Cross-Layer Memory Management to Improve DRAM Energy Efficiency

MATTHEW BENJAMIN OLSON, JOSEPH T. TEAGUE, DIVYANI RAO,
and MICHAEL R. JANTZ, University of Tennessee
KSHITIJ A. DOSHI, Intel Corporation
PRASAD A. KULKARNI, University of Kansas

Controlling the distribution and usage of memory power is often difficult, because these effects typically depend on 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. This work describes 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 this framework by employing it to control memory power consumption more effectively. First, we design a custom memory-intensive workload to show the potential of this approach to reduce DRAM energy consumption. Next, we develop sampling and profiling-based analyses and modify the code generator in the HotSpot VM to understand object usage patterns and *automatically* 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 modules according to user-defined provisioning goals. The evaluation shows that our Java VM-based framework achieves our goal of significant DRAM energy savings across a variety of workloads, without any source code modifications or recompilations.

CCS Concepts: • **Software and its engineering → Virtual machines**; **Virtual memory**; **Allocation/deallocation strategies**; *Software performance*;

Extension of a conference paper: This work extends our earlier conference submission, titled *Cross-Layer Memory Management for Managed Language Applications*, published in Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) (Jantz et al. 2015). New contributions include the following: (a) we update the original MemBench benchmark and experimental framework to investigate the potential of our approach on NUMA-based platforms, (b) we extend our online profiling approach to enable higher-frequency sampling and to make it more suitable for multi-threaded applications, (c) we design and build a sampling-based tool to monitor the system's bandwidth and use this tool to regulate the number of memory devices available for hot application data and automatically eliminate performance losses, without sacrificing energy improvements, and (d) we evaluate our approach with additional benchmarks and several new experimental configurations that explore the impact of varying parameters for each partitioning strategy and different memory resource constraints. Additionally, we port our complete experimental framework and conduct our experiments on the latest HotSpot Java VM (migrate from v. 1.6 to v. 1.9) and memory hardware (migrate from DDR3 to DDR4-based system).

This research was supported in part by the National Science Foundation under CCF-1619140, CCF-1617954, and CNS-1464288, as well as a grant from the Software and Services Group (SSG) at Intel.

Authors' addresses: M. B. Olson, J. T. Teague, D. Rao, and M. R. Jantz, Min H Kao Building, Room 605, 1520 Middle Dr., University of Tennessee, Knoxville, TN 37996; emails: {molson5, jteague6, drao, mrjantz}@utk.edu; K. A. Doshi, Intel Corporation, 5000 W. Chandler Blvd., Chandler, AZ 85226; email: kshitij.a.doshi@intel.com; P. A. Kulkarni, Nichols Hall, Room 137, 2335 Irving Hill Rd., University of Kansas, Lawrence, KS 66045; email: kulkarni@ittc.ku.edu.

https://doi.org/10.1145/3196886

Additional Key Words and Phrases: Memory management, DRAM, power, energy, optimization

ACM Reference format:
Matthew Benjamin Olson, Joseph T. Teague, Divyani Rao, Michael R. Jantz, Kshitij A. Doshi, and Prasad A. Kulkarni. 2018. Cross-Layer Memory Management to Improve DRAM Energy Efficiency. *ACM Trans. Archit. Code Optim.* 15, 2, Article 20 (April 2018), 27 pages.
https://doi.org/10.1145/3196886

1 INTRODUCTION

Rising demands for high-volume, high-velocity data analytics have led to a greater reliance on in-memory data processing to achieve performance goals. These trends are driving bandwidth and capacity requirements to new heights and have led to the adoption of memory systems with larger power, bandwidth, and capacity requirements. However, as memory systems continue to scale, DRAM energy consumption has become a major concern. Previous studies have found that memory devices account for about 30–50% of overall power consumption in a typical server node (Lefurgy et al. 2003; Hoelzle and Barroso 2009; Ware et al. 2010; Vetter and Mittal 2015). Furthermore, since memory power is directly related to the number and size of DRAM modules, this fraction is expected to remain significant with continued scaling.

Unfortunately, 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 on 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 (a) 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 (b) the *OS-layer* incorporates this guidance when deciding which physical page to use to back a particular virtual page, according to user-specified power/performance goals.

Some OS-level system calls, such as *madvise* (Gomez 2001) and *mbind* (Kleen 2004a), and frameworks have been developed to facilitate communication across multiple layers of the vertical execution stack during memory management (Jantz et al. 2013; Cantalupo et al. 2015; Dulloor et al. 2016; Kleen 2004b). In particular, the framework proposed by Jantz et al. (2013) 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 determine the set of memory usage guidance to provide to the OS, which may be infeasible for many complex applications and workloads. It is often also necessary to update and re-apply usage guidance as the program executes to reflect changes in application behavior. Furthermore, where relevant, 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 (VM), divides the application's *heap* into separate regions for objects with different (expected) usage patterns. At application runtime, our custom VM automatically partitions and allocates heap data into separate regions using an integrated object partitioning strategy. Our framework adopts a recent Linux OS extension and its associated API (Jantz

et al. 2013) to transfer this information to the OS, where it is used to guide physical memory management. This work extends HotSpot with two new object partitioning strategies: an *offline* profiling-based approach that classifies program allocation sites statically, as well as an *online* sampling-based approach that segregates data objects at runtime. For this journal extension, we have also integrated our approach with hardware-based sampling to detect shifting bandwidth requirements and use this information to adapt data management policies at runtime.

In this article, 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 custom benchmarks to conduct experiments that reveal interesting aspects of the power-performance tradeoff in the memory subsystem and the potential for our approach to achieve DRAM energy savings on uniform and non-uniform memory access (NUMA) architectures. We also present detailed evaluation of the proposed partitioning schemes and cross-layer policies using a standard set of Java benchmarks. For these experiments, we isolate the impact of different parameters and strategies from resource sharing and/or NUMA effects by executing each workload one at a time in a uniform memory access environment.

This work makes the following important contributions: (1) We develop, implement, and evaluate a framework to automatically partition application data objects into distinct sets based on their expected usage patterns; (2) we analyze the potential of memory power savings and the interplay between program performance, memory power, and bandwidth requirements by designing and employing a custom benchmark program; (3) we build profiling and sampling routines into the HotSpot VM to automatically predict and react to memory usage behavior; and (4) we provide detailed experimental results, including analysis of DRAM energy and performance, to evaluate our cross-layer policies using a standard set of Java applications with a range of power, bandwidth, and capacity requirements.

2 RELATED WORK

A few projects have used architectural- and/or OS-based profiling and analysis to guide low-level memory management. Some researchers have proposed custom hardware counters (Meswani et al. 2015), or lightweight sampling techniques (Agarwal and Wenisch 2017), to identify and keep hot pages in a tier of high-bandwidth memory devices. The Carrefour system (Dashti et al. 2013) employed similar strategies to reduce access congestion in NUMA systems. These approaches are similar to ours in that they re-locate data in the physical address space to increase efficiency. However, they rely on coarse-grained profiling of physical memory pages and may require non-standard hardware. Our approach enables the runtime to consider more fine-grained profiling of application data objects and is entirely software based.

Some other approaches also integrate guidance from the application runtime with management of physical resources. Agarwal et al. (2015) and Dulloor et al. (2016) used a profiling tool to assign pre-tagged data structures to the appropriate tier in a heterogeneous memory system. Guo et al. (2015) integrated reactive profiling with memory allocation to reduce pollution in a shared cache. While these studies demonstrate some of the benefits of application feedback during memory management, they have different goals than our approach and do not provide any mechanism for migrating program data as usage patterns change.

A number of other works have proposed integrating information at the application-level with the OS and hardware to aid resource management (Engler et al. 1995; Belay et al. 2012; Banga et al. 1999; Liu et al. 2011). Prior system calls, such as *madvise* (Gomez 2001) and *mbind* (Kleen 2004a), and various libraries and APIs (Kleen 2004b; Gonzalez 2010; Magenheimer et al. 2009) have allowed applications to provide hints to the memory management system. Similar frameworks were recently developed to allow applications to directly control the placement of their data on

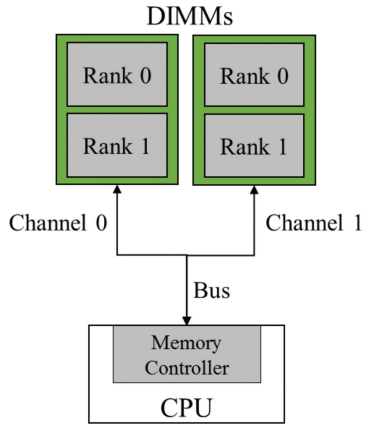

Fig. 1. Organization of devices in a DDR-style memory system.

heterogeneous memory systems through hints in their source code (Cantalupo et al. 2015; Dulloor et al. 2016) . In contrast, our approach obtains the needed guidance directly from lightweight execution profiling and eschews the need to recompile.

Prior work has also explored techniques and policies for making the virtual memory system power aware (Lebeck et al. 2000; Huang et al. 2003; Garg 2011). Others have proposed keeping track of various metrics, such as page miss ratios (Zhou et al. 2004) or memory module accesses (Delaluz et al. 2002), to guide allocation and reduce DRAM energy. 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 inefficiencies resulting from the OS and applications working at cross purposes.

3 MEMORY ORGANIZATION AND POWER CONSUMPTION

3.1 Organization of the Memory Subsystem

Although our cross-layer framework has the potential to benefit numerous memory technologies, we focus our current study on today's pervasive JEDEC-style DDR memory systems (JEDEC 2009). Figure 1 shows the organization of devices in a typical DDR memory system. In standard DDR-style systems, 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 as well as a set of *ranks*, which constitute the actual memory storage. 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 rate of memory accesses to each device, while background power depends solely on the operating frequency and

[1]For a full accounting of the factors that contribute to DDR memory power, see Micron Technology Inc. [2001].

the current 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. For general purpose DDR devices, ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level (Bhati et al. 2016). As with other power-management techniques, placing a portion of the device into a power-down state incurs a wakeup penalty the next time the device is accessed. Current DDR3/4 technology supports multiple power-down states, each of which poses a different tradeoff between power savings and wakeup latency. Table 3 in David et al. (2011) provides a summary of power requirements and wakeup latencies for the various power-down states for a typical DDR device. Thus, controlling memory power 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, while low-power configurations attempt to minimize the number of active memory devices.

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. Our approach uses a custom Java VM (based off the open source HotSpot VM) that divides its application heap into separate regions for objects with different expected usage patterns. We integrate our modified VM with the OS-based framework proposed by Jantz et al. (2013), which allows application-level software to communicate usage intents and provisioning goals to the OS-level memory manager. 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 host which physical pages and to use this detail in tailoring memory allocation to usage patterns. 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.

4.2 Organizing Memory Objects in HotSpot

This work complements the OS framework by integrating application guidance with a custom Java VM that *automatically* partitions program objects into separately colored regions. Each region 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 to be effective.

For our implementation, we use the standard HotSpot Java VM with the "parallel-scavenge" garbage collector (PS-GC), which is typical for server-class applications (Sun-Microsystems 2006).

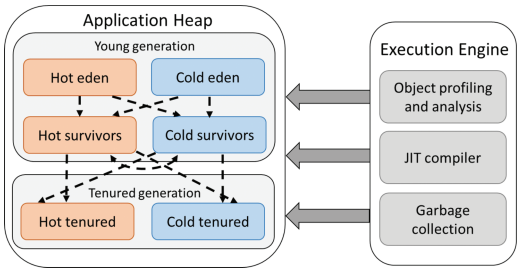

Fig. 2. Colored spaces in our VM framework. Dotted lines indicate possible paths for objects to move from one colored space to another.

PS-GC is a generational, "stop-the-world" collector (i.e., application threads do not run during GC), that is capable of spawning multiple concurrent scavenging threads during GC. Our implementation, illustrated in Figure 2, subdivides each heap space into two equal-sized regions[2]: one colored orange for hot (frequently accessed) objects and the other colored blue for cold objects.

We implement two distinct strategies for ensuring objects reside in the correct space: (1) an *offline* strategy that uses profiling information from a previous run to guide object coloring decisions in later runs and (2) an *online* strategy that collects samples of program activity to predict memory usage in the same run. The VM consults the available usage information at each object allocation site to determine which color to assign the new object. During GC, colors are used to copy surviving objects to the appropriate space but do not otherwise impact how or which objects are collected. For instance, if an object allocation triggers a GC cycle (e.g., because its color's eden space is full), then the VM will collect the entire young generation (for minor GC) or entire heap (for major GC) regardless of the color of the new object.

5 EXPERIMENTAL FRAMEWORK

Platform. The experiments in this article use a Microway NumberSmasher Xeon Server machine with two sockets, each with their own Intel E5-2620v3 (Haswell-EP) processor. Each socket has six 2.4GHz cores with hyperthreading enabled (for a total of 12 hardware threads) and four 16GB DIMMs of Samsung DDR4 SDRAM with a clock speed of 2133MHz (product #: M393A2G40DB0-CPB). Each DIMM is connected to its own channel and is composed of two 8GB ranks. All experiments execute on an otherwise idle machine and, aside from Section 6.2, use only one socket and its local memory. We install 64-bit CentOS 6.6 and select Linux kernel version 2.6.32-431 (released in 2013) as our default operating system. Oracle's HotSpot Java VM (build 1.9.0-ea-b76) (Paleczny et al. 2001) provides the base for our VM implementation.

Performance, power, and bandwidth measurement. Performance for each benchmark is reported in terms of throughput or execution time, depending on the benchmark set. To estimate memory power and bandwidth, we employ Intel's Performance Counter Monitor (PCM) tool (Willhalm et al. 2012). 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 average read/write bandwidth on each memory channel (in MB/s), cache hit ratios, and an estimate (in Joules) of energy consumed by the CPU (cores and caches) and memory system. We use PCM to collect these statistics once per second

[2]The default PS-GC uses heuristics to dynamically tune the size of each heap region for throughput or pause time goals. To simplify our implementation, all of our experiments use static heap sizes, which are computed for each benchmark using a prior run with the default configuration.

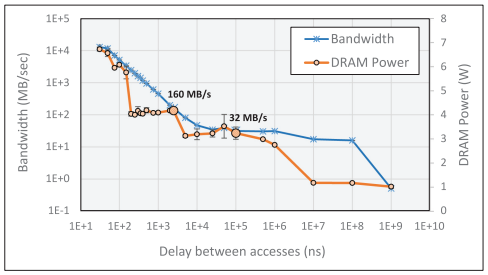

Fig. 3. Power (W) of one 16GB DDR4 DIMM with different amounts of time between accesses.

during each program run and aggregate the collected data in post-processing. All experimental results report an average of five program runs. Other details regarding the benchmarks and evaluation methodology for a particular set of experiments are presented later in the relevant sections.

To estimate the degree of variability in our performance and energy results, we compute 95% confidence intervals for the difference between the means (Georges et al. 2007) and display these intervals as error bars on the appropriate graphs. In several cases, we omit these intervals for every other bar in the graph to reduce clutter and improve presentation. Overall, performance variability is quite low (typically <2%), and in some cases, such as in Figures 4(a) and 5(a), confidence intervals are not visible in print. However, some benchmarks and configurations, such as the multi-threaded SciMark benchmarks with the online partitioning strategies, exhibit larger variations from run to run. Our methodology for measuring CPU and DRAM energy relies on architectural counters, which are known to have additional sources of error, such as non-determinism and overcounting (Weaver et al. 2013). While these issues do increase the variability of our energy results, in general, we find that this approach is precise enough for clear comparisons between configurations.

6 POTENTIAL OF CROSS-LAYER MANAGEMENT TO REDUCE DRAM ENERGY

To better understand the relationship between application data usage and DRAM power, we created a custom benchmark, called MemBench. On initialization, MemBench creates a set of objects, each with a single integer field, and stores them in an in-memory array. The array is divided into 12 roughly equal parts (one for each hardware thread on one server node) that are each passed to their own software thread. The software threads then (simultaneously) iterate over their own portion of the array and write a new value to the integer stored with each object they pass. Between integer stores, MemBench (optionally) spins each thread in a loop to vary the rate of access to data in the object array. Spinning for a larger number of iterations increases the delay between memory accesses and reduces the average bandwidth requirements for the workload.

For our initial experiments, we configure MemBench to allocate approximately 2.5GB of data objects and to iterate over the object array for 100s.[3] The entire object array is assigned to a single DRAM module using the cross-layer framework. We also vary the number of iterations each thread spins between memory accesses, and use the system timer (i.e., *System.nanoTime()*) to select numbers that induce delays at regular intervals. All reported measurements are collected only during the portion of the run where software threads iterate over the object array.

Figure 3 plots the average bandwidth to the DRAM module (in GB/s) on the left y-axis (log scale) and the average DRAM power (in W) on the right y-axis with different delays between

[3]All experiments in this section allocate 2.5GB of objects to ensure out-of-cache accesses. We did not vary the amount of data allocated, because capacity utilization is not a significant factor of DRAM power.

object accesses. We find that increasing the delay between accesses significantly reduces DRAM power. However, the power consumption of the DRAM module is not directly proportional to its bandwidth. The device transitions to a lower power state if the time between accesses exceeds some fixed threshold value, which results in a power profile resembling a step function. Additionally, even relatively low-bandwidth configurations consume significantly more power than the idle configuration. For instance, bandwidths of 160MB/s ($delay = 2.5\mu$s) and 32MB/s ($delay = 0.1$ms), require more than 4× and 3× DRAM power than idle, respectively. Thus, achieving power efficiency requires a system that can identify and isolate objects that will only be accessed very infrequently. In the most extreme case, a completely idle DDR4 module consumes about 15% of the power of a fully active module.

6.1 Extending MemBench with Hot and Cold Objects

Our next experiments evaluate the potential of using object placement to control DRAM power using a simple workload with pre-defined sets of hot and cold objects. We extended MemBench to differentiate hot and cold data *statically* by declaring objects with distinctive program types in its source code. The extended MemBench creates and stores two types of objects in a single array: *HotObject* and *ColdObject*. Each object contains a single integer value representing the memory associated with the object. As the benchmark threads iterate over their portion of the object array, they pass over the cold objects and write new integer values to the hot objects.

We constructed three configurations for these experiments: (1) *default* employs the unmodified Linux kernel with the default HotSpot VM, (2) *tray-based kernel* runs MemBench with the unmodified HotSpot VM on the custom kernel framework described in Section 4.1, and (3) *hot/cold organize* uses the custom kernel with the custom Java VM with separate spaces for hot and cold objects. We configure MemBench to allocate roughly 2GB of cold program objects and 200MB of hot program objects. Objects with different types are allocated in random order. Hence, in the default and tray-based kernel configurations, HotObjects and ColdObjects are stored intermittently throughout the application heap and ultimately, across all four memory modules. In hot/cold organize, the modified VM allocates hot and cold objects to separate heap regions, and the OS co-locates data in the hot heap region onto a single DRAM module. Hot objects are shuffled and accessed in random order (using the same random seed for each experiment) to ensure comparable cache utilization across all configurations. Similarly to the original MemBench experiments, we iterate the accessor threads for 100 seconds and measure performance as the number of hot objects processed. We test each configuration with a range of delay factors and report all results as the average of five runs with each configuration-delay pair.

The lines in Figure 4(a) (plotted on the left y-axis) compare the performance (throughput) of the tray-based kernel and hot/cold organize configurations to the default performance with the same delay. (Higher values indicate better performance.) Thus, we can see that tray-based kernel performs about the same as the default regardless of the delay factor. In contrast, 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. Not surprisingly, this observed difference in throughput is directly related to bandwidth on the memory controller. The bars 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 exhibit much higher bandwidth than hot/cold organize when the delay factor is low. These configurations both distribute hot objects across the memory devices, allowing the system to exploit rank and channel parallelism to achieve higher bandwidths. Alternatively, hot/cold organize co-locates all the hot objects onto a single DIMM on a single channel and cannot attain the same bandwidth as the other configurations when the delay is low. Increasing the delay between memory accesses

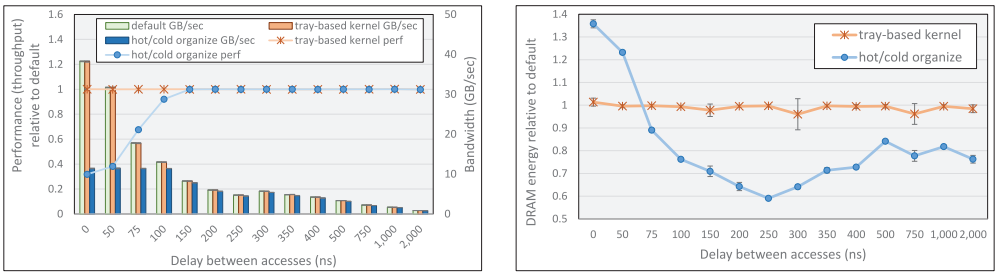

Fig. 4. MemBench evaluation. (a) Plots performance relative to the default configuration as lines on the left *y*-axis (higher is better) and average bandwidth as bars on the right *y*-axis. (b) DRAM energy relative to default (lower is better).

reduces the bandwidth requirements for the workload, which enables hot/cold organize to achieve performance similar to the other configurations.

While co-locating the hot objects onto a single DIMM restricts bandwidth, it also consumes much less power, on average, than distributing accesses across all the memory devices. 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 DRAM energy consumed with the tray-based kernel and hot/cold organize configurations compared to default with different delay factors. Thus, while hot/cold organize may hurt both performance and energy efficiency for very high bandwidth workloads (with delay <75ns), this approach can significantly reduce DRAM energy with little or no impact on performance for lower-bandwidth workloads (with delay ≥150ns). At a maximum, the hot/cold organize configuration achieves 41% DRAM energy savings with delay = 250ns. Increasing the delay beyond 250ns allows the default configuration to transition memory hardware to lower power states more frequently, which diminishes the relative energy savings of this technique.

It is important to consider how these DRAM energy savings impact the total energy consumption of the system. We find that memory power contributes between 10.6% (with delay = 2,000ns) and 25.5% (with delay = 0ns) of the combined CPU+DRAM power on our single node platform with 64GB of DDR4 memory. Thus, this technique reduces the CPU+DRAM power of MemBench by about 2% to 8% with delays ≥150ns and achieves a best case reduction of 8.8% with delay = 250ns. Older systems that use DDR3 memory, which operates at a higher voltage than DDR4 will achieve more substantial overall energy savings using this technique. For instance, a similar experiment in the previous version of this work (Jantz et al. 2015) found that this approach reduces CPU+DRAM energy consumption by more than 15% on a DDR3-based machine.

6.2 Implications of Cross-Layer Management for NUMA-Based Platforms

The experimental platform we use in this work contains less capacity than many enterprise and scientific computing systems. Modern and future systems with larger memory capacities will necessarily consume more power in the memory subsystem, and so, increasing memory efficiency in these systems will lead to greater reductions in overall energy consumption. One potential complication of realizing these energy savings is that such systems often employ a non-uniform memory access (NUMA) architecture to accommodate the additional capacity. To better understand the impact and implications of our approach with larger capacity, NUMA-based systems, we designed

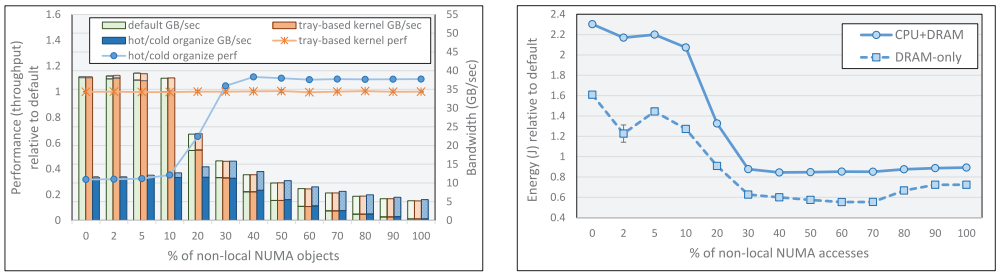

Fig. 5. NUMABench evaluation. (a) Performance relative to the default configuration as lines on the left *y*-axis and average bandwidth as stacked bars on the right *y*-axis. The lower (darker) bar is NUMA-local bandwidth, and the upper (lighter) bar is NUMA-remote bandwidth. (b) DRAM energy and CPU+DRAM energy relative to default.

and conducted another series of experiments that use all 128GB of memory on both nodes of our experimental platform.

These experiments use another custom benchmark called *NUMABench*. NUMABench is identical to MemBench with one important difference: rather than only one type of hot program objects, it includes two distinct types for NUMA-local and NUMA-remote hot objects. Arguments to NUMABench control the number of threads created on each NUMA node, the amount of hot and cold objects, as well as the proportion of NUMA-remote hot objects. To complete the implementation, we modified our custom Java VM to divide each colored region into separate spaces for each NUMA node. Specifically, on our platform with two NUMA nodes, each heap region contains four spaces each with their own distinct color: a hot and cold space for node 0 and a hot and cold space for node 1.

The experiments with NUMABench use parameters and configurations similar to those used in the previous subsection for MemBench. The *default* and *tray-based kernel* configurations for NUMABench use the custom Java VM to assign program threads and data to the appropriate node but are otherwise identical to the earlier experiments. *hot/cold organize* is also similar, and uses a single DIMM on each node to fill demands for hot program data. Each NUMABench experiment spawns a number of software threads to access the hot program data and allocates the same amount and proportion of hot and cold program data as MemBench. All of the threads are bound to the same NUMA node and continuously access the (local or remote) hot program objects with some spin-delay between each access for 100 seconds. For presentation purposes, we only show detailed results for experiments that use 12 software threads on one compute node with no spin-delay between memory accesses. Later in this subsection, we summarize and discuss results for configurations that use threads on both compute nodes and vary the delay between memory accesses.

Figure 5(a) presents the performance (throughput) and bandwidth of NUMABench with the percentage of NUMA-remote hot data varied between 0% and 100%. The figure uses stacked bars to distinguish bandwidth on the local node (the lower, darker bar) from bandwidth on the remote node (the upper, lighter bar). As expected, NUMABench exhibits similar bandwidth as the earlier MemBench experiments when there are no hot objects on the remote node. Increasing the proportion of remote hot objects does increase system throughput up to a point–default and tray-based kernel perform best with 5% of hot objects on the remote node, while hot/cold organize is best with 30% remote hot objects—but these improvements quickly diminish as the percentage of traffic on the local node is reduced. Furthermore, assigning more hot objects to the remote node degrades

Table 1. NUMABench Bandwidth with No Remote Data

# Hot	Total GB/s	Hot GB/s
1	11.7	11.7
2	23.2	11.6
3	32.1	10.7
4	39.5	9.9

total system bandwidth (due to the heavy remote traffic) and ultimately reduces the performance difference between the configurations.

Surprisingly, we find that hot/cold organize actually performs about 10% *better* than default when 40% or more of the hot objects are assigned to the remote node. This result suggests that confining the remote data accesses to a single DRAM module reduces the amount of time between accesses to the module and prevents it from transitioning to a lower power (longer latency) state. For NUMABench, reducing latency to the remote data increases bandwidth and performance. Interestingly, this effect also occurs on the local NUMA node even when there is no traffic on the remote memory devices. For example, consider Table 1, which shows the total bandwidth and the average bandwidth of only the hot DRAM modules for four runs of NUMABench that place all hot objects on the local node and use a different number of hot modules for each run. Thus, although using all four DRAM modules for the hot objects produces the highest total bandwidth (due to increased channel parallelism), the average bandwidth for each hot DRAM module is 1.5%, 10%, and 19% higher when only one hot module is used as compared to configurations with two, three, and four hot modules, respectively.

Figure 5(b) plots lines showing the average CPU+DRAM energy and DRAM-only energy of NUMABench with hot/cold organize with different percentages of remote hot objects relative to the default configuration. Thus, in the best-case with 40% of hot objects on the remote node, hot/cold organize reduces CPU+DRAM energy by over 15% on our platform. Comparing to the results in Figure 4(b), we see that hot/cold organize yields even greater energy savings with NUMABench than with MemBench. These increased energy savings are not only due to better performance but also because memory power is a larger component of total system power when DRAM on both nodes is in use. In the best-case configurations for hot/cold organize, memory is only 15.2% of CPU+DRAM power with MemBench (delay = 250ns), while it is 22.9% of CPU+DRAM power with NUMABench (40% remote accesses).

In addition to the experiments described above, which use only 12 software threads bound to a single compute node, we ran several other sets of experiments with NUMABench using 24 software threads (one per hardware thread) across both compute nodes. As expected, increasing the number of software threads raises the total power and bandwidth requirements for the application, especially when most memory accesses are local and there is little or no delay between accesses. In 24-thread experiments with no remote data accesses, hot/cold organize exhibits similar performance and energy effects as in the MemBench experiments with all threads and data bound to a single NUMA node. Specifically, performance of hot/cold organize with no remote accesses is 68.1% worse than default with no delay between accesses and no worse than default with delays $\geq$150ns. In the best case with delay = 400ns, hot/cold organize with no remote accesses consumes 43.4% less DRAM energy and 6.9% less CPU+DRAM energy than the default configuration.

We also find that placing some portion of the data accesses on the remote nodes with the 24-thread experiments causes similar performance/energy effects as we observed with the single-node NUMABench experiments *when there are no delays between memory accesses*. However, due to the

increased bandwidth on both nodes, hot/cold organize does not achieve the same performance as default until at least 70% to 80% of accesses are on the remote node. Additionally, increasing the delay between accesses diminishes the impact that NUMA effects have on this workload. For instance, with delay = 250ns, hot/cold organize consistently yields similar performance and better energy efficiency (29% to 44% less DRAM energy, 5% to 9% less CPU+DRAM energy) than the default configuration, regardless of the portion of NUMA-remote data accesses.

While these results demonstrate that there is potential for cross-layer management to produce substantial energy savings on NUMA platforms, a number of challenges still remain for real-world applications. In addition to knowing data access rates, achieving efficiency in NUMA systems requires knowledge of which compute resources (i.e., software or hardware threads) drive traffic to which data objects or addresses. There are some existing NUMA management strategies that rely on heuristics and/or profiling techniques to predict this information and use it to improve performance. For instance, since version 6u2, HotSpot includes an option for NUMA-aware data allocation with the PS-GC (Oracle 2017). Some recent Linux distributions also include OS-level optimizations to improve access locality in NUMA systems (Corbet 2012; van Riel 2014). There is some potential to combine cross-layer management with these existing strategies to achieve the benefits of multiple approaches. For example, one possible configuration might be to use a NUMA management strategy to assign data to each node and to use cross-layer management to assign objects to the devices within each node to improve efficiency. Since the primary focus of this work is to understand the effectiveness of the proposed approaches without NUMA effects, the remainder of this study uses only a single node of our experimental platform.

7 AUTOMATED COLLECTION OF MEMORY USAGE GUIDANCE

7.1 Object Partitioning Strategies

7.1.1 Offline Profiling for Hot and Cold Allocation Sites. Our offline partitioning strategy profiles memory usage activity related to *program allocation sites*. For this approach, we instrument the HotSpot bytecode interpreter to construct a hash table relating objects and their usage activity to their allocation sites in the interpreted code. The hash table is indexed by program allocation sites (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 site. The modified interpreter intercepts each object allocation request (`_new` and its variants) and each object reference (`getfield`, `putfield`, etc.) to compute sizes and access counts for each allocation site.

With the profiling collected, the next step is to derive a placement decision, across allocation sites, that distinguishes the hot and cold objects. 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 express this problem as an instance of the classical 0/1 knapsack optimization problem. In this formulation, each allocation site is an item with a value equal to the total size of its objects, and a weight equal to the sum of its access counts. The optimization problem then is to select sites such that the combined size (value) is maximized without the combined access counts (weight) exceeding the some pre-defined capacity limit. 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 (Martello and Toth 1990). To compare colorings across different workloads, we select knapsack capacities as a fraction of the total number of accesses, D. Thus, if this fraction is 10%, the knapsack approach ensures that the aggregate access count for the sites assigned to the cold set is just below $D/10$.

At the start of any guided/measured program run, the VM loads the partitioning information from a file into memory. Each method object maintains a list of its associated allocation sites with their hot/cold classification. Since each node in this list requires only a few (16, currently) bytes of space, and since each method has only a few allocation sites (see Table 3), the space overhead is small. At each object allocation, the VM checks the appropriate list to determine whether the allocation site has a known color. If so, then the new object is allocated to this color's eden space, and otherwise, to some default color's eden space. For this work, we opt to assign unknown (uncolored) allocation sites 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.

7.1.2 Call Stack Sampling to Distinguish Hot Methods and Classes. Our online approach is inspired by a technique proposed by Huang et al. (2004). Huang et al.'s strategy uses bytecode analysis during JIT compilation to mark data classes and fields as hot if they are accessed by frequently executing program methods. In contrast to this earlier work, which used this information to reorder fields within each object and improve data locality, our approach aims to partition the application's data objects into hot and cold sets and does not alter the structure of individual objects.

Specifically, our approach works as follows. The VM maintains a `temperature` field for each method and class object that indicates whether the data associated with these constructs is currently "hot" or "cold." Initially, all methods and classes are considered cold. When a method is JIT compiled, the VM analyzes its bytecodes and records the list of classes (data types) that it might access. As the program executes, a separate profiling thread periodically examines the call stacks of each application thread and marks the methods at the top of each stack, as well as the classes that they access, as hot. If a method or class is not sampled for some pre-defined number of intervals, then the VM may also reset its `temperature` to cold. Then, whenever a method allocates a new object, the allocator examines the `temperature` of the method and of the new object's class. If either field indicates that the new object is likely to be hot, then the VM allocates it to the hot space and, otherwise, to the cold space. In contrast to the offline approach, which is completely static, the online approach also reassigns surviving objects' colors at each GC cycle according to the `temperature` of their classes.

The original implementation of the online partitioning strategy, presented in Jantz et al. (2015), used a timer-driven signal to pause the application threads and read their call stacks at regular intervals. While this "stop-the-world"-style approach allows the VM to parse the call stacks without error, it also degrades performance when the sampling rate is increased. For this extension, we have updated the implementation of the online strategy to enable high-frequency profiling without slowing down the application. During initialization, the VM pins the sampling thread to a single core and then ensures that all other VM and application threads use only the remaining compute cores. During execution, the sampling thread attempts to read each active call stack without stopping or interrupting the application threads.

While this design enables faster sampling rates, it may cause errors if a call stack changes as the profiling thread attempts to read it. To handle such cases, the VM employs a custom signal handler to catch and discard SIGSEGV signals raised by the profiling thread and uses internal checks to verify that the sampled stack frame corresponds to a valid program method.[4] We found that the proportion of successful call stack samples varies significantly depending on the benchmark. In the best case, with *tomcat*, less than 20% of attempts to read the application call stack result in error, while in the worst case, with *fft (11-thread)*, call stack sampling fails more than 99% of the

[4]Specifically, these checks (1) ensure the address of the sampled frame is within the appropriate heap region and (2) cast the sampled address to a Method object and verify that the virtual table pointer (`__vptr`) associated with this object matches that of the Method class.

Table 2. Number of Hot DIMMs for Each Bandwidth Range

# Hot	BW (GB/s)
1	<10.1
2	10.1–12.5
3	12.5–16.3
4	>16.3

time. Even with large error rates, this implementation enables the collection of a few thousand to over 1.5M valid samples per second (as shown in Table 3) with very little impact on performance in most cases.

7.2 Sampling Architectural Counters to Estimate Bandwidth Requirements

If an application requires high bandwidth, then co-locating its hot objects onto a small set of devices (and channels) may actually hurt performance and efficiency. To address this issue, we have integrated our framework with a custom tool that monitors the system's bandwidth using architectural counters. With this information, the framework can remap application data across different sets of DRAM devices depending on the current requirements.

The remapping policy relies on the construction of an offline model to determine the best configuration for applications with different bandwidth requirements. We can construct the model for a given machine using the extended MemBench workload from Section 6.1. For each possible device configuration (i.e., using 1, 2, 3, or 4 DIMMs for the hot objects on our platform), we ran MemBench with varying delays and recorded the average bandwidth during the run. We then mapped the observed bandwidth to the most energy efficient device configuration. Specifically, if MemBench with a particular delay performs worse with a limited number of devices, then the model assumes that additional devices are necessary to sustain the observed bandwidth. Otherwise, it recommends to assign data with similar or lower bandwidth requirements to the limited number of devices, since doing so will not hurt performance and will reduce energy consumption. Table 2 shows the number of hot devices for each bandwidth range as recommended by the MemBench model on our platform.

During subsequent program executions, the VM periodically samples hardware activity counters to compute an estimate of current bandwidth.[5] If the model indicates that the current configuration with the observed bandwidth is not optimal, then the VM will invoke the OS to unmap all of the pages associated with colored heap regions and then remap them to a different set of hot and cold devices depending on the model's recommendation before resuming the program. Since these remapping operations are expensive,[6] our implementation is conservative: It samples system bandwidth only once per second and will only remap to a new configuration if two consecutive samples fall within its corresponding bandwidth range.

[5]This mechanism is similar to the approach used by the Intel PCM tool described in Section 5. At each sampling interval, the VM reads the CAS_COUNT.RD and CAS_COUNT.WR events for each channel and uses these to compute an estimate of the total bandwidth on the platform (see Intel (2015)).

[6]We found that remapping overhead is 0.75 to 1.15 seconds/GB with a microbenchmark on our platform.

Table 3. Benchmark Statistics

Benchmark	Perf.	GB/s	DRAM W	OA (GB)	# Sites	S/M	# Hot	# Types	Samps/s
DaCapo									
avrora	4.999	0.042	3.786	0.344	3,476	2.82	1,108	892	151,803
fop	0.416	1.631	7.188	0.754	10,228	5.66	1,200	1,443	133,406
h2	8.613	0.420	5.218	4.721	3,361	2.59	932	814	118,324
jython	4.257	0.837	6.862	11.194	11,470	4.71	707	2,593	155,383
luindex	0.699	0.333	5.247	0.195	2,968	2.71	739	715	167,691
lusearch	0.996	8.565	8.628	35.765	2,628	2.73	836	642	521,303
pmd	3.445	0.958	5.884	4.376	3,516	2.45	1,311	1,156	239,909
sunflow	2.247	1.905	7.388	14.472	3,895	2.72	842	919	408,436
tomcat	2.187	1.879	5.518	4.067	8,510	2.61	1,049	2,212	551,227
tradebeans	13.492	0.418	5.235	16.314	16,148	2.39	494	4,978	1,517,788
tradesoap	6.441	2.083	6.461	35.250	16,266	2.37	492	5,212	1,095,354
xalan	0.749	4.068	7.878	6.381	3,587	2.84	1,185	939	360,641
SciMark									
fft-1T	2.586	8.118	8.432	3.015	6,588	3.05	551	1,528	4,140
lu-1T	4.427	14.087	8.848	3.495	6,593	3.05	550	1,527	4,052
sor-1T	5.254	3.231	7.380	0.074	6,583	3.05	544	1,528	3,193
sparse-1T	2.478	10.753	7.726	4.745	6,585	3.06	547	1,527	4,334
fft-11T	0.633	34.899	14.506	12.697	6,588	3.05	549	1,528	6,272
lu-11T	2.460	38.785	14.325	7.268	6,593	3.05	555	1,527	88,991
sor-11T	0.644	25.753	12.404	0.416	6,583	3.05	550	1,528	57,585
sparse-11T	0.851	35.853	13.374	14.156	6,585	3.06	549	1,527	97,998

Note: For each benchmark, the columns show performance (execution time (seconds) for DaCapo and throughput (ops/minute) for SciMark), bandwidth (GB/s), DRAM power (W), objects allocated (GB), number of allocation sites, allocation sites per method, number of methods compiled in steady state, number of unique program types, and valid call stack samples per second with continuous online profiling.

8 EVALUATION

8.1 Benchmarks and Experimental Configuration

We employ 12 of 14 benchmarks from the DaCapo benchmark suite (v. 9.12) (Blackburn et al. 2006) as well as four of the SciMark computational kernels from SPECjvm2008 (SPEC2008 2008) to evaluate our cross-layer framework.[7] Table 3 shows all of the selected benchmarks along with relevant performance, usage, and runtime statistics. Some of the experiments require the collection of information during a profile run with a small input size to guide coloring decisions in a run with a default or large input size. Thus, all of the benchmarks we selected for this study include at least two inputs, and we omit benchmarks from SPECjvm2008 with only one input size. The

[7]The two DaCapo benchmarks that we omit from this study, *batik* and *eclipse*, fail to run with the default configuration of HotSpot-9 due to incompatibilities that were introduced in OpenJDK 8 and have been observed and reported by others (github.com 2014a, 2014b).

SciMark kernels also include a parameter to duplicate their workload across multiple, concurrent application threads within a single VM instance. Thus, to test our framework with a wider range of usage requirements, we include two configurations of each SciMark kernel: a lower-bandwidth single-threaded version and a higher-bandwidth multi-threaded version. Although our platform includes 12 hardware threads, the high-bandwidth experiments use only 11 workload threads to allow for evaluation of the online approach without contention between the application and sampling threads. We examine the performance overhead of reserving one hardware thread for online sampling in Section 8.2.1.

All of our experiments measure *steady-state* performance and energy consumption. For each DaCapo run, 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 each SciMark run, the harness continuously iterates the workload, starting a new operation as soon as a previous operation completes. Each run includes a warmup period of 1 minute and an iteration period of at least 4 minutes. The score for each SciMark run is the average time it takes to complete one operation during the iteration period.

Baseline configuration. Our cross-layer framework disables channel interleaving in the system BIOS and uses a custom Linux kernel to allocate physical memory pages to individual DRAM modules. Since we wish to isolate the performance and energy impact of our custom Java VM and data partitioning strategies, our baseline configuration uses the same platform and custom kernel configuration. To approximate the effect of channel interleaving, the baseline configuration fills demands with physical memory from alternating channels using a round-robin policy. For our benchmarks, this configuration exhibits very similar performance and energy as a system with an unmodified Linux kernel with channel interleaving enabled (Jantz et al. 2015), which is the default for most commercial off-the-shelf server systems.

8.2 Evaluation of Different Partitioning Strategies

To evaluate the offline partitioning strategy, we profile each application using the small program input and partition their allocation sites using knapsack capacities of 1%, 2%, 5%, 10%, and 20%. We then use these partitionings to color object allocations during an evaluation run with a larger input size. For the online approach, we implement a mechanism to vary the rate of call stack sampling by programming the profiling thread to sleep for different amounts of time between samples. We evaluate the online approach with a *continuous sampling* configuration that does not sleep at all between samples as well as with six other configurations that use the following sleep intervals: 10μs, 100μs, 1ms, 10ms 100ms, and 1 second. Each online configuration also uses a "cool down" mechanism that periodically marks all methods and classes that were not sampled in the previous interval as cold. After some experimentation, we opted to use a constant cool down rate of 0.1 seconds. This approach allows for a reasonable rate of adaptation and makes it easy to control and compare the aggressiveness of the different sampling frequencies.

We evaluated the partitioning schemes using three different cross-layer configurations. While each configuration may reserve a different number of memory devices (DIMMs) for the hot or cold space, all of the experiments use similar physical page allocation strategies for mapping colored data to the memory hardware. For data in the cold space, the OS allocates all of the pages from one DIMM before moving on to the next DIMM and therefore minimizes the number of modules in use. In contrast, the demands for the hot space are filled using physical pages from alternating memory channels (in a round-robin fashion) to maximize bandwidth to the hot program objects.

8.2.1 Overhead of Online Sampling. We first evaluate the performance overhead of sampling in our online approach. For these experiments, the VM samples the application call stacks at different

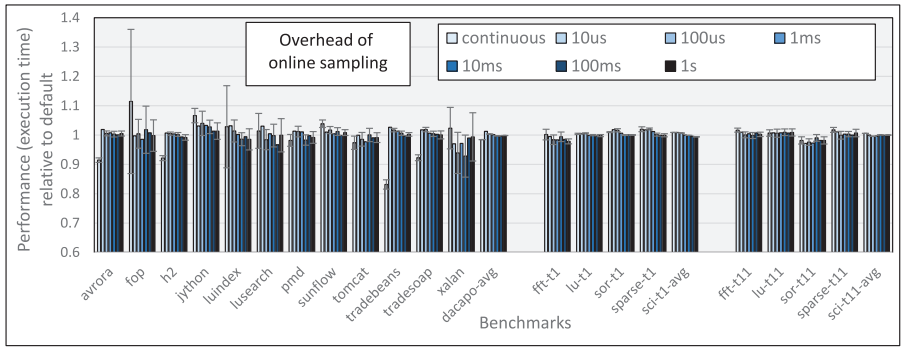

Fig. 6. Performance (execution time) with online sampling relative to default.

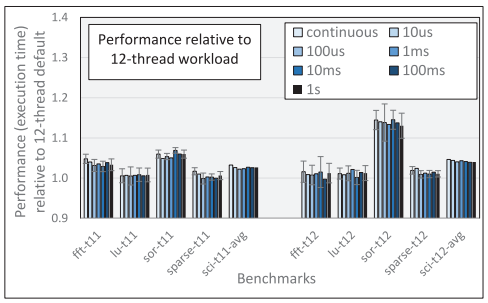

Fig. 7. Performance with online sampling relative to default with 12 application threads.

Table 4. Performance/Energy of 11-Thread SciMark Relative to 12-Thread SciMark

Bench	Perf	CPU J	DRAM J
fft	1.031	1.028	1.020
lu	0.999	0.988	0.994
sor	1.080	1.050	1.025
sparse	1.000	0.994	0.986
geomean	1.027	1.015	1.006

rates but does not enforce object placement decisions. Figure 6 displays the performance of each benchmark with each sampling rate. Thus, in most cases, the online profiler has little to no impact on program performance. In a few cases, especially with very high frequency sampling, performance improves, perhaps due to interactions with processor caches or JIT compilation heuristics.

This result makes sense, because none of the selected benchmarks use more than 11 application threads, and so there is always at least one hardware thread available for online sampling. However, reserving one core for online sampling may still hurt performance if the application is well-parallelized and utilizes all available computing cores. To evaluate the impact of our approach on such programs, we conducted another set of experiments with SciMark using 12 concurrent software threads for each workload. Figure 7 shows the performance of the online approach with the 11-thread and 12-thread versions of SciMark relative to the 12-thread configuration with no online sampling. Table 4 also gives the performance, CPU energy, and DRAM energy of the default

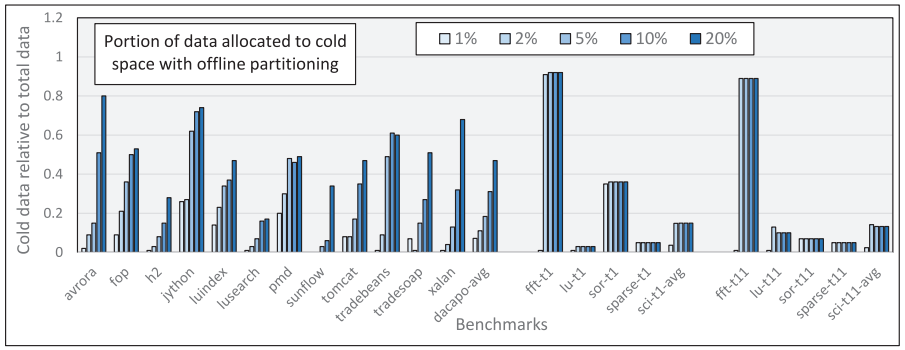

Fig. 8. Portion of cold data with offline partitioning strategies with different knapsack capacities.

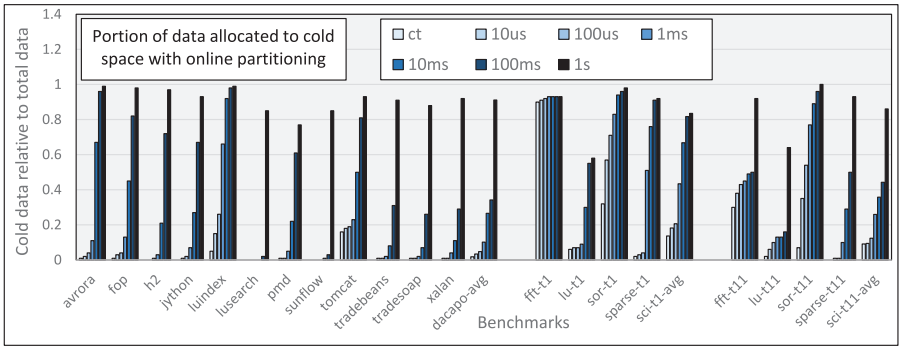

Fig. 9. Portion of cold data with online partitioning strategies with different sampling frequencies.

11-thread SciMark relative to the default 12-thread version (lower is better). The results show that two workloads, *fft* and *sor*, exhibit some performance and energy penalties when only 11 software threads are used, regardless of whether online sampling is enabled (at any frequency). For *sor*, the 12-thread configuration with online sampling yields even larger performance losses (of about 14%).[8] Thus, it is important to consider the availability of compute resources when applying the online approach. The remaining evaluation assumes that there is at least one hardware thread available for online sampling, and therefore uses the 11-thread configuration as baseline for the multi-threaded SciMark experiments.

8.2.2 Cold Heap Size. Figures 8 and 9 show the aggregate size of objects assigned to the cold space as a percentage of the total size with each of the object partitioning strategies. For example, Figure 8 shows that the 1% knapsack partitioning scheme for the *sor (1-thread)* workload finds a set of allocation sites that accounts for about 32% of total heap size. In the profile run with smaller input, these same allocation sites account for less than 1% of memory accesses.

We can make a few observations from these results. First, with the DaCapo benchmarks, increasing the knapsack capacity increases the cold space size for the offline partitioning strategy

[8]There is not much performance variation between sampling rates, because our current profiler always reserves an entire core for the sampling thread, even when it uses a very low frequency. We use this approach because it makes it easier to control the amount of time between each sample and compare different sampling frequencies. We have also found that removing this restriction, and allowing the OS to schedule each thread on any core, eliminates about half of the performance losses with lower frequency sampling ($\geq$10ms).

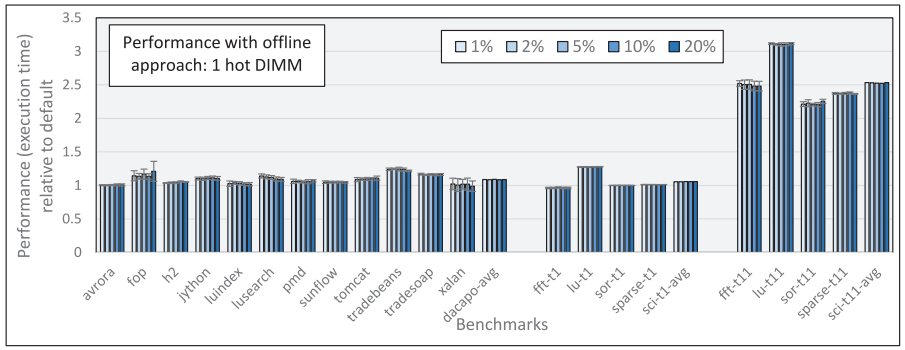

Fig. 10. Performance of offline partitioning strategies with unconstrained capacity relative to default.

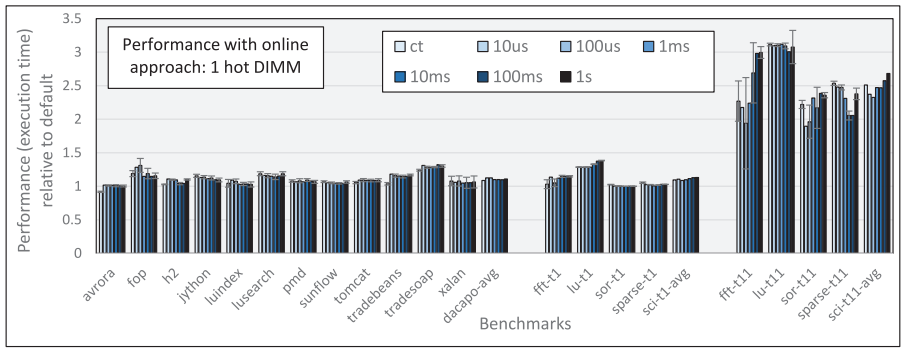

Fig. 11. Performance of online partitioning strategies with unconstrained capacity relative to default.

as expected. Second, however, with the exception of the very smallest capacity (1%), the knapsack capacity has very little effect on the SciMark benchmarks. These workloads allocate the vast majority of their heap objects from only a few allocation sites, and so there is less distinction between different knapsack sizes. Third, increasing the sampling interval for the online approach reduces the portion of objects assigned to the hot space, as expected. Fourth, however, the effect of different sampling intervals can vary significantly depending on the benchmark. For some benchmarks, such as *lusearch*, relatively long sampling intervals (≤100ms) assign most program data to the hot space, while for others, such as *fft (1-thread)*, even the most aggressive sampling rate assigns most of the objects to the cold space.

8.2.3 Performance and Energy Evaluation.

Evaluation with Unconstrained Capacity. The first configuration models the scenario where there is always enough capacity in a fixed set of memory devices to contain all of the application's hot program data. Since the capacity of each DRAM module in our platform is always larger than the requirements of hot program data, the implementation of this configuration is straightforward. All hot program data are simply placed on a single DIMM, and the cold data are assigned to the other three DIMMs.

Figures 10 and 11 display the performance of the offline and online partitioning strategies with unconstrained capacity relative to the baseline configuration. In general, most of the low-to-medium bandwidth benchmarks (i.e., DaCapo and single-threaded SciMark) experience little to no performance losses compared to default. For these workloads, only one memory channel is

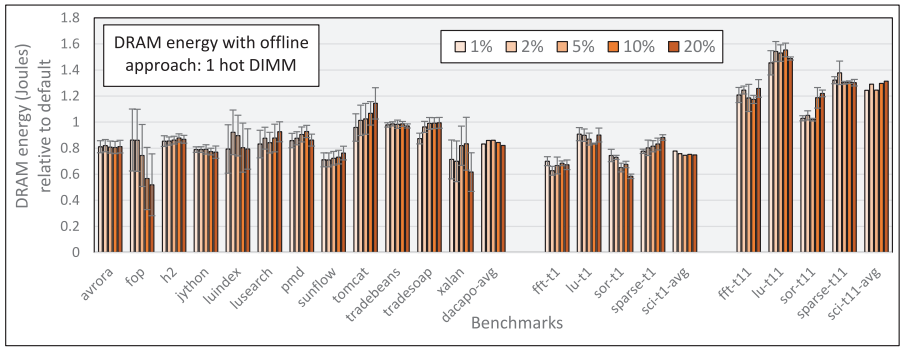

Fig. 12. DRAM energy with offline partitioning strategies with unconstrained capacity relative to default.

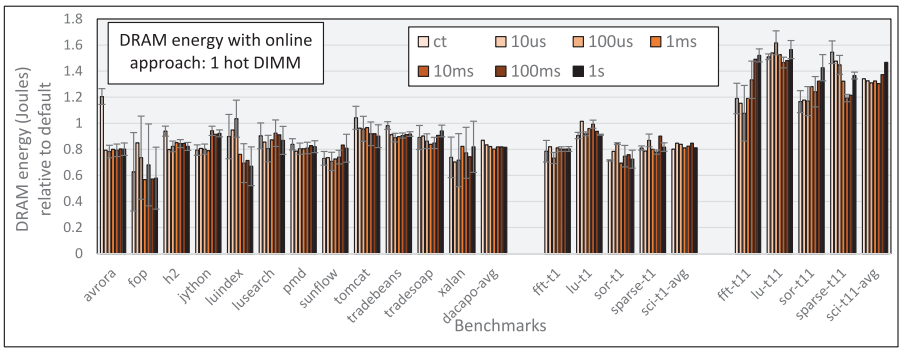

Fig. 13. DRAM energy with online partitioning strategies with unconstrained capacity relative to default.

sufficient to provide all of the necessary bandwidth. Some of these benchmarks, such as *lu (1-thread)*, do exhibit performance losses due to their higher-bandwidth requirements. Others, such as *fop* and *tradebeans*, show that, even if one memory channel is sufficient to support all of the application's memory traffic, the overhead of classifying objects at runtime can still cause slowdowns, in some cases. On average, the offline approach performs slightly better than the online approach for the low-bandwidth benchmarks. The best case offline configuration is, on average, 7.6% (knapsack capacity = 20%) and 5.1% (knapsack capacity = 2%) slower than default with the DaCapo and single-threaded SciMark benchmarks, respectively, while the best case online approach is 9.0% (100ms sampling) and 9.0% (continuous sampling) slower than default with these benchmark sets.

In contrast, the multi-threaded SciMark workloads often exhibit severe performance losses (2× to 3× worse than default), because one memory channel does not provide enough bandwidth for their hot data. Interestingly, some configurations of the online approach are able to obtain better performance by more evenly distributing memory traffic to the hot and cold devices. Later in this section, we describe a technique that relies on a similar effect to mitigate most of these performance losses.

Figures 12 and 13 show the DRAM energy of the custom partitioning strategies relative to the default configuration. Thus, the framework clearly achieves our goal of DRAM energy savings for the low-bandwidth workloads. On average, the best partitioning strategy for DaCapo (online, 1ms sampling rate) reduces DRAM energy by 19.8%, and by 25.5% for the single-threaded SciMark

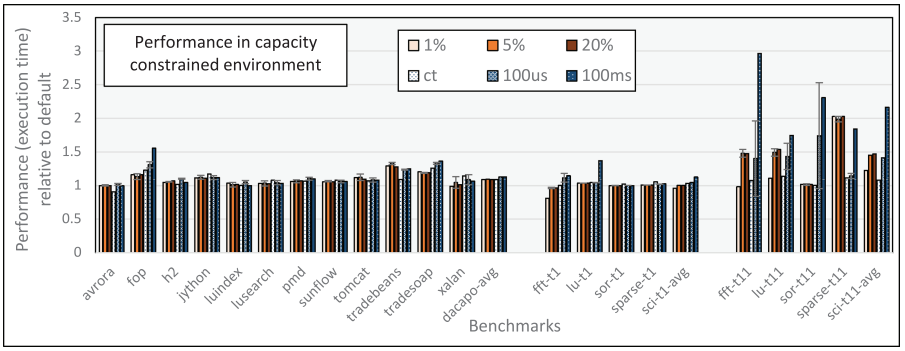

Fig. 14. Performance of different partitioning strategies with capacity constraints relative to default.

workloads (with the offline, 5% knapsack). For the high-bandwidth multi-threaded SciMark benchmarks, although the average rate of DRAM power consumption is significantly reduced (by more than 50%, on average), all of the strategies actually increase energy consumption due to the large performance losses incurred by these workloads.

We find that the partitioning strategies have almost no impact on CPU power, with two exceptions. Multi-threaded SciMark performs so poorly that the rate of CPU energy consumption decreases by 15%, on average, compared to default. Despite this reduction, these benchmarks still consume 2× more CPU energy per operation than default, on average. We also find that continuous online sampling *increases* CPU power by more than 22.2% and 24.4% for the single-threaded SciMark and DaCapo benchmarks, respectively. In many cases, continuous sampling prevents the CPU from transitioning more frequently into low-power states. Reducing the sampling rate to only once every 10μs completely eliminates the additional power consumption. Thus, if the CPU is under-utilized, using an "always-on" strategy for online profiling can thwart potential energy savings.

Evaluation with capacity constraints. The next set of experiments simulates a *constrained capacity environment* where each application must use *all* of the platform's DRAM modules to satisfy its capacity requirements. To implement this configuration, we first measure the peak resident set size (RSS) of each benchmark using a separate (default) run. For each experimental run, the framework configures the hot memory devices to have a maximum capacity of only 25% of the peak RSS. During execution, if the partitioning scheme allocates more data in the hot space than is able to fit in the current set of hot devices, then the OS will interrupt the application, add a new (previously cold) DIMM to the hot set of memory devices, and then remap the application's data across the new set of hot and cold devices. Thus, partitioning strategies that assign more data to the hot space will spread their data across more DRAM devices, potentially driving energy costs higher.

Figures 14 and 15 show the performance and DRAM energy of the partitioning strategies with the constrained capacity configuration. To simplify presentation and reduce clutter, these figures only show results for a subset of partitioning strategies. To present a range of tuning options, we selected configurations that are (1) very aggressive (1% knapsack, continuous sampling), (2) somewhat aggressive (5% knapsack, 100μs sampling), and (3) more conservative (20% knapsack, 100ms sampling) when assigning application data to the hot space.

In general, the lower-bandwidth (DaCapo and single-threaded SciMark) benchmarks exhibit similar performance with capacity constraints as with unconstrained capacity. In a few cases, such as *fop* and *tradebeans*, remapping the application's data to model capacity constraints significantly degrades performance relative to the unconstrained capacity configuration. Other benchmarks,

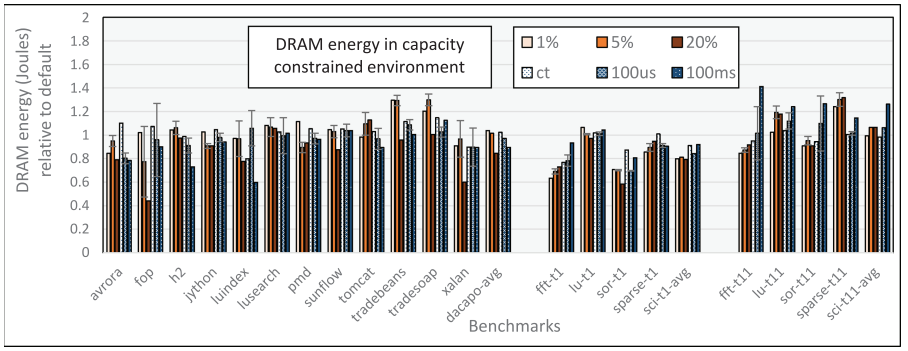

Fig. 15. DRAM energy of different partitioning strategies with capacity constraints relative to default.

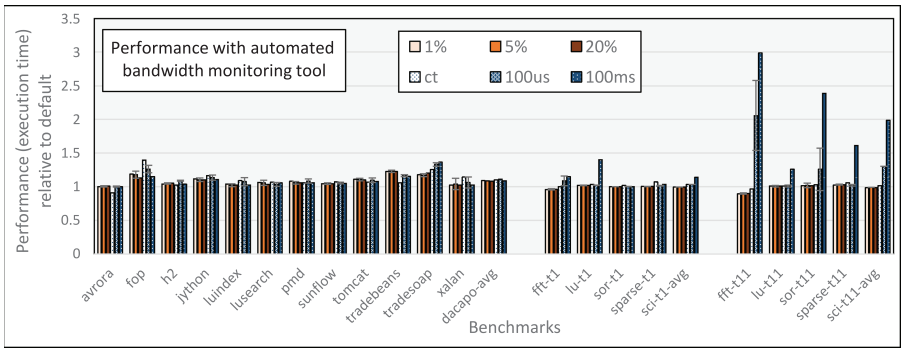

Fig. 16. Performance of partitioning strategies with automatic bandwidth monitoring relative to default.

such as *fft (1-thread)* and *lu (1-thread)*, show significant performance improvements for both the offline and online approaches. For these workloads, the partitioning schemes often assign more than 25% of application data to the hot space, and thus, the cross-layer framework allocates their data across multiple DRAM modules. Since the data actually does require frequent access, remapping it across multiple channels increases bandwidth and improves performance. A similar effect improves performance for several of the higher-bandwidth multi-threaded SciMark benchmarks, especially in cases where the partitioning scheme is more aggressive in assigning data to the hot space, such as with *offline-1%* and *online-ct*.

The DRAM energy results in Figure 15 show that capacity constraints can reduce the energy benefits of our approach or may even make it counterproductive. For instance, several DaCapo workloads, such as *jython* and *sunflow*, consume 20% to 50% more DRAM energy in a constrained capacity environment. The results also show some variation in the effectiveness of different partitioning schemes. For example, with the DaCapo benchmarks, partitioning schemes that are less aggressive in assigning data to the hot space, such as *offline-20%* and *online-100ms*, tend to consume less DRAM energy than more aggressive schemes. Since most of these benchmarks do not actually require much bandwidth, distributing their data across additional DRAM channels increases power consumption without improving performance. In contrast, workloads that require higher bandwidths, such as *fft (11-thread)* and *lu (11-thread)*, achieve better performance and efficiency by increasing the number of active channels.

Evaluation with automatic bandwidth monitoring and remapping. Figures 16 and 17 show the performance and DRAM energy of the partitioning strategies with the automatic bandwidth

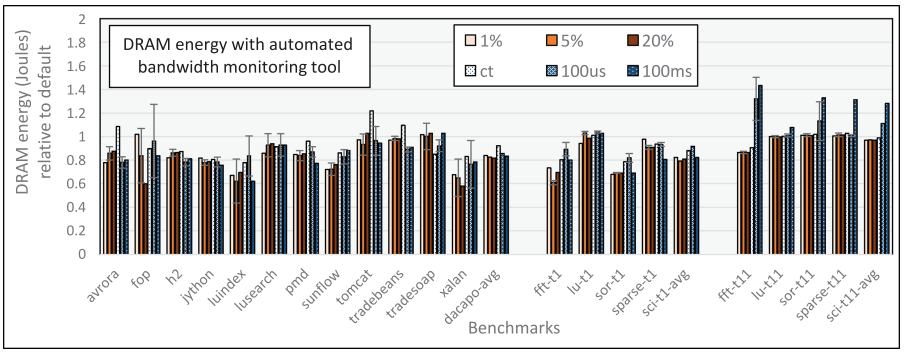

Fig. 17. DRAM energy of partitioning strategies with automatic bandwidth monitoring relative to default.

monitoring and remapping tool with no constraints on the capacity of the hot devices. Thus, in most cases, the bandwidth monitoring tool successfully mitigates the performance losses of the multi-threaded SciMark benchmarks. As expected, these high-bandwidth benchmarks consume about the same amount of DRAM energy as the default configuration, because the bandwidth tool distributes their hot objects across the same number of memory devices. For the offline approach, CPU power is also similar to the default, while the online approach may still require additional CPU power (1.3% more, on average) with the continuous sampling configuration. The less aggressive online strategies still exhibit significant performance losses with the high-bandwidth workloads. Due to their low frequency, these strategies are unable to accurately distinguish hot and cold program data. Thus, remapping the hot space to a larger number of devices fails to redistribute the workload's memory traffic, because most of the accesses are to data in the cold space.

As expected, this technique has less performance or energy impact on the lower-bandwidth benchmarks. With the exception of *lusearch*, which exhibits a 10% performance improvement with data remapping, the DaCapo workloads do not generate enough bandwidth for the tool to remap their data from the initial configuration with only one hot device. With SciMark, the tool remaps the data of both *lu-1t* and *sparse-1t* to a larger number of devices early in the run, regardless of the partitioning strategy. In the case of *sparse-1t*, this remapping increases DRAM energy but does not have any beneficial impact on performance. This result demonstrates a limitation of our current approach, which is that it cannot distinguish cases where bandwidth is limited by the application, and therefore should not be remapped, from when it is limited by the current hot/cold device configuration. One potential avenue to overcome this limitation would be to extend our tool to undo a remapping operation if the new device configuration does not actually increase system bandwidth.

Although some benchmarks, such as *lusearch*, exhibit brief spikes in bandwidth, the conservative remapping policy prevents a series of expensive migrations. We did find that a more aggressive scheme, which remaps data after observing only one sample outside the current bandwidth range, results in substantial performance losses in some cases (e.g., *lusearch* degrades by more than 107%), with no additional benefits. Additionally, we observed that all of our benchmarks exhibit relatively stable bandwidth throughout the entire run. Indeed, in every case that a benchmark remaps its data, it only does so early in the run and quickly settles into a single configuration that it uses for the rest of the run.

To evaluate the effectiveness of this approach when application bandwidth changes over time, we modified the SciMark harness to run 10 iterations of each benchmark within a single VM instance. Each iteration executes the benchmark for 1 minute with a different random number of

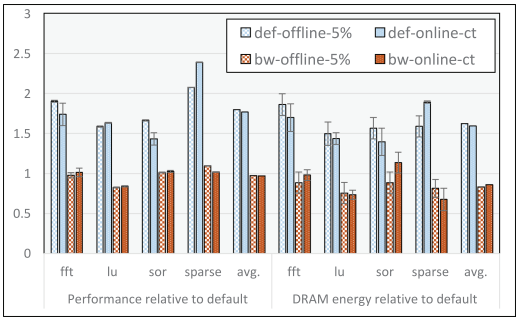

Fig. 18. Performance and DRAM energy of SciMark with changing bandwidth relative to default.

Table 5. Number of Remapping Operations and Gigabytes Remapped for Each SciMark Workload

	offline-5%		online-ct	
Benchmark	**# ops**	**GBs**	**# ops**	**GBs**
fft	12	1.09	11	15.14
lu	3	8.63	2	7.71
sor	11	3.26	8	2.42
sparse	3	9.01	3	7.48

(up to 10) threads to induce changing bandwidth requirements. For comparisons across runs, we use the same random seed for each experiment. Figure 18 shows the performance and DRAM energy of the offline (5% knapsack) and online (continuous sampling) approaches with and without the bandwidth tool relative to default. Not only does the bandwidth tool achieve similar performance as the default configuration, it also reduces DRAM energy (by 16.7% and 13.8%, on average). This finding indicates that the sampling tool is able to automatically and seamlessly adapt data mapping policies to current usage requirements, enabling energy savings during low-bandwidth periods without sacrificing performance during high-bandwidth periods. However, since these experiments have relatively long and stable execution phases, the tool induces only a few remapping operations for each program run, as shown in Table 5. Applications with shorter phase intervals or with more frequent fluctuations in bandwidth will likely induce additional remapping operations, which may reduce the effectiveness of this approach. Similar to results presented earlier in this section, the offline approach does not require any additional CPU energy, while the online approach with continuous sampling consumes almost 4% more CPU power, negating some of the combined energy savings.

9 CONCLUSIONS AND FUTURE WORK

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 automatically collects information about data object usage patterns to steer low-level memory management. Our cross-layer approach integrates the HotSpot VM and a custom Linux kernel to automatically provide efficient distribution of memory resources for a wide range of managed language applications.

While this work primarily focuses on the impact and effectiveness of cross-layer data management on systems with uniform access, there is greater potential for this approach on NUMA systems with larger and more complex memory architectures. In the future, we plan to evaluate this approach with realistic applications in a NUMA environment with hundreds or thousands of GBs of DRAM, and explore the potential challenges and energy savings. Another potential application is for emerging hybrid memory systems, such as those that package conventional DRAM with high-performance "on-chip" memories (Sodani 2015) and/or durable in-memory storage (Intel 2016). Implementing cross-layer management on such heterogeneous memories will allow applications to automatically adapt to the underlying hardware, potentially exposing powerful new efficiencies. Overall, our cross-layer framework shows great promise for enabling existing applications to use memory resources more efficiently, without requiring any source code modifications or recompilations.

REFERENCES

Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page placement strategies for GPUs within heterogeneous memory systems. *ACM SIGPLAN Not.* 50, 607–618. DOI: http://dx.doi.org/10.1145/2775054.2694381

Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 631–644. DOI: http://dx.doi.org/10.1145/3037697.3037706

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI'99)*. 14.

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 335–348.

I. Bhati, M. T. Chang, Z. Chishti, S. L. Lu, and B. Jacob. 2016. DRAM refresh mechanisms, penalties, and trade-offs. *IEEE Trans. Comput.* 65, 1, 108–121.

Stephen M. Blackburn and others. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. 169–190.

Christopher Cantalupo, Vishwanath Venkatesan, and Jeff R. Hammond. 2015. User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies. Retrieved April 5, 2018, from https://memkind.github.io/memkind/memkind_arch_20150318.pdf.

Jonathan Corbet. 2012. AutoNUMA: The Other Approach to NUMA Scheduling. Retrieved April 5, 2018, from https://lwn.net/Articles/488709/.

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. *ACM SIGPLAN Not.* 48, 381–394.

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

V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. 2002. Scheduler-based DRAM energy management. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, New York, NY, 697–702. DOI: http://dx.doi.org/10.1145/513918.514095

Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, New York, NY, 15.

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

Ankita Garg. 2011. Linux VM Infrastructure to Support Memory Power Management. Retrieved April 5, 2018, from http://lwn.net/Articles/445045/.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*. 57–76.

Github.com. 2014a. DaCapo Batik Benchmark Fails. Retrieved April 5, 2018, from https://github.com/RedlineResearch/OLD-OpenJDK8/issues/1.

Github.com. 2014b. DaCapo Eclipse Benchmark Fails. Retrieved April 5, 2018, from https://github.com/RedlineResearch/OLD-OpenJDK8/issues/2.

David Gomez. 2001. MADVISE (2). Retrieved April 5, 2018, from http://man7.org/linux/man-pages/man2/madvise.2.html.

Alex Gonzalez. 2010. Android Linux Kernel Additions. Retrieved April 5, 2018, from http://www.lindusembedded.com/blog/2010/12/07/\\android-linux-kernel-additions/.

Rentong Guo, Xiaofei Liao, Hai Jin, Jianhui Yue, and Guang Tan. 2015. NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 307–318.

Urs Hoelzle and Luiz Andre Barroso. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool.

Hai Huang, Padmanabhan Pillai, and Kang G. Shin. 2003. Design and implementation of power-aware virtual memory. In *Proceedings of the USENIX Annual Technical Conference (ATEC'03)*. 5.

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: Improving program locality. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM, New York, NY, 69–80.

Intel. 2015. Intel®Xeon®Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual. Retrieved April 5, 2018, from https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html.

Intel. 2016. 3D XPoint. Retrieved April 5, 2018, from http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html.

Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer memory management for managed language applications. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, New York, NY, 488–504.

Michael R. Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A. Doshi. 2013. 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)*. 155–166.

JEDEC. 2009. DDR3 SDRAM Standard. Retrieved April 5, 2018, from http://www.jedec.org/standards-documents/docs/jesd-79-3d.

Andi Kleen. 2004a. MBIND (2). Retrieved April 5, 2018, from http://man7.org/linux/man-pages/man2/mbind.2.html.

A. Kleen. 2004b. *A NUMA API for Linux*. White Paper. SUSE Labs.

Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. 2000. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, 105–116.

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

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, 213–224.

D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. 2009. Transcendent memory and Linux. In *Proceedings of the Ottawa Linux Symposium*. 191–200.

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

M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *Proceedings of the High Performance Computer Architecture (HPCA'15)*. 126–136.

Oracle. 2017. Java HotSpot Virtual Machine Performance Enhancements. Retrieved April 5, 2018, from https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. 1–12.

Avinash Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS'15)*. IEEE, Los Alamitos, CA, 1–24.

SPEC2008. 2008. SPECjvm2008 Benchmarks. Retrieved April 5, 2018, from http://www.spec.org/jvm2008/.

Sun-Microsystems. 2006. Memory Management in the Java HotSpot Virtual Machine. Retrieved April 5, 2018, from http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

Micron Technology Inc. 2001. *Calculating Memory System Power for DDR*. Technical Report TN-46-03. Micron Technology.

Rik van Riel. 2014. Automatic NUMA Balancing. Retrieved April 5, 2018, from https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Comput. Sci. Eng.* 17, 2, 73–82.

Malcolm Ware, Karthick Rajamani, Michael Floyd, Bishop Brock, Juan C. Rubio, Freeman Rawson, and John B. Carter. 2010. Architecting for power management: The IBM® POWER7 approach. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'16)*. IEEE, Los Alamitos, CA, 1–11.

V. M. Weaver, D. Terpstra, and S. Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. 215–224. DOI: http://dx.doi.org/10.1109/ISPASS.2013.6557172

Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2012. Intel Performance Counter Monitor. Retrieved April 5, 2018, from http://www.intel.com/software/pcm.

Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, 177–188.

Received August 2017; revised February 2018; accepted March 2018