors D.4 [*Software*]: Operating Systems; D.4.2 [*Operating Systems*]: Storage Management—Allocation / deallocation strategies

General Terms Design, Performance

Keywords Memory virtualization, Memory management, Resource allocation, Power, Performance, Containerization

1. Introduction

Recent trends in computer systems include an increased focus on power and energy consumption and the need to support multi-tenant use cases in which physical resources need to be multiplexed efficiently without causing performance interference. When multiplexing CPU, network, and storage facilities among multiple tasks a system level scheduling policy can perform fine-grained reassignments of the resources on a continuing basis to take into account task deadlines, shifting throughput demands, load-balancing needs and supply constraints. Many recent works address how best to allocate CPU time, and storage and network throughputs to meet competing service quality objectives [24, 28], and to reduce CPU power during periods of low demand [1].

By comparison, it is very challenging to obtain precise control over distribution of memory capacity, bandwidth, or power, when virtualizing and multiplexing system memory. That is because these effects intimately depend upon how an operating system binds virtual addresses to physical addresses. An operating system uses heuristics that reclaim either the oldest, or the least recently touched, or a least frequently used physical pages in order to fill demands. Over time, after repeated allocations and reclaims, there is little guarantee that a collection of intensely accessed physical pages would remain confined to a small number of memory modules (or DIMMs). Even if an application reduces its dynamic memory footprint, its memory accesses can remain spread out across sufficiently many memory ranks to keep the ranks from saving much power. The layout of each application's hot pages affects not just which memory modules can transition to lower power states during intervals of low activity, but also how much one program's activity in memory interferes with the responsiveness that other programs experience. Thus a more discriminating approach than is available in current systems for multiplexing of physical memory is highly desirable.

This paper proposes a collaborative approach in which applications can provide guidance to the operating system in allocation and recycling of physical memory. This guidance helps the operat-

ing system take into account several factors when choosing which physical page should be used to back a particular virtual page. Thus, for example, an application whose high-intensity accesses are concentrated among a small fraction of its total address space can achieve power-efficient performance by guiding the operating system to co-locate the active pages among a small fraction of DRAM banks. Recent studies show that memory consumes up to 40% of total system power in enterprise servers [22] making memory power a dominant factor in overall power consumption. Conversely, an application that is very intensive in its memory accesses may prefer that pages in its virtual address span are distributed as widely as possible among independent memory channels; and it can guide the operating system accordingly.

In order to provide these hints without entangling applications into lower level memory management details, we introduce the concept of coloring. Application software or middleware uses a number of colors to signal to the operating system a collection of hints. The operating system takes these hints into account during physical page allocation. Colors are applied against application selected ranges of virtual address space; a color is simply a concise way for an application to indicate to the operating system that some common behavior or intention spans those pages, even if the pages are not virtually contiguous. Colors can be applied at any time and can be changed as well, and the OS makes the best effort possible to take them into account when performing memory allocation, recycling, or page migration decisions. In addition to specializing placement, colors can also be used to signify memory priority or cache grouping, so that an OS can optionally support color-based displacement in order to implement software-guided affinitization of memory to tasks. This is particularly useful in consolidated systems where memory provisioning is important for ensuring performance quality-of-service (QoS) guarantees.

Once an application colors various portions of its range with a few distinct colors, it tells the operating system what attributes (or combinations of attributes) it wants to associate with those colors. Depending upon how sophisticated an operating system implementation is, or what degrees of freedom are available to it in a given hardware configuration, the operating system tunes its allocation, displacement, migration, and power management decisions to take advantage of the information. At the same time, system properties and statistics that are necessary for the application to control its own behavior flow back from the OS, creating a closed feed-back loop between the application and host. That applications can guide the OS at a fine grained level in allocation and placement of pages is also an essential element of adapting applications to systems in which all memory is not homogeneous: for example, NVDIMMs, slower but higher capacity DIMMs, and secondary memory controllers may be used in an enterprise system to provide varying capabilities and performance. Mobile platforms represent another potential application as system-on-chip (SoC) designs are beginning to incorporate independently powered memory banks. The approach proposed in this paper takes a critical first step in meeting the need for a fine-grained, power-aware, flexible provisioning of memory.

The physical arrangement of memory modules, and that of the channels connecting them to processors, together with the power control domains are all opaque to applications in our approach. In line with what has come to be expected from modern computer systems, our approach virtualizes memory and presents the illusion to every task that it has at its disposal a large array of memory locations. By hiding the lower level physical details from applications we preserve the benefits of modularity – applications do not need to become hardware aware in order to deliver memory management guidance to the operating system. Comparatively minor changes to the operating system bring about the necessary three way collaboration between hardware, supervisory software, and application tasks. We re-architect the memory management of a recent Linux kernel in order to achieve this objective. The OS reads the memory hardware configuration and constructs a software representation (called "trays") of all the power manageable memory units. We used Linux kernel version 2.6.32 as a vehicle upon which to implement trays. We modified the kernel's page management routines to perform allocation and recycling over trays. We created application programming interfaces (APIs) and a suite of tools by which applications can monitor memory resources in order to implement coloring and associating intents with colors.

Our framework is the first to provide a system-level implementation of flexible, application-guided memory management. It supports such usage scenarios as prioritization of memory capacity and memory bandwidth, and saving power by transitioning more memory ranks into self-refresh states. It is easy to admit new scenarios such as application guided read-ahead or page prefill on a subset of ranges, differential placement of pages between fast and slow memory, and aggressive recycling of pages that applications can flag as transient. The major contributions of this work are:

- We describe in detail the design and implementation of our framework, which we believe is the first to provide for three way collaboration between hardware, supervisory software, and application tasks.
- We show how our framework can be used to achieve various objectives, including power savings, capacity provisioning, performance optimization, etc.

The next section places the contributions of this work in the context of related work and is followed by a description of relevant background information. We then describe the detailed design of our framework and present several experiments to showcase and evaluate potential use cases of application-guided memory management. We finally discuss potential future work before concluding the paper.

2. Related Work

Researchers and engineers have proposed various power management schemes for memory systems. Bi et. al. [4] suggest predicting memory reference patterns to allow ranks to transition into low power states. Delaluz et. al. [8] track memory bank usage in the operating system and selectively turn banks on and off at context switch points to manage DRAM energy consumption. Along these same lines, *memory compaction* has recently been integrated into the Linux kernel [7]. This technique, which reduces external fragmentation, is also used for power management because it allows memory scattered across banks to be compacted into fewer banks, in a way that is transparent to the application. Fan et. al. [10] employ petrinets to model and evaluate memory controller policies for manipulating multiple power states. Lin et. al. [23] construct an adaptive thread grouping and scheduling framework which assigns groups of threads to exclusive DRAM channels and ranks in order to jointly manage memory performance, power, and thermal characteristics. While these techniques employ additional hardware and system-level analysis to improve memory power management, our framework achieves similar objectives by facilitating collaboration between the application and host system.

Other works have explored integrating information at the application-level with the OS and hardware to aid resource management. Some projects, such as Exokernel [9] and Dune [3], attempt to give applications direct access to physical resources. In contrast to these works, our framework does not expose any physical structures or privileged instructions directly to applications. More similar to this work are approaches that enable applications to share

additional information about their memory usage with lower levels of memory management. Prior system calls, such as *madvise* and *vadvise*, and various NUMA interfaces [20] have allowed applications to provide hints to the memory management system. Some API's (such as Android's ashmem [12] and Oracle's Transcendent Memory [25]) allow the application or guest OS to allocate memory that may be freed by the host at any time (for instance, to reduce memory pressure). 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 [6] 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. While these works evince some of the benefits of increased collaboration between applications and the OS, the coloring API provided by our framework enables a much broader spectrum of hints to be overlapped and provides a concise and powerful way to say multiple things about a given range of pages.

Also related is the concept of cache coloring [19], where the operating system 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 facilitate communication between the application and system-level memory management.

Finally, prior work has also explored virtual memory techniques for energy efficiency. Lebeck et. al. [21] propose several policies for making page allocation power aware. Zhou et. al. [29] 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. [26] propose virtual page tag reduction for low-power translation look-aside buffers (TLBs). Huang et. al. [18] 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 this work, however, these systems do not provide any sort of closed-loop feedback between application software and lower level memory management, and thus, are vulnerable to learning inefficiencies as well as inefficiencies resulting from the OS and application software working at cross purposes.

3. Background

In order to understand the design and intuition of our framework, we first describe how current memory technologies are designed and viewed from each layer of the vertical execution stack, from the hardware up to the application.

Modern server systems employ a Non-Uniform Memory Access (NUMA) architecture which divides memory into separate regions (*nodes*) for each processor or set of processors. Within each NUMA node, memory is spatially organized into *channels*. Each channel employs its own memory controller and contains one or more DIMMs, which, in turn, each contain two or more *ranks*. Ranks comprise the actual memory storage and typically range from 2GB to 8GB in capacity. The memory hardware performs aggressive power management to 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. Thus, different memory allocation strategies must consider an important power-performance tradeoff: distributing memory evenly across the ranks improves bandwidth which leads to better performance, while minimizing the number of active ranks consume less power.

The BIOS is responsible for reading the memory hardware configuration and converting it into physical address ranges used in the operating system. The BIOS provides several *physical address interleaving* configurations, which control how the addresses are actually distributed among the underlying memory hardware units. Different physical address interleaving configurations can have significant power-performance implications. For example, systems tuned for performance might configure the BIOS to fully interleave physical addresses so that consecutive addresses are distributed across all the available ranks.

On boot, the operating system reads the physical address range for each NUMA node 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. Different zone types are suitable for different types of usage (e.g. the lowest 16MB of physical memory, which certain devices require, is placed in the *DMA* zone). Next, the operating system creates physical page frames (or simply, *pages*) from the address range covered by each zone. Each page typically addresses 4KB of space. 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.

Finally, the operating system provides each process with its own virtual address space for managing memory at the application level. Virtual memory relieves applications of the need to worry about the size and availability of physical memory resources. Despite these benefits, virtualization adds a layer of abstraction between the application and operating system which makes it impossible for the lower-level memory management routines to derive the purpose of a particular memory allocation. Furthermore, even at the operating system level, many of the details of the underlying memory hardware necessary for managing memory at the rank level have been abstracted away as well. Thus, excessive virtualization makes it extremely difficult to design solutions for applications which require more fine-grained controls over memory resource usage.

4. Application-Guided Memory Management

This section describes the design and implementation of our framework. Enabling applications to guide management of memory hardware resources requires two major components:

1. An interface for communicating to the operating system 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

We address the first component in the next subsection, which describes our *memory coloring* framework for providing hints to the operating system about how applications intend to use memory resources. Next, we describe the architectural modifications we made to the system-level memory management software to enable management of individual memory hardware units.

4.1 Expressing Application Intent through Colors

A color is an abstraction which allows the application to communicate to the OS hints about how it is going to use memory resources. Colors are sufficiently general as to allow the application to provide different types of performance or power related usage hints. In using colors, application software can be entirely agnostic about how virtual addresses map to physical addresses and how those physical addresses are distributed among memory modules. By coloring any *N* different virtual pages with the same color, an application communicates to the OS that those *N* virtual pages are alike in some significant respect, and by associating one or more attributes with that color, the application invites the OS to apply any discretion it may have in selecting the physical page frames for those *N* virtual pages. As one rather extreme but trivial example, suppose that the application writer uses one color for *N* virtual pages and then binds with that color a guidance to "use no more than 1 physical page frame" for that color. The OS, if it so chooses, can satisfy a demand fault against any of those *N* page addresses simply by recycling one physical page frame by reclaiming it from one mapping and using it for another. Or, more practically, the OS may simply interpret such a guidance to allocate normally but then track any page frames so allocated, and reclaim aggressively so that those particular page frames are unlikely to remain mapped for long. A scan-resistant buffer-pool manager at the application level may benefit from this kind of guidance to the operating system.

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. An application that uses no colors and therefore provides no guidance is treated normally– that is, the OS applies some default strategy. And even when an application provides extensive guidance through coloring, depending on the particular version of the operating system, the machine configuration (such as how much memory and how finely interleaved it is), and other prevailing run time conditions in the machine, the OS may veer little or a lot from a default strategy. The specializations that an OS may support need not be confined just to selection of physical pages to be allocated or removed from the application's resident set, and they may include such other options as whether or not to fault-ahead or to perform read-aheads or flushing writes; whether or not to undertake migration of active pages from one set of memory banks to another in order to squeeze the active footprint into fewest physical memory modules. In this way, an OS can achieve performance, power, I/O, or capacity efficiencies based on guidance that application tier furnishes through coloring.

Using colors to specify intents instead of specifying intents directly (through a system call such as `madvise`) is motivated by three considerations- (a) efficiency through conciseness – an application can create the desired mosaic of colors and share it with the OS, instead of having to specify it a chunk at a time, and (b) the ability to give hints that say something horizontal across a collection of pages with the same color, such as, "it is desirable to perform physical page re-circulation among this group of virtual addresses", or, "it is desirable to co-locate the physical pages that happen to bind to this set of virtual addresses", etc., and (c) modularity – the capabilities supported by a particular version of an OS or a particular choice of hardware and system configuration may not support the full generality of hints that another version or configuration can support. An application developer or deployer, or some software tool, can bind colors to the menu of hints at load time or run time. This flexibility also means that even at run time, colors can be altered on the basis of feedback from the OS or guidance from a performance or power tuning assistant. In our prototype implementation, colors are bound to hints by a combination of configuration files and library software. A custom system call actually applies colors to virtual address ranges.

Let us illustrate the use of colors and hints with a simple example. Suppose we have an application that has one or more address space extents in which memory references are expected to be relatively infrequent (or uniformly distributed, with low aggregate probability of reference). The application uses a color, say *blue* to color these extents. At the same time, suppose the application has a particular small collection of pages in which it hosts some frequently accessed data structures, and the application colors this collection *red*. The coloring intent is to allow the operating system to manage these sets of pages more efficiently – perhaps it can do so by co-locating the *blue* pages on separately power-managed units from those where *red* pages are located, or, co-locating *red* pages separately on their own power-managed units, or both. A possible second intent is to let the operating system page the *blue* ranges more aggressively, while allowing pages in the *red* ranges an extended residency time. By locating *blue* and *red* pages among a compact group of memory ranks, an operating system can increase the likelihood that memory ranks holding the *blue* pages can transition more quickly into self-refresh, and that the activity in *red* pages does not spill over into those ranks. (Other possible tuning options may be desirable as well – for example, allowing a higher clock frequency for one set of memory ranks and reducing clock frequency on others, if that is supported by the particular OS-hardware mix). Since many usage scenarios can be identified to the operating system, we define "intents" and specify them using configuration files. A configuration file with intents labeled *MEM-INTENSITY* and *MEM-CAPACITY* can capture two intentions: (a) that red pages are hot and blue pages are cold, and (b) that about 5% of application's dynamic resident set size (RSS) should fall into red pages, while, even though there are many blue pages, their low probability of access is indicated by their 3% share of the RSS.

1. Alignment to one of a set of standard intents:
   `INTENT MEM-INTENSITY`
2. Further specification for containing total spread:
   `INTENT MEM-CAPACITY`
3. Mapping to a set of colors:
   `MEM-INTENSITY RED 0` //hot pages
   `MEM-CAPACITY RED 5` //hint- 5% of RSS
   `MEM-INTENSITY BLUE 1` //cold pages
   `MEM-CAPACITY BLUE 3` //hint- 3% of RSS

Next let us give an overview of the approach we have taken in implementing a prototype memory management system for receiving and acting upon such guidance. In our implementation, we have organized memory into power-management domains that are closely related to the underlying hardware. We call these management units *trays*, and we map colors to trays and tray based memory allocation and reclaim policies, as described in the following section.

4.2 Memory Containerization with Trays

We introduce a new abstraction called "trays" to organize and facilitate memory management in our framework. A *tray* is a software structure which contains sets of pages that reside on the same power-manageable memory unit. Each zone contains a set of trays and all the lists used to manage pages on the zone are replaced with corresponding lists in each tray. Figure 1 shows how our custom

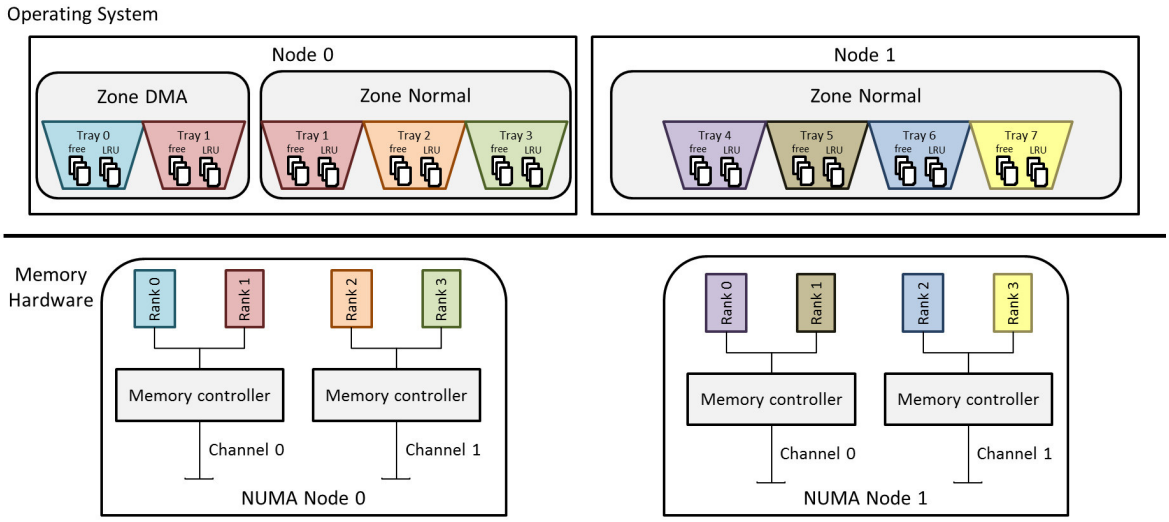


Figure 1. Physical memory representation in the Linux kernel with trays as it relates to the system's memory hardware.

kernel organizes its representation of physical memory with trays in relation to the actual memory hardware.[1]

We have used Linux kernel version 2.6.32 as the baseline upon which to implement trays. The kernel's page management routines, which operate on lists of pages at the zone level were modified quite easily no 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. While most of these changes are straightforward, the breadth of routines that require modification make the size of our patch substantial. (On last accounting, our kernel patch included modifications to approximately 1,800 lines of code over 34 files). This approach has the advantage that trays are defined as objects with attributes on each zone, which we find easier to reason about and maintain than another approach we considered: implicitly defining a "tray dimension" on each of the memory manager's lists of pages. Finally, this design requires less additional space overhead compared to other approaches. In contrast to the memory region approach proposed by A. Garg [13], which duplicates the entire zone structure for each memory hardware unit, the only structures we duplicate are pointers to list heads for each list of pages in each tray.

Assigning pages to the appropriate tray requires a mapping from the physical addresses served up by the BIOS to the individual memory hardware units. The ACPI 5.0 specification defines a memory power state table (MPST) which exposes this mapping in the operating system [14]. Unfortunately, at the time of this writing, ACPI 5.0 has only been recently released, and we are not able to obtain a system conforming to this specification. Therefore, as a temporary measure, in our prototype implementation we construct the mapping manually from a knowledge of the size and configuration of memory hardware in our experimental system. Specifically, we know that each memory hardware unit stores the same amount of physical memory (2GB, in our case). We can also configure the BIOS to serve up physical addresses sequentially (as opposed to interleaving addresses across memory hardware units). Now, we can compute physical address boundaries for each tray in our system *statically* using the size of our memory hardware units. Finally, at runtime, we simply map each page into the appropriate tray via the page's physical address. Note that, while, in our system, this mapping is simple enough that it does not require any additional per-page storage, a more complex mapping might require storage of tray information on each page after computing it once during initialization. Our immediate future work will be to rebase the implementation on an ACPI 5.0 compliant kernel that has the MPST information available to the Linux kernel at the point of handoff from BIOS.

[1] The page interleaving shown in Figure 1 is for pictorial simplicity and is not a restriction.

5. Experimental Setup

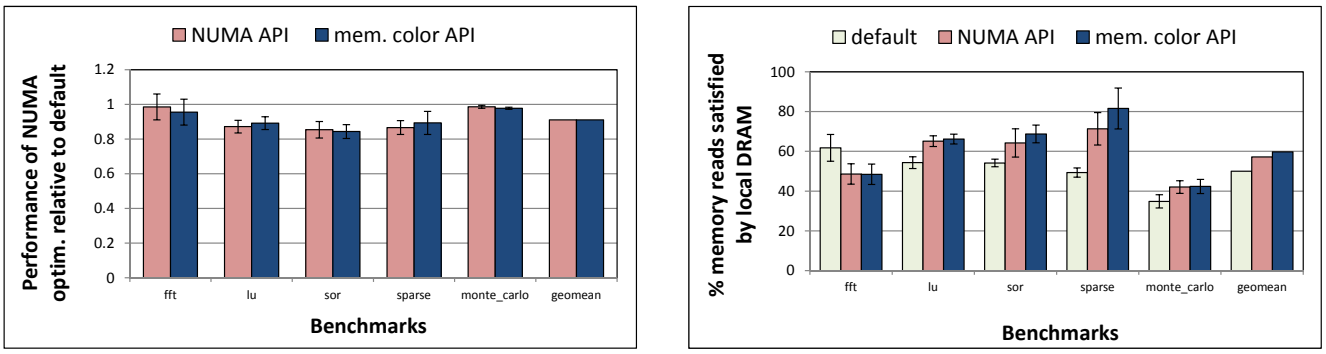
This work presents several experiments to showcase and evaluate our framework's capabilities. In this section, we describe our experimental platform as well as the tools and methodology we use to conduct our experiments, including how we measure power and performance.

5.1 Platform

All of the experiments in this paper were run on an Intel 2600CP server system with two Intel Xeon E5-2680 sockets (codename "Sandy Bridge"). Each socket has 8 2.7GHz cores with hyper-threading enabled and 16 GB of DDR3 memory (for a total of 32 threads and 32 GB of memory). Memory is organized into four channels per socket and each channel contains exactly one DIMM (with two 2 GB ranks each). We install 64-bit SUSE Linux Enterprise Server 11 SP 1 and select a recent Linux kernel (version 2.6.32.59) as our default operating system. The source code of this Linux kernel (available at `kernel.org`) provides the basis of our framework's kernel modifications.

5.2 The HotSpot Java Virtual Machine

Several of our experiments use Sun/Oracle's HotSpot Java Virtual Machine (build 1.6.0_24) [16]. The latest development code for the HotSpot VM is available through Sun's OpenJDK initiative. The HotSpot VM provides a large selection of command-line options, such as various JIT compilation schemes and different garbage collection algorithms, as well as many configurable parameters to tune the VM for a particular machine or application. For all of our experiments with HotSpot, we select the default configuration for server-class machines [15]. We conduct our HotSpot VM experiments over benchmarks selected from two suites of applications: SPECjvm2008 [17] and DaCapo-9.12-bach [5]. Each suite employs a *harness* program to load and iterate the benchmark applications



(a) (b)

Figure 2. Comparison of implementing the HotSpot NUMA optimization with the default NUMA API vs. our memory coloring framework (a) shows the performance of each implementation relative to the default HotSpot performance. (b) shows the % of NUMA-local memory reads with each configuration.

multiple times in the same run. The SPECjvm2008 harness continuously iterates several benchmark operations for each run, starting a new operation as soon as a previous operation completes. Each run includes a warmup period of two minutes (which is not counted towards the run score) and an iteration period of at least four minutes. The score for each run is the average time it takes to complete one operation during the iteration period. Similarly, the DaCapo harness executes each benchmark operation a specified number of iterations per run. For these applications, we execute the benchmark a total of 31 iterations per run and take the median runtime of the final 30 iterations as the score for each run. For all of our experiments, we report the average score of ten runs as the final score for the benchmark. We discuss the particular applications we use as well as further details about the HotSpot VM relevant to each experiment in the subsequent experimental sections.

5.3 Application Tools for Monitoring Resources

We designed several tools based on custom files in the `/proc` filesystem to provide applications with memory resource usage information. These tools provide an "on-demand" view of the system's memory configuration as well as an estimate of the total and available memory for each memory hardware unit (rank). Additionally, we have written tools to map specific regions of a process' virtual address space to the memory units backing these regions. These tools are essential for applications which require feedback from the OS to control their own memory usage.

5.4 DRAM Power Measurement

In addition to basic timing for performance measurements, many of our experiments read various activity counters (registers) to measure the impact of our framework with additional metrics, such as power consumption. We estimate DRAM power consumption with Intel's *power governor* library. The power governor framework employs sampling of energy status counters as the basis of its power model. The accuracy of the model, which is based on proprietary formulae, varies based on the DRAM components used. For our experiments, we run a separate instance of a power governor based-tool during each run to compute a DRAM power estimate (in Watts) every 100ms. We report the mean average of these estimates as the average DRAM power consumption over the entire run.

6. Emulating the NUMA API

Modern server systems represent and manage memory resources at the level of individual NUMA nodes. These systems typically provide a library and/or set of tools for applications to control how their memory usage is allocated across NUMA nodes. In contrast, our framework enables the operating system to manage resources at the more fine-grained level of individual hardware units. While our system provides more precise control, it is also powerful enough to emulate the functionality of tools designed specifically for managing memory at the level of NUMA node. In this section, we demonstrate that our framework effectively emulates the NUMA API by implementing a NUMA optimization in the HotSpot JVM with our framework.

6.1 Exploiting HotSpot's Memory Manager to improve NUMA Locality

The HotSpot JVM employs *garbage collection* (GC) to automatically manage the application's memory resources. The HotSpot memory manager allocates space for the application heap during initialization. As the application runs, objects created by the application are stored in this space. Periodically, when the space becomes full or reaches a preset capacity threshold, the VM runs a collection algorithm to free up space occupied by objects that are no longer reachable. The HotSpot garbage collector is *generational*, meaning that HotSpot divides the heap into two different areas according 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. Objects that survive some number of young generation collections are eventually promoted, or tenured, to the old generation.

On NUMA systems, this generational organization of memory can be exploited to improve DRAM access locality. The critical observation is thus: *newer objects are more likely to be accessed by the thread that created them*. Therefore, binding new objects to the same NUMA node as the thread that created them should reduce the proportion of memory accesses on remote NUMA nodes, and consequently, improve performance. In order to implement this optimization, HotSpot employs the NUMA API distributed with Linux. During initialization, HotSpot divides the eden space into separate virtual memory areas and informs the OS to back each area with physical memory on a particular NUMA node via the `numa_tonode_memory` library call. As the application runs, HotSpot keeps track of which areas are bound to which NUMA node and ensures that allocations from each thread are created in the appropriate area.

To implement the NUMA optimization in our framework, we create separate *node affinitization* colors for each NUMA node in the system. Next, we simply replace each call to `numa_tonode_memory` in HotSpot with a call to color each eden space area with the appropriate node affinitization color. The OS interprets this color as a

hint to bind allocation for the colored pages to the set of trays corresponding to memory hardware units on a particular NUMA node. In this way, the node affinitization colors emulate the behavior of the `numa_tonode_memory` library call.

6.2 Experimental Results

We conduct a series of experiments to verify that our framework effectively implements the NUMA optimization in HotSpot. For these experiments, we select the SciMark 2.0 subset of the SPECjvm2008 benchmarks with the large input size. This benchmark set includes five computational kernels common in scientific and engineering calculations and is designed to address the performance of the memory subsystem with out-of-cache problem sizes [17]. We found that enabling the HotSpot NUMA optimization significantly affects the performance of several of these benchmarks, making them good candidates for these experiments. We use the methodology described in Section 5.2 to measure each benchmark's performance. We also employ activity counters to measure the ratio of memory accesses satisfied by NUMA-local vs. NUMA-remote DRAM during each run.

Figure 2(a) shows the performance of each implementation of the NUMA optimization relative to the default configuration's performance of each benchmark. In Figure 2(a), lower bars imply better performance (e.g., the *lu* benchmark runs about 15% faster with the NUMA optimization enabled compared to the default HotSpot configuration). Also, for each result, we plot 95% confidence intervals using the methods described by Georges et. al. [11], and the rightmost bar displays the average (geometric mean) of all the benchmarks. It can be observed that the NUMA optimization improves performance for three of the five SciMark benchmarks, yielding about a 9% average improvement. More importantly, the performance results for each implementation of the NUMA optimization are very similar across all the benchmarks, with the differences between each implementation always within the margin of error. On average, the two implementations perform exactly the same. Similarly, Figure 2(b) plots the percentage of total memory accesses satisfied by NUMA-local memory for each configuration. As expected, the optimization increases the proportion of memory accesses satisfied by local DRAM for most of the benchmarks. Interestingly, the percentage of local memory accesses for one benchmark (*fft*) actually reduces with the optimization enabled. However, in terms of emulating the NUMA API, while these results show slightly more variation between the two implementations, the differences, again, are always within the margin of error. Thus, our framework provides an effective implementation of the NUMA optimization in HotSpot. Furthermore, this experiment shows that our design is flexible enough to supersede tools which exert control over memory at the level of NUMA nodes.

7. Memory Priority for Applications

Our framework provides the architectural infrastructure to enable applications to guide memory management policies for all or a portion of their own memory resources. In this section, we present a "first-of-its-kind" tool which utilizes this infrastructure to enable memory prioritization for applications. Our tool, called *memnice*, allows applications to prioritize their access to memory resources by requesting alternate management policies for low priority memory. Later, we showcase an example of using *memnice* to prioritize the memory usage of a Linux kernel compile.

7.1 *memnice*

Operating systems have long had the ability to prioritize access to the CPU for important threads and applications. In Unix systems, processes set their own priority relative to other processes via the *nice* system call, and the process scheduler uses each process' nice value to prioritize access to the CPU. Prioritizing access to memory resources is much more difficult due to the varying time and space requirements for memory resources and the layers of abstraction between the application and memory hardware. Our framework facilitates solutions for each of these issues enabling us to create the first-ever tool for prioritizing access to memory resources: *memnice*.

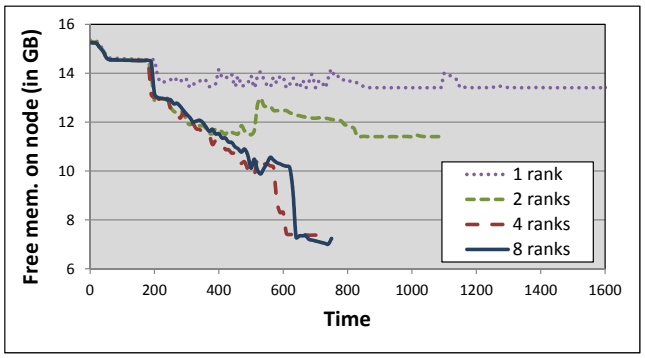


Figure 3. Free memory available during kernel compilations with different memory priorities

In our initial implementation, *memnice* accomplishes memory prioritization by limiting the fraction of physical memory against which allocations can happen[2]. Consider a scenario in which several low-priority applications and one high-priority application compete over the same pool of memory. If left alone, the low-priority applications can expand their memory footprint contending with memory allocations of a higher-priority application, and forcing premature reclaims against a higher-priority application's pages.

In order to ensure the high-priority application runs smoothly, we can restrict the low-priority applications from using certain portions of memory using *memnice*. To implement this scheme in our framework, we enable applications to color portions of their address spaces to reflect the urgency with which pages must be stolen from elsewhere to meet demand fills. In the operating system, we restrict low-priority allocations to only search for free pages from an allowed set of trays. In this way, the low-priority applications are constrained to a smaller fraction of the system's memory, and the high-priority application can call upon larger fractions of system's memory during demand fills.

7.2 Using *memnice* with Kernel Compilation

We showcase our implementation by running several Linux kernel compilations with different memory priority settings. Kernel compilation, unless restricted, can allocate and keep a large number of pages busy, as it proceeds through reading, creating, and writing, a lot of files, including temporary files, from each of a number of threads that run in parallel. For each experiment, we use the *memnice* utility, which is implemented as a simple command-line wrapper, to set the memory priority for the entire address space of the compilation process. We also configured our framework so that children of the root compilation process would inherit its memory priority and apply it to their own address spaces. We run all of our compilations on one node of the system described in Section 5.1.

[2] We chose a simple way to force prioritization. In a more refined implementation we would use colors to signal to the operating system that it should force a colored set to do color-local recycling subject to a minimum length of time for which pages are kept mapped before being recycled, to prevent thrashing

For each compilation, we employ four different memory priority configuration to restrict the compilation to a different set of memory resources in each run. These configurations restrict the compilation to use either 1 tray, 2 tray, 4 tray, or all 8 tray of memory on the node. Finally, during each run, we sampled (at a rate of once per second) the `proc` filesystem to estimate the amount of free memory available on the node.

Each line in Figure 3 shows the free memory available on the node during kernel compilations with differing memory priorities. As we can see, each compilation proceeds at first using the same amount of memory resources. However, at around the 200th sample, the compilation restricted to use memory from only 1 tray is constrained from growing its memory footprint any larger than 2GB (recall that our system contains 2GB trays). Similarly, the compilation restricted to two trays stops growing after its footprint reaches about 4GB. The compilations restricted to 4 and 8 trays proceed in pretty much the same way for the entire run, presumably because kernel compilation does not require more than 8GB of memory. Finally, as expected, compilations restricted to use fewer memory resources take much longer to complete. Thus, *memnice* properly constrains the set of memory resources available for each application and is an effective tool for providing efficient, on-demand prioritization of memory resources.

8. Reducing DRAM Power Consumption

Memory power management in current systems occurs under the aegis of a hardware memory controller which transitions memory ranks into low power states (such as ”self-refresh”) during periods of low activity. To amplify its effectiveness, it is desirable that pages that are very lightly accessed are not mixed up with pages that are accessed often within the same memory ranks. In this section, we show how our framework can bring about periods of low activity more consciously at a memory rank level, instead of relying on such an effect resulting from just happenstance.

For all the experiments in this section, we use a utility that we call *memory scrambler* to simulate memory state on a long-running system. As its name suggests, the“memory scrambler” is designed to keep trays from being wholly free or wholly allocated – it allocates chunks of memory until it exhausts memory, then frees the chunks in random order until a desired amount of memory is freed up (holding down some memory), effectively occupying portions of memory randomly across the physical address space. We perform the experiments described in this section using 16GB of memory from one socket of the server. Before each experiment, we run the scrambler, configured such that it holds down 4GB of memory after it finishes execution.

8.1 Potential of Containerized Memory Management to Reduce DRAM Power Consumption

The default Linux kernel views memory as a large contiguous array of physical pages, sometimes divided into NUMA nodes. That this array is actually a collection of independent power-managed ranks at the hardware level is abstracted away at the point that the kernel takes control of managing the page cache. Over time, as pages become allocated and reclaimed for one purpose after another, it is nearly impossible to keep pages that are infrequently accessed from getting mixed up with pages that are frequently accessed in the same memory banks. Thus, even if only a small fraction of pages are actively accessed, the likelihood remains high that they are scattered widely across all ranks in the system.

The use of *trays* in our customization of the default kernel makes it possible to reduce this dispersion. Trays complement coloring– the application guides the kernel so that virtual pages that are frequently accessed can be mapped to physical pages from one set of trays, while those that are not, can be mapped to physical

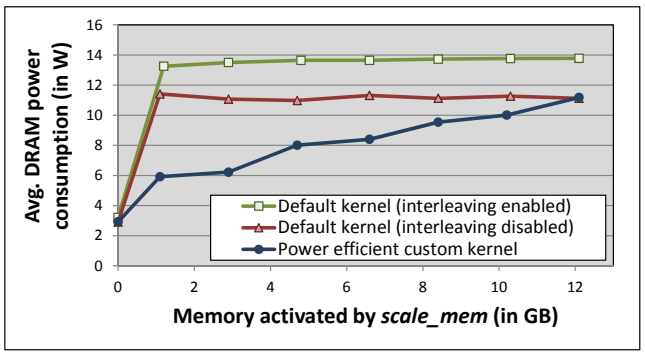


Figure 4. Relationship between memory utilization and power consumption on three different configurations

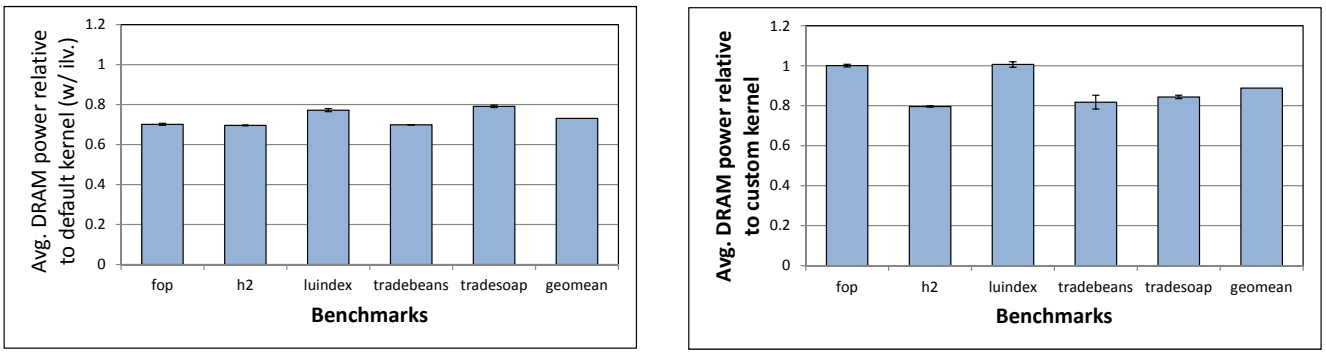
pages from the remaining trays. If performance demand is such that very high degree of interleaving is requested by an application, the operating system may need to spread out page allocation among more trays; but if that is not necessary then the operating system can keep page allocation biased against such a spread-out. In order to demonstrate the power-saving potential of our framework, we designed a “power-efficient” memory management configuration. We opt, when allocating a page, to choose a tray that has furnished another page for similar use. In this way we reduce the number of additional memory ranks that need to stay powered up.

To keep the first experiment simple, we have used a simplified workload. In this workload, we don’t need application coloring because the workload simply allocates increasing amounts of memory in stages, so that in each stage it just allocates enough additional memory to fit in exactly one power-manageable unit. This unit is 2GB, in our case. In each stage, the workload continuously reads and writes the space it has allocated, together with all of the space it allocated in previous stages, for a period of 100 seconds. During each stage, we use power governor (Section 5.4) to measure the average DRAM power consumption.

We compare our custom kernel running the staged workload to the default kernel with two configurations: one with physical addresses interleaved across the memory hardware units (the default in systems tuned for performance) and another with physical addresses served up sequentially by the BIOS. Recall that our custom kernel requires that physical addresses are served up sequentially in order to correctly map trays onto power-manageable units.

Figure 4 shows the average DRAM power consumption during each stage for each system configuration. Thus, during stages when only a fraction of the total system memory is active, our custom kernel consumes much less power than the default kernel. Specifically, during the first stage, the custom kernel consumes about 55% less DRAM power than the default kernel with physical address interleaving enabled and 48% less than the default kernel with interleaving disabled. This is because, with no other processes actively allocating resources, the containerized kernel is able to satisfy the early stage allocations one memory unit at a time. The default kernel, however, has no way to represent power-manageable memory units and will activate memory on every hardware unit during the first stage. As the workload activates more memory, the custom kernel activates more memory hardware units to accommodate the additional resources and eventually consumes as much DRAM power as the default kernel (with interleaving disabled).

In sum, these experiments show that our framework is able to perform memory management over power-manageable units and that this approach has the potential to significantly reduce DRAM power consumption.



(a) (b)

Figure 5. Local allocation and recycling reduces DRAM power consumption. (a) shows DRAM power relative to the default kernel (with interleaving enabled) and (b) shows the results relative to the custom kernel without local allocation and recycling.

8.2 Localized Allocation and Recycling to Reduce DRAM Power Consumption

In this section, we show that basic capabilities of our framework can be used to reduce DRAM power consumption over a set of benchmark applications. We create colors to enable applications to restrict their memory allocation and recycling to a population of pages that become allocated to each color (i.e., the OS performs color-local recycling). By restricting allocations of a given color to a subset of memory ranks, the operating system can translate color-confinement into confinement of the active memory footprints to the corresponding groups of ranks. We then measure the impact on power and performance from such confinements.

For these experiments, we selected five Java benchmarks from the DaCapo-9.12-bach benchmark suite for their varying memory usage characteristics. The *h2*, *tradebeans*, and *tradesoap* applications execute thousands of transactions over an in-memory database and require relatively large memory footprints with non-uniform access patterns. In contrast, *fop*, which converts files to PDF format, and *luindex*, which indexes documents using the lucene library, require smaller footprints with more uniform access patterns.

We employ an experimental setup similar to the setup we use in the previous section. All experiments are run on one node of our server system and we again employ the memory scrambling tool to occupy random portions of the physical address space during each run. Each benchmark is run within the HotSpot JVM and we record performance and DRAM power measurements as described in Section 5. Each of the selected benchmarks requires no more than 4GB of total memory resources and we run the benchmarks one at a time. Thus, for each experimental run, we color the entire virtual address space of each benchmark to bind all memory allocation and recycling to a single 4GB DIMM.

Figure 5(a) shows the ratio of DRAM power consumed for each benchmark, between when it is run with local allocation and recycling and when it is run on the default kernel. As the figure shows, the local allocation and recycling consumes significantly less DRAM power than the default kernel configuration; indeed, DRAM power reduces by no less than 20% for each benchmark and the average savings are about 27%. It turns out, however, there are overlapping factors contributing to these power savings. The default configuration interleaves physical addresses across memory hardware units, which typically consumes more power than the alternative of non-interleaved addresses employed by our custom kernel configuration. Additionally, the custom kernel's tray based allocation may help by itself if the application's overall footprint is small enough. In order to isolate the effect of color-based allocation and recycling, we compare the average DRAM power consumed with and without color guidance with our custom kernel configuration (Figure 5(b)). We find that the database benchmarks are able to reduce power consumption based on color guidance, while the benchmarks with smaller memory footprints do not improve. This is because, without local allocation and recycling enabled, the database benchmarks scatter memory accesses across many power-manageable units, while the memory footprints of *fop* and *luindex* are small enough to require only one or two power-manageable units. Finally, it also appears that the potential reduction in memory bandwidth from localizing allocation and recycling to a single DIMM has very little impact on the performance of these benchmarks. We find that there is virtually no difference in performance between any of the configurations, including when compared to the default Linux kernel.

8.3 Exploiting Generational Garbage Collection

While it is useful to apply coarse-grained coloring over an application's entire virtual address space, let us use this section to describe the potential from using finer-grained control over portions of an application's address space, by evaluating in reality a proposal that has been previously explored through simulation.

The technique we evaluate in this section was first proposed and simulated by Velasco et. al. [27] as a way to reduce DRAM power consumption in systems which employ generational garbage collection. Section 6.1 contained a description of the HotSpot garbage collector. The optimization [27] exploits the observation that during young generation collection, only objects within the young generation are accessed. Thus, if objects in the tenured generation reside on an isolated set of power-manageable units (i.e. no other type of memory resides on them), then these units will power down during young generation collection. It is important to note that this optimization also relies on the assumption that the system "stops the world" (i.e. does not allow application threads to run) during garbage collection to ensure that application threads do not access the tenured generation objects during young generation collection.

In order to implement this optimization in our framework, we arrange to house the tenured generation in a different set of power managed memory units than that allocated to the remainder, by modifying the HotSpot VM to color the tenured generation. When the operating system attempts to allocate resources for a virtual address colored as the tenured generation, it only searches for free memory from a subset of trays that it has earmarked for that color.

We ran the *derby* benchmark from the SPECjvm2008 benchmark suite for evaluation. *derby* is a database benchmark which spends a fair portion of its runtime doing garbage collection. We

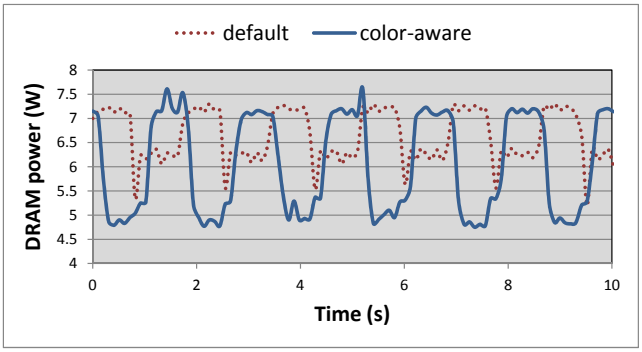


Figure 6. Raw power governor samples with and without "tenured generation optimization" applied

use the same machine and memory scrambler configuration that we use in the previous subsections and we measure performance and DRAM power consumption as described in Section 5.

Figure 6 plots the raw samples collected by the power governor tool during a portion of an experimental run with and without the optimization applied. Each dip corresponds to a separate invocation of the garbage collector. As we can see, the optimized configuration consumes slightly less power during garbage collection. Overall, the optimization reduces DRAM power by almost 9% and does not affect performance on our custom kernel. Thus, isolating the tenured generation memory in HotSpot on its own set of power-manageable units has a small but measurable impact on DRAM power consumption.

9. Future Work

Our current implementation of trays uses memory topology information that we provide to the operating system during its boot-up. This is a temporary measure; an immediate next task is to build upon MPST information that would be available in a Linux kernel that implements ACPI 5.0. Simultaneously we will implement color awareness in selected open source database, web server, and J2EE software packages, so that we can exercise complex, multi-tier workloads at the realistic scale of server systems with memory outlays reaching into hundreds of gigabytes. In these systems, memory power can reach nearly half of the total machine power draw, and therefore they provide an opportunity to explore dramatic power and energy savings from application-engaged containment of memory activities.

The experiments we reported in this paper were on a small scale, in a system with just 16 or 32 gigabytes of memory as our first phase intent was to demonstrate the concept of application influenced virtualization of memory. In the next phase of this work, in addition to exploring the saving of power on a non-trivial scale of a terabyte data or web application services, we plan to explore memory management algorithms that permit applications to maximize performance by biasing placement of high value data so that pages in which performance critical data resides are distributed widely across memory channels. Another element of optimization we plan to explore is application guided read-ahead and/or fault-ahead for those virtual ranges of applications in which there is reason to expect sequential access patterns. We also plan to implement a greater variety of page recycling policies in order to take advantage of color hints from applications; for instance, we would like to create the possibility that trays implement such algorithm options as (a) minimum residency time– where a page is not reclaimed unless it has been mapped for a certain threshold duration, (b) capacity allocation – where, physical pages allocated to different color groups are recycled on an accelerated basis as needed in order to keep the time-averaged allocation in line with application guided capacities, (c) reserve capacities – in which a certain minimum number of pages are set aside in order to provide a reserve buffer capacity for certain critical usages, etc. We envision that as we bring our studies to large scale software such as a complex database, we will inevitably find new usage cases in which applications can guide the operating system with greater nuance about how certain pages should be treated differently from others. Finally, one avenue of considerable new work that remains is the development of tools for instrumentation, analysis, and control, so that we can facilitate the generation and application of memory usage guidance between application software and operating system.

10. Conclusion

This paper presents a framework for application-guided memory management. We create abstractions called colors and trays to enable three-way collaboration between the application tasks, operating system software, and hardware. Using a re-architected Linux kernel, and detailed measurements of performance, power, and memory usage, we demonstrate several use cases for our framework, including emulation of NUMA APIs, enabling memory priority for applications, and power-efficiency for important applications. Hence, by providing an empirical evaluation that demonstrates the value of application-guided memory management, we have shown that our framework is a critical first step in meeting the need for a fine-grained, power-aware, flexible provisioning of memory.

Acknowledgments

We thank the reviewers, and especially Harvey Tuch, whose thoughtful comments and suggestions significantly improved this paper. We also thank Prasad Kulkarni at the University of Kansas who provided valuable feedback during the review process.

References

[1] V. Anagnostopoulou, M. Dimitrov, and K. A. Doshi. Sla-guided energy savings for enterprise servers. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120–121, 2012.

[2] G. Banga, P. Druschel, and J. 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] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. 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] M. Bi, R. Duan, and C. Gniady. Delay-Hiding energy management mechanisms for DRAM. In *International Symposium on High Performance Computer Architecture*, pages 1–10, 2010.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[6] A. D. Brown and T. 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.

[7] J. Corbet. Memory Compaction, Jan. 2010. URL `http://lwn.net/Articles/368869/`.

[8] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Scheduler-based dram energy management. In *Design Automation Conference*, pages 697–702, 2002.

[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, Dec. 1995.

[10] X. Fan, C. Ellis, and A. Lebeck. Modeling of dram power control policies using deterministic and stochastic petri nets. In *Power-Aware Computer Systems*, pages 37–41, 2003.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 57–76, 2007.

[12] A. Gonzalez. Android Linux Kernel Additions, 2010. URL `http://www.lindusembedded.com/blog/2010/12/07/android-linux-kernel-additions/`.

[13] http://lwn.net/Articles/445045/. Ankita Garg: Linux VM Infrastructure to support Memory Power Management, 2011.

[14] http://www.acpi.info/spec.htm. Advanced Configuration and Power Interface Specification, 2011.

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

[16] http://www.oracle.com/technetwork/java/whitepaper 135217.html. Oracle HotSpot JVM, 2012.

[17] http://www.spec.org/jvm2008/. SPECjvm2008, 2008.

[18] H. Huang, P. Pillai, and K. Shin. Design and Implementation of Power-aware Virtual Memory. In *USENIX Annual Technical Conference*, 2003.

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

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

[21] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *ACM SIGOPS Operating Systems Review*, 34(5):105–116, 2000.

[22] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36 (12):39–48, Dec. 2003.

[23] C.-H. Lin, C.-L. Yang, and K.-J. King. Ppt: joint performance/power/thermal management of dram memory for multi-core systems. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 93–98, 2009.

[24] L. Lu, P. Varman, and K. Doshi. Decomposing workload bursts for efficient storage resource management. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):860 –873, may 2011.

[25] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. In *Ottawa Linux Symposium*, pages 191–200, 2009.

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

[27] J. M. Velasco, D. Atienza, and K. Olcoz. Memory power optimization of java-based embedded systems exploiting garbage collection information. *Journal of Systems Architecture - Embedded Systems Design*, 58(2):61–72, 2012.

[28] H. Wang, K. Doshi, and P. Varman. Nested qos: Adaptive burst decomposition for slo guarantees in virtualized servers. *Intel Technology Journal*, June, 16:2 2012.

[29] 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*, ASPLOS XI, pages 177–188. ACM, 2004.