Automated Program Profiling and Analysis for Managing Heterogeneous Memory Systems

Adam Palmer Howard
University of Tennessee, Knoxville, ahowar31@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Adam Palmer Howard entitled "Automated Program Profiling and Analysis for Managing Heterogeneous Memory Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael R. Jantz, Major Professor

We have read this thesis and recommend its acceptance:

Gregory D. Peterson, James S. Plank

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Automated Program Profiling and Analysis for Managing Heterogeneous Memory Systems

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Adam Palmer Howard

December 2017

For Chae, my best friend and constant companion. I couldn't have gotten this far without you.

Acknowledgments

I would like to thank my loving wife, Chae, for always being there for me. I would also like to thank my adviser, Dr. Michael Jantz, for his teaching and guidance, and my family, friends, Dan, and Lily for their constant love and support.

Abstract

Many promising memory technologies, such as non-volatile, storage-class memories and high-bandwidth, on-chip RAMs, are beginning to emerge. Since each of these new technologies present tradeoffs distinct from conventional DRAMs, next-generation systems are likely to include multiple tiers of memory storage, each with their own type of devices. To efficiently utilize the available hardware, such systems will need to alter their data management strategies to consider the performance and capabilities provided by each tier.

This work explores a variety of cross-layer strategies for managing application data in heterogeneous memory systems. We propose different program profiling-based techniques to automatically partition program allocation sites into sets corresponding to expected allocation and usage patterns. As the application executes, it consults the collected guidance to assign new data objects to distinct regions, which can be independently managed and mapped to distinct types of hardware memory devices. Our approach is fully automatic, does not rely on any non-standard hardware or architectural modifications, and is flexible enough to adapt management strategies as the application behavior changes. Evaluation with a set of standard benchmarks (SPEC cpu2006) shows that our guidance-based approach outperforms, and can even improve, other state-of-the-art management techniques.

Table of Contents

List of Tables

List of Figures

Chapter 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 [59, 71, 56, 70, 58, 7]. These trends are driving bandwidth and capacity requirements to new heights and have led to the adoption of larger, and more complex, data storage and management systems. However, further scaling faces a number of difficulties. Processor and memory speed disparities continue to persist, and increasing demands and complexity are exacerbating the performance losses of long memory latencies [8, 13, 22, 33, 45, 49, 53, 63]. DRAM energy consumption is also a major concern as exascale processing enters the near horizon of computing and converges with Big Data environments [60, 66]. Memory systems already account for about 30-50% of overall power consumption in a typical server node [41, 27, 68, 42, 65]. Furthermore, since DRAM power is directly related to the number and size of memory modules, any attempt to limit this fraction will necessarily constrain its capacity.

The emergence of several new memory technologies has given reason for optimism in the face of these challenges. Storage-class memories (SCMs), such as STT-RAM [37], PCM [38, 40, 69], and ReRAM [26], enable durable byte-addressable storage, with random access latencies multiple orders of magnitude faster than state-of-the-art solid state and spinning disks. Several SCM technologies also allow finer resolution semiconductor integration and, since they do not require refresh power, are more energy efficient than traditional DRAM, all while providing far greater capacity. Intel's 3D XPoint technology [29], for example, will make 512GB DIMMs possible in its introductory generation itself – furnishing up to 12 TB of memory capacity in a 4 socket system! Similarly, another emerging technology, often referred to as "on-package" or "die-stacked" memory, places one or more 3D stacks of memory inside the same package as the processing unit to deliver orders of magnitude higher bandwidth, and thus has great potential to address the problems of the memory wall.

Despite their promise, each of these new technologies also come with their own set of drawbacks. For example, first-generation SCMs are not expected to match the performance of modern volatile memory, with projected longer access latencies, as well as reduced and less uniform read/write bandwidths than DRAM [36]. Likewise, while on-package memory delivers high access bandwidth, it is generally only available in limited capacity: for example, in Intel's Xeon Phi processor, only eight MCDRAM modules (2GB each) are currently available [62]. Thus, any enterprise or scientific computing solution that has a potential footprint ranging into hundreds or thousands of GBs must manage and apportion this limited capacity judiciously at different phases of computation.

In the immediate horizon, high-end computer systems will begin to provide multiple tiers of memory storage, including: 1) an on-package tier with high-performance but limited capacity, 2) a tier of conventional DRAM (e.g. DDR*), and 3) a non-volatile memory tier that provides durable storage with high capacity, but with less bandwidth, and longer latencies for reads and writes. Current memory management strategies must be altered to take advantage of the different types of memory in each tier.

One approach is to exercise the faster, lower capacity tier(s) as a large, hardware-managed cache. Several researchers have applied this approach, most often with on-package, die-stacked memory as a large L4 cache to conventional memory[14, 43, 17, 44], but also with DRAM as a cache for SCM[51]. While this strategy provides some immediate advantages (specifically, software-transparency and backwards compatibility, it is often inflexible, less efficient, and reduces the system's available capacity. It also imposes unpalatable architectural costs as the high bandwidth tier must either be implemented as a tagless direct mapped (non-associative) cache, or it requires logic and storage for associative tags [52]. These issues become even more problematic as the capacity of the hardware-managed tier(s) increases, and so scalability of this technique is a major concern.

An alternative approach is to push much of the responsibility for placement of data in different memory tiers from the hardware to the upper-level software. With this approach, either the operating system (OS) by itself, or the OS in conjunction with application software, may assign data into different memory tiers, and may further facilitate migration of data between those tiers as needed.

Some recent work has proposed system-level monitoring of per-page memory traffic to assign application data to hardware tiers without requiring application software to be cognizant [50]. While this approach preserves application transparency, it is strictly reactive and relies on non-standard hardware to predict data usage patterns. Another common approach is for user-level applications to directly control the placement of their data through the use of source code annotations [6, 16, 1]. This approach permits developers to coordinate tier assignments with finer-grained management of program objects and usage patterns, but requires expert knowledge and manual modifications to source code.

Our approach aims to combine the power and control of application-directed management with the transparency and adaptability of OS- and hardware-based approaches. We rely on offline profiling of memory usage behavior associated with program allocation sites. Prior to execution, we use a custom tool chain to divide the allocation sites into distinct sets corresponding to different tiers in the underlying memory hardware. The application then uses the resulting profiling information during subsequent program runs to guide upper- and lower-level data management across the hybrid memory hierarchy.

For this thesis, we developed a set of system utilities, dynamic binary instrumentation, and custom allocation routines to evaluate our approach. Specifically, this work consists of the following primary components:

1. Memtracer - Dynamic binary instrumentation, built using Intel's Pin binary translation framework, for recording and profiling an application's memory access patterns.

2. Knapsack/Hotset - A set of scripts to divide program allocation sites into distinct sets based on their observed usage behaviors.

3. Marena - A custom memory allocator, based on the jemalloc arena allocator, which uses allocation site context to guide page placement.

4. Experipy - A scripting framework for orchestrating our experiments as well as managing and parsing the resulting data.

Later, these components were used to conduct a study on the effectiveness of different guidance-based strategies for hybrid memory management. Our study relies on Effler's extensions to the Ramulator DRAM simulator [Kim et al., 15], to estimate the performance of the proposed strategies in a hybrid memory system. Significant findings from this study conclude:

1. Our application-level, guidance-based approach is able to provide comparable performance to hardware- and OS-based approaches.
2. Fine-grained grouping of application objects by their usage patterns can have a significant impact on DRAM efficiency, frequently increasing the DRAM's row-buffer hit ratio.

The rest of this thesis is organized as follows: Chapter 2 describes related work. Chapter 3 provides a high-level overview of our proposed hybrid memory management strategies. Chapter 4 describes the design and implementation of the software developed for this work. Chapter 5 provides details about our experimental platform and benchmark set. Chapter 6 describes our evaluation and presents results. Chapter 7 discusses future directions for this work, and Chapter 8 concludes the thesis.

Chapter 2

Related Work

The architecture and systems communities have recently proposed a number of frameworks, techniques, and strategies for managing heterogeneous memory systems [14, 43, 17, 44, 51, 50, 1, 6, 16, 57, 11, 2, 19]. Of these, the works most closely related to our approach are those that employ profiling of application data usage to direct data placement across the memory hierarchy. Agarwal et al. [1] employ offline profiling to find frequently accessed data structures, and then use this information to insert tiering hints into the application source code. Dulloor et al. [16] manually tag data structures in the application, and then use an automated profiling tool to assign each tagged structure to the appropriate tier. While these works demonstrate some of the benefits of application guidance for hybrid memory systems, our approach combines program profiling with online allocation site detection and a custom arena allocator to enable memory usage guidance without source code modifications.

Jantz et al. [31] used memory coloring to divide a program's objects based on user controlled traits and then mapping those collections to memory using various policies aimed at boosting bandwidth or reducing power consumption. We utilize a similar concept of dividing the memory into arenas based on how frequently those objects are utilized, and mapping as many of the hot objects as will fit into the upper tier of memory to improve bandwidth.

Also related are the various architectural, compiler, and runtime allocation strategies that have been proposed and used to optimize cache and/or memory utilization on conventional architectures [5, 61, 25, 9, 39, 10, 24, 28, 32, 67, 4, 3, 64, 20]. Since our approach affects object layout in the virtual address space, it may induce effects that have been observed by these previous studies. For instance, Sudan et al. [64] propose OS/architectural techniques that control the placement of data to improve utilization of the DRAM row buffer. As we discuss in Section 6.4, the profile-guided arena allocation schemes produce data layouts that similarly improve DRAM efficiency and performance.

Chapter 3

Approach

In this chapter, we discuss our design for collecting and applying memory usage guidance during hybrid memory management.

3.1 Profiling Application Memory Usage

Since it would not be practical to conduct detailed profiling of object usage patterns online, our approach applies profiling during a separate program run, and uses program allocation sites to relate the collected information to the next run. The profiling run collects the following information for each allocation site: 1) the size of the data allocated at the site, and 2) the total number of accesses to data allocated at the site. Since modern systems do not actually reserve physical memory for allocated data until it is accessed, our profiling tool does not count pages that are allocated, but not touched, towards the size of the allocation site.

3.2 Automatic Allocation Site Partitioning

After collecting the profiling data, the next step is to partition the allocation sites to create guidance for the hybrid memory manager. Using this guidance, the upper- and lower-level memory managers will work together to assign data corresponding to different sets of allocation sites to different tiers in the memory hierarchy. Thus, the goal is to construct a partition that takes advantage of the performance and capabilities of the underlying hardware in consideration of its capacity and/or usage constraints. For this work, we consider a two-level memory system with a fast upper-level tier with limited capacity, and a lower-level tier with more space, but worse performance. We propose two simple approaches for constructing allocation site guidance for this system.

The first approach, which we call *knapsack*, is inspired by Jantz et al.'s [30] approach for partitioning hot and cold data to reduce memory power consumption. In their formulation, the partitioning problem is expressed as an instance of the classical 0/1 knapsack optimization problem. Each allocation site is an item, which has a value equal to the total number of accesses to data allocated at that site, and a weight equal to the size of the data allocated at the site. The optimization problem then is to select items (allocation sites) such that the combined value (access counts) is maximized without the combined weight (size) exceeding the capacity of the knapsack. Although the knapsack problem is NP-complete, there exist well-known polynomial-time algorithms that can approximate the solution to any specified degree[48].

While the knapsack approach is theoretically optimal, it may fail to assign some hot data to the upper-level memory if it is created at a site that is not able to fit within the knapsack capacity. In such cases, a better approach might be to assign the allocation site to the faster tier with the expectation that a portion of its data will not be able to fit within the limited capacity of the upper-level memory at run-time. Thus, our second approach, called *hotset*, favors adding more traffic to the upper tier over staying within its capacity constraints. To construct a hotset, we sort the allocation sites by the number of accesses per byte of allocated data. Then, we simply add sites with the highest number of accesses per byte to the hotset until we exceed its capacity. In this way, the hotset approach always allows more data into the hotset than is specified by its capacity. To compare the partitioning strategies across different workloads, we select knapsack and hotset capacities as a percentage of the total size of allocated data in the profile run. For example, with a capacity of 12.5%, the knapsack approach selects allocation sites such that the aggregate size of the data created at these sites together accounts for no more than 12.5% of the total size of all data created during the profile run, while the hotset would select the set of allocation sites that just passes this threshold by including the next most utilized allocation site that broke the limit.

3.3 Profile-Guided Arena Allocation

Once the allocation site guidance has been computed, it can then be used to direct memory management during each subsequent run of the application. During a guided run, the application partitions its address space into a set of *arenas*. The arenas consist of independent sets of memory pages that can be independently managed across the tier hierarchy. Using a system interface, such as the NUMA API [35] or memory coloring [31], the application or runtime can also direct the low-level memory manager with additional information about how to manage the pages of each arena. For example, if the guidance indicates that an arena contains data that is expected to require frequent access, the application may instruct the OS to map its virtual addresses to the high-performance memory tier.

Our framework supports two arena allocation schemes: *static arena allocation*, and *per-phase arena allocation*. The static arena allocation scheme employs a single set of allocation site guidance throughout the entire run. At the start of the run, the application creates two arenas: a hot arena for data allocated at sites that the guidance designates as hot, and a cold arena for all other program data. Whenever the application reaches an allocation site, it consults the guidance, and allocates the new data to the appropriate arena.

In some cases, it might be beneficial to adjust the allocation site guidance as usage patterns change throughout the run. The per-phase arena allocation scheme allows the application to incorporate multiple sets of guidance corresponding to different phases of program execution. Allocation sites in the per-phase scheme are grouped by the phases of the program during which they were considered heavily utilized, and these distinct groups were each assigned their own arena. The goal of the per-phase arena allocation scheme is to ensure that hot data and cold data never share the same arena during the same phase of program execution. In this way, the lower-level memory manager is able to independently manage each arena as the program proceeds from phase to phase.

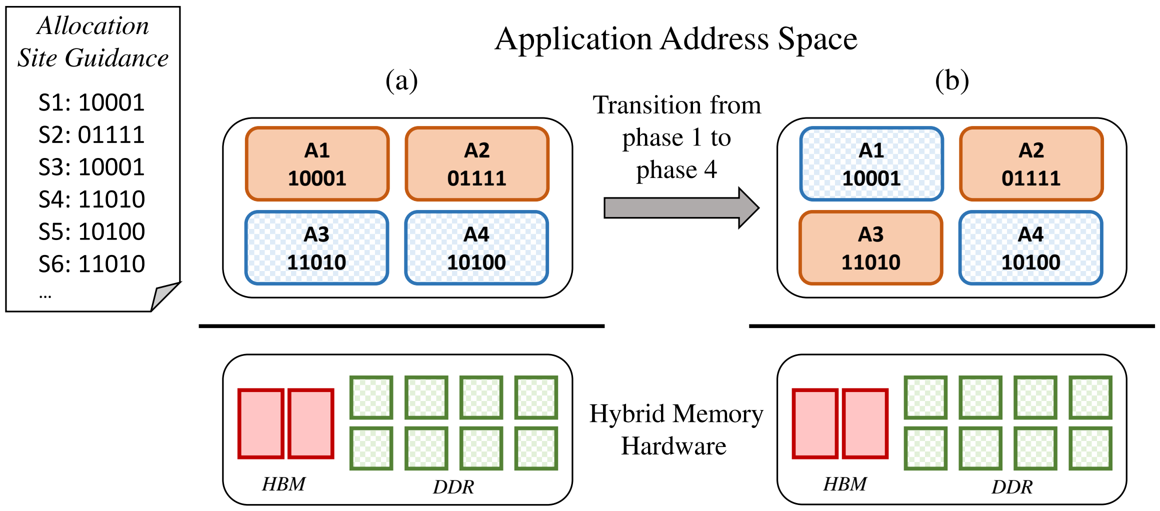

Figure 3.1: Per-phase strategy for managing hybrid memories. (a) In phase 1, A1 and A2 correspond to hot allocation sites and are originally mapped to the HBM tier. (b) On transition to phase 4, the guidance indicates A3 will become hot, and A1 is now cold. The application communicates this guidance to the lower-level memory manager, which may now attempt to remap the data in A1 to DDR and the data in A3 to HBM.

Figure 3.1 illustrates the per-phase arena allocation scheme. To apply this scheme, we divide program execution into a set of phases and compute independent sets of allocation site guidance over each phase. Next, we assign a bit vector to each allocation site that describes the hot/cold classification of the site during each program phase. For example, the bit vector '10100' indicates that the site is hot in phases 3 and 5, and cold in phases 1, 2, and 4. At the start of the guided run, the application creates a set of arenas corresponding to the unique bit vectors in the allocation site guidance. During execution, whenever the application makes an allocation request, it looks up the allocation site's bit vector, and assigns the new data to the corresponding arena. The application or runtime also detects program phase transitions online, perhaps using a pre-constructed model or by inserting instructions that mark the start of each phase. At each phase transition, the application adjusts the hot/cold classification of each arena, and communicates the updated classifications to the low-level memory manager in the OS. The low-level manager interprets the updated guidance and may use it to remap physical memory data corresponding to each arena. Thus, this design adapts data placement to each program phase while allowing virtual addresses to remain unchanged, and avoids the need to update pointers or aliases to migrated data in the application.

Chapter 4

Implementation Details

In this chapter, we describe the implementation of our profiling tools and simulation-based evaluation framework, and provide details for how we setup and run the experiments presented in Chapter 6.

4.1 Associating Memory Usage Profiles with Program Allocation Sites

To collect the necessary memory usage profiles, we developed a custom dynamic binary instrumentation tool using Intel's Pin framework (v. 2.14) [46]. Our custom Pintool intercepts all of the application's allocation and deallocation requests (specifically, all calls to `malloc`, `calloc`, `realloc`, and `free`) and uses the arguments to these routines to build a shadow structure representing the application's current virtual address space. For each allocated region, the shadow structure maintains the following information: 1) the region's allocation site with context, 2) the *access size*, which is the total size of pages within the region that have been accessed at least once, and 3) the total number of accesses within the region. For 2), if the application accesses a portion of a memory region that does not span an entire page, then only the portion of the page belonging to the memory region is added to the total access size.

At the time of each allocation request, we record the region's *allocation site* as the source code filename and line number of each currently active function on the application call stack. To record accesses to each memory region, our tool dynamically instruments the application's memory load and store instructions. The targets of each memory instruction are filtered through an online cache simulator, and any instruction that misses the last level cache is recorded as an access to the target's corresponding memory region. Whenever a memory region is freed, the size and access counts associated with the region are added to a record corresponding to its allocation site. At the end of application execution, the data from all remaining live regions is added to the allocation site records, and the usage profiles associated with each site are then output to a file on disk.

4.2 Selecting Candidate Allocation Sites for Promotion

Once allocation sites have been profiled, it is necessary to select candidate allocation sites to attempt to place on the upper tier. As mentioned previously, The two strategies implemented for this work were Knapsack and Hotset.

4.2.1 The Knapsack Implementation

To efficiently solve the 0/1 knapsack problem, a library was written in C++ and called into using our Python scripting harness. This implementation calculates each allocation site's weight as a percentage of the application's total memory allocated, and it's value as a percentage of total memory accesses that fall within objects from that allocation site. These percentages are rounded to five significant figures, such that the dynamic programming solution of the problem can be utilized. Thanks to the fixed number of significant figures, this means that the problem can be solved in linear time (with a large constant factor based on the number of significant figures) with respect to the number of allocation sites in the input.

4.2.2 The Hotset Implementation

When the Knapsack implementation was found to not be aggressive enough in it's selection, a second selection algorithm was developed. The Hotset approach can best be described as a Greedy approximation of the knapsack problem, where the algorithm is allowed to overshoot the capacity of the knapsack by a single item. This more aggressive algorithm simply sorts the allocation sites by value, taking the next highest value allocation site until it has chosen a site which exceeds the knapsack's capacity.

4.3 Hybrid Memory Management Strategies

Evaluation of our proposed hybrid memory management strategies requires two major components: 1) an allocator that is able to partition application data into distinct arenas according to pre-collected allocation site guidance, and 2) a way to execute or model the effect of guidance-based management strategies across a hybrid memory architecture.

4.3.1 Profile-Guided Arena Allocation

To support the first component, we employ library preloading to dynamically link each evaluation run to a custom shared library. The custom library implements a set of functions to replace calls to the application's standard allocator with calls to our own arena allocation routines based on the popular jemalloc allocator [18].

In most cases, the custom allocator simply uses the corresponding jemalloc arena allocation routine to allocate the request to the appropriate arena. Some calls to `realloc` may request a different arena than was used to allocate the original data. In such cases, the allocator always allocates enough space in the new arena to satisfy the entire `realloc` and moves any surviving data from the original allocation request to the new arena.

On initialization, the program creates a set of arenas corresponding to the application site guidance provided in the input file. When an allocation site is reached during execution, the application first identifies the site through context and then consults the guidance information to determine the arena in which to allocate the request.

To identify each program allocation site, our framework uses the filename and offset of each allocation request as well as seven layers of additional context (method name and offset) from the application call stack. We currently employ the `backtrace_symbols` routine from the C standard library to collect this information during execution, and use string comparisons with the pre-loaded guidance to determine the appropriate arena for each allocation request. While this approach is straightforward and easy to implement, it can incur substantial overhead, especially if the application issues a large number of allocation requests. In a direct comparison of a configuration that uses allocation site detection, but does not actually alter the structure of the heap, to a configuration that does not attempt to detect allocation context, we find an average performance loss of 25.3% across the 18 benchmarks listed in Table 5.2. As expected, the overhead for each benchmark tracks closely with the number of allocations it requests. For example, *dealII* and *omnetpp* incur, by far, the largest performance penalties of all the benchmarks, with degradations of 237% and 333%, respectively.

Since the primary goal of this work is to study the potential benefits of employing automated application guidance during hybrid memory management, our simulation-based experiments do not account for this overhead, and we did not attempt to reduce it any further. However, we expect that additional compiler support to statically mark each allocation site with guidance, and perhaps to clone sites that can only be distinguished with additional context, would almost or completely eliminate these overheads.

4.3.2 Simulation of Hybrid Memory Architectures

Our framework for modeling the behavior and performance of hybrid memory systems adopts and extends the Ramulator DRAM simulator [Kim et al.]. Ramulator is a trace-based simulator that provides cycle accurate performance models for a variety of DRAM standards, including: conventional (DDR3/4), low-power (LPDDR3/4), graphics (GDDR5), and die-stacked (HBM, WIO2) memories, as well as a number of other academic and emerging memory technologies. The simulation engine includes a simple CPU model as well as a memory controller that receives and sends requests to a model of the current DRAM standard. During regular operation, Ramulator reads application memory requests (loads

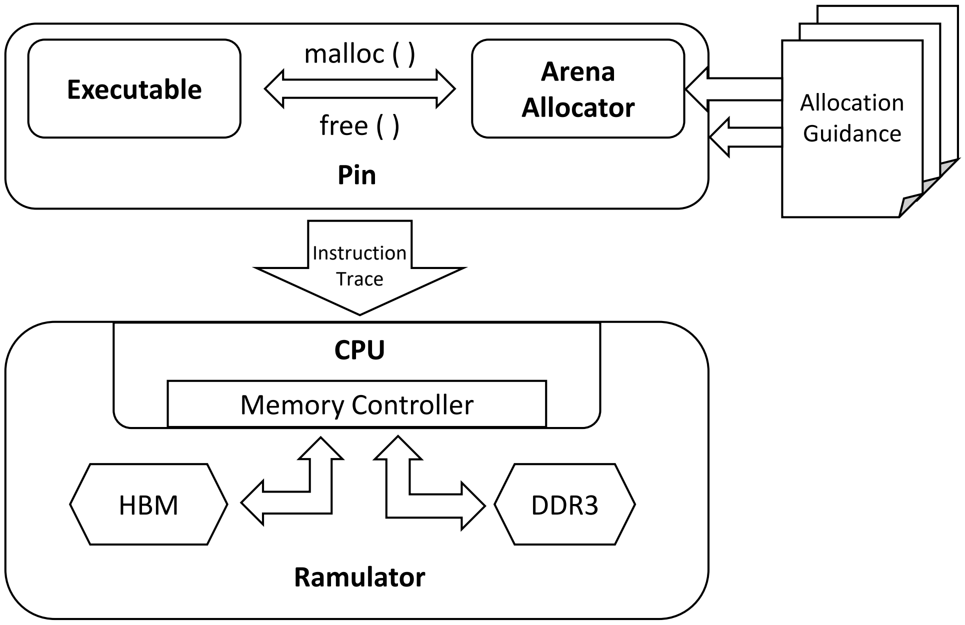

Figure 4.1: Our framework for simulating hybrid memory management strategies.

and stores) from a dynamic instruction trace, which is typically generated using a binary instrumentation tool, such as Intel's Pin [47].

For this work, we utilized extensions made to Ramulator by Dr. Michael Jantz and Chad Effler which modified Ramulator's memory controller to support requests to multiple tiers with distinct DRAM standards *simultaneously.*

This modified version of Ramulator maintains a map of which physical pages correspond to each tier, and sends each request to the appropriate DRAM model depending on its address. To support our proposed guidance-based strategies, the multi-tier Ramulator also accepts an alternative instruction trace format with annotations describing the preferred tier of each memory request. When a page is first accessed, the simulator uses the annotations to map the page to the appropriate tier, depending on the current policy and system configuration.

Figure 4.1 illustrates our approach for simulating the proposed hybrid memory management strategies. At the start of execution, the application connects to a Pintool to generate the instruction trace for the multi-tier Ramulator. The Pin instrumentation filters each memory load and store instruction through an online cache simulator, and outputs instructions that miss the last-level cache directly to a buffer connected to a running instance of Ramulator. At the same time, the application dynamically links to the custom allocator, allowing it to automatically partition its data objects into separate arenas according to the pre-collected memory usage guidance. The Pintool itself is also aware of the guidance provided to the application, and may use it to annotate the instruction trace with each request's preferred memory tier. Ramulator interprets the instruction trace one request at a time, mapping new data to the appropriate memory tier, until completion.

4.4 Managing the Experiment Workload

Over the course of this study, we conducted over 10,000 experimental runs with more than 87 benchmark-input pairs [1]. To manage the execution and output of these experiments, we employed a harness program written as a set of Python scripts.The original harness developed by Dr. Michael Jantz, and expanded by our research team was capable of executing many of our experiments, queuing those experiments onto a cluster using PBS scripts to make our long running simulations tenable, and generating reports based on the output files from those runs.

While the initial harness allowed us to run and organize experiments across a set of named configurations, as our work advanced, we found it difficult to extend the original harness with new configurations. The original harness was not well-modularized, and even configurations with only small differences, such as an additional argument or option, required duplication of significant amounts of code. Major changes, such as those required to integrate Simpoints for our phase-based studies required major reworks and configuration-specific logic within multiple core harness modules. The harness was also difficult to debug, as a failure during an experiment would result in no output from the run at all. These limitations led to the development of a second, complementary experiment harness, designed around the core functionality of the original, but focused on flexibility and improving debugging.

In this new harness, components of an experiment were described as instances of an Executable class. These instances could be composed together following a set of simple

[1]The results presented in Chapter X only use a subset of the benchmarks to show our most important findings

grammar rules, and the resulting composition would then be rendered into a shell script for execution. Using this approach, new types of experiments could be added as separate functions, and without touching the existing code base to integrate them, which greatly improved modularity. To help improve debugging, with the exception of creating the run script and result directory, all actions in the experiment would be performed by, and captured in, the generated shell script, including the copying of files into and out of the experiment directory. This way, in the event of a failure, the user would have access to a complete and executable record of their experiment from which to start debugging.

The final results of this effort were documented, packaged, and published as *experipy* on the Python Package Index as a general-purpose framework for writing computational science experiments. The full documentation describing the framework can be found at `https://experipy.readthedocs.io`.

Chapter 5

Experimental Setup

5.1 Platform Description

We ran all of the simulation experiments on a cluster of Dell PowerEdge 1950 server machines running CentOS 7.2 (Linux kernel version: 3.10.0-327.el7.x86_64). All benchmarks were compiled using gcc (version 4.8.5) with -O2 as well as -g and -rdynamic to facilitate the collection of method names and line numbers for allocation site contexts.

Ramulator's execution model includes a 3.2GHz, 4-wide issue CPU with 128-entry ROB, and assumes one cycle for each non-memory instruction. We modified Ramulator to model a hybrid memory architecture with two tiers: a high-performance tier with limited capacity, and a slower tier with no space restriction. To compare workloads with different space requirements, the capacity of the upper tier is configured to be a percentage of the application's peak resident set size (RSS). All experiments use the unmodified HBM and DDR3 configurations included with Ramulator to simulate the fast and slow tiers, respectively. Table 5.1 displays the rate, timing, data bus width, and bandwidth for these configurations. Although Ramulator includes a DDR4 memory standard, we select HBM and DDR3 to model a wider asymmetry between the upper and lower tiers. We also note that the proposed techniques are not specific to a particular architecture or devices, but can be applied to any memory system with multiple tiers of capacity and performance.

Table 5.1: DRAM standards (taken from [Kim et al.]).

Standard	MT/s	CL-RCD-RP	Data-bus	GB/s
HBM	1,000	7-7-7	128-bit × 1	119.2
DDR3	1,600	11-11-11	64-bit × 1	11.9

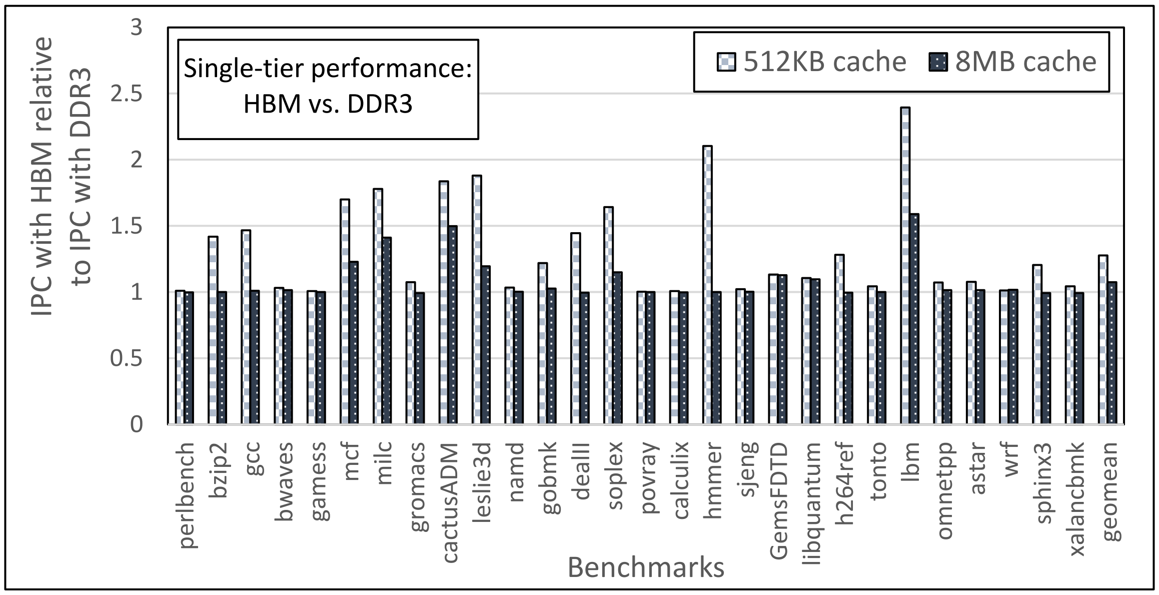

Figure 5.1: Application performance with HBM compared to performance with DDR3.

5.2 Benchmarks and Experimental Design

Our evaluation employs the standard SPEC cpu2006 benchmark suite [23]. Memory usage guidance is collected using both the *train* and *ref* inputs, and all evaluation is performed using the *ref* input. In cases where the benchmark-input pair requires multiple invocations of the application, we conduct independent experiments for each invocation and aggregate the results to compute a single score for each benchmark.

In order to accurately capture the effect of the guidance-based strategies on allocation behavior and heap layout, each experiment executes the entire program run from start to finish. However, detailed cache and memory simulations are limited to only a representative portion of the run using Simpoints [21]. The experiments in this section and Section 6.1 simulate up to 10 program phases with 1B instructions per phase for each program run. Later experiments simulate a much larger portion of each program run using an alternate experimental setup, which we describe in Section 6.3.

Table 5.2: Benchmark statistics. From left to right, the columns show: the benchmark name, native execution time (in seconds), peak resident set size (in MB), # of unique allocation sites reached during the run, # of allocation requests, and for both the 512 KB and 8 MB cache configurations: instructions per cycle (IPC) of the DDR3-only configuration, and last level cache misses per thousand instructions (LLCPKI).

Benchmark	Time (s)	MB	# Sites	# Mallocs	512 KB cache		8 MB cache	
					IPC	LLCPKI	IPC	LLCPKI
bzip2	736	853	10	174	1.751	15.43	-	-
gcc	716	901	19,671	28,458,625	1.872	32.18	-	-
mcf	1,086	1,683	5	6	0.596	95.26	1.076	46.17
milc	739	711	56	6,522	0.773	47.77	1.154	23.72
gromacs	752	24	177	23,796	3.132	3.16	-	-
cactusADM	1,292	668	5,345	131,556	1.172	15.46	1.696	5.07
leslie3d	972	146	101	306,119	0.479	65.23	0.945	22.59
gobmk	659	39	175	658,039	2.902	4.27	-	-
dealII	831	2,279	1,704	151,259,191	2.640	7.45	-	-
soplex	717	604	363	310,613	0.675	57.30	1.400	22.07
hmmer	1,043	45	188	2,474,270	0.657	46.31	-	-
GemsFDTD	1,728	884	509	745,779	0.580	31.42	0.645	17.26
libquantum	1,156	105	10	180	1.761	40.95	1.878	29.06
h264ref	1,136	83	260	177,782	2.748	7.39	-	-
lbm	806	415	4	5	0.678	66.72	1.585	38.75
omnetpp	830	398	3,308	267,064,937	3.101	3.11	-	-
astar	654	361	184	4,799,957	2.294	12.51	-	-
sphinx3	1,080	72	281	14,224,559	1.089	18.18	-	-
average	940	571	1,797	25,252,137	1.605	31.67	1.297	25.59

Our study includes two processor cache configurations to estimate the impact of each hybrid memory strategy in different computing domains: 1) a single-level, 512 KB, 32-way cache, which would be suitable for embedded devices, and 2) a two-level cache with 32 KB, 32-way L1, and an 8MB, 16-way L2, which is more typical for desktop and server machines. Using these configurations along with the default homogeneous memory model included with Ramulator, we conducted an initial set of experiments to identify applications where efficient utilization of the upper-level memory can have a significant impact on program performance. Figure 5.1 shows the performance, in instructions per cycle (IPC), of each benchmark-cache pair with a single tier of (unlimited capacity) HBM relative to the performance of a single-tier DDR3 configuration.[1] Thus, for many applications, there is significant potential to improve performance with HBM. Not surprisingly, the smaller cache configuration exhibits more substantial improvements with the high-bandwidth device due to its increased memory traffic.

For the remainder of this work, we focus our evaluation on the 18 (of 28) benchmarks that exhibit more than 5% performance improvement with HBM relative to DDR3 with the

[1]We omit *zeusmp* from our study because of an incompatibility with our adopted basic block vector collection tool [55, 54].

small cache configuration, as well as the 8 benchmarks that show similar improvements with the larger cache. Table 5.2 lists each of our selected benchmark applications along with relevant statistical information for each application. Thus, the selected benchmarks exhibit a wide range of computational and memory usage requirements.

Chapter 6

Evaluation

In this chapter, we present detailed evaluation of several guidance-based management strategies for hybrid memory systems.

6.1 Performance Potential of Application Guidance for Hybrid Memory Systems

6.1.1 Baseline Policy: Static First Touch

To provide a baseline for comparison of our hybrid memory strategies, we implement the *first touch* policy derived from the NUMA allocation strategy and recently proposed for use with hybrid memory systems [50]. The first touch policy attempts to utilize all of the capacity in the upper-level memory (i.e. HBM) before mapping any data to the lower tier (i.e. DDR3). When the application accesses a new page, the operating system simply maps it to the upper-level memory if there is space available. Otherwise, if the upper-level memory is full, the new page is mapped to the lower tier. Our *static* version of the first touch policy keeps each page in its original tier until the end of the run or unless it is explicitly released by the application (e.g. using the *munmap* system call). Similar to our own guidance-based strategies, first touch can be implemented in commodity systems and does not require any architectural modifications or non-standard hardware.

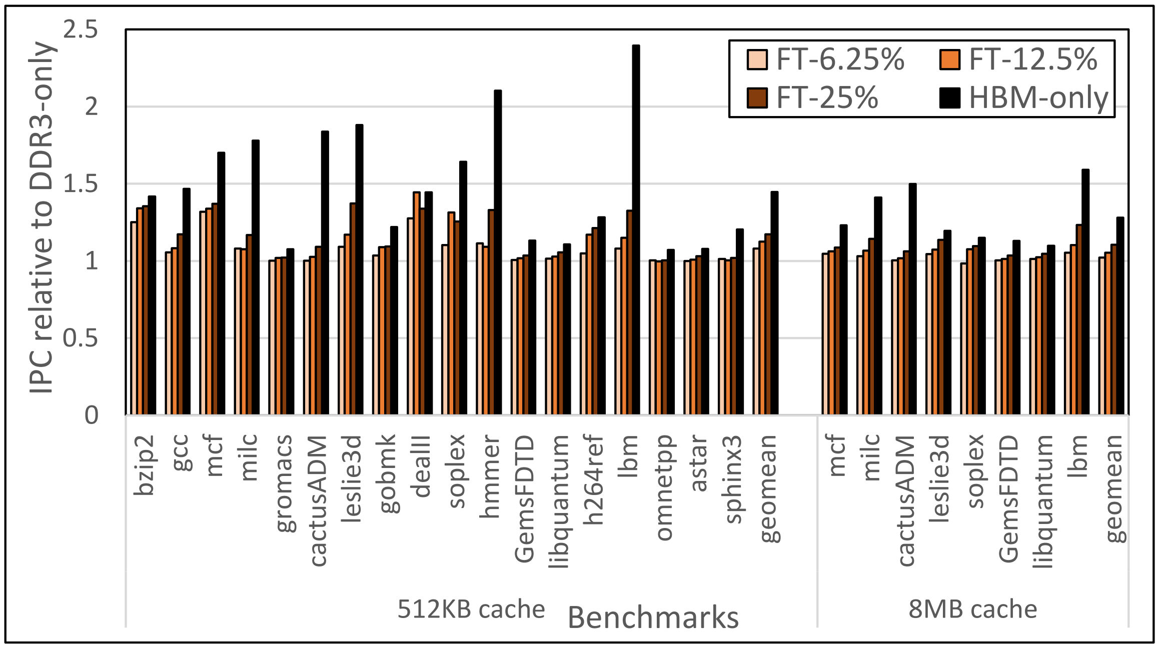

Figure 6.1: Performance with static first touch policy on hybrid memory architecture with varying upper-level memory capacity relative to DDR3-only.

We evaluated the static first touch policy with three different capacity limitations on the size of the upper-level memory: 6.25%, 12.5%, and 25% of the benchmark's peak RSS. Figure 6.1 shows the performance (in IPC) of the static first touch policy with different upper-level capacities alongside a configuration with unlimited HBM capacity (HBM-only). The results are shown relative to a configuration that uses a single tier of DDR3 memory (DDR3-only). Thus, we see that the HBM significantly improves the performance of the selected benchmarks. Compared to DDR3-only, the HBM-only configuration achieves a 44.6% average speedup with the smaller (512 KB) cache, and a 28.2% average speedup with the larger (8 MB) cache. However, when the upper-level capacity is reduced, the static first touch policy is not able to achieve all or even most of the performance benefits of the HBM. For instance, with an HBM tier capacity of 12.5%, the first touch policy yields average speedups of only 12.5% with the smaller cache and only 5.5% with the larger cache, compared to DDR3-only. The rest of the results in this section use the static first touch policy with the upper-level memory limited to 12.5% of total space requirements as the baseline configuration.

6.1.2 Performance Potential of Static Application Guidance for Hybrid Memory Systems

A critical disadvantage of the first touch approach is that a page with relatively cold data may be accessed and mapped to the upper-level tier early in the run, potentially crowding out performance critical data later in the run. For the next set of experiments, we augment the static first touch policy with profile guidance to only map data associated with certain "hot" allocation sites into the upper-level memory. For each benchmark, we profiled the memory usage behavior associated with each allocation site for an entire program run with both the *train* and *ref* inputs. We then computed knapsacks and hotsets for the collected profile information using five different capacities: 3.125%, 6.25%, 12.5%, 25.0%, and 50.0%. Next, we use the resulting sets of allocation site guidance to conduct experiments with the static guidance allocation strategy. During each experimental run, the memory manager attempts to map pages in the hot arena to the upper-level memory on a first touch basis, while data in the cold arena is always mapped to the lower tier. The experimental runs use a fixed upper level capacity (12.5% of the total space needed), independent of the size of the knapsack or hotset used to construct the guidance, for comparisons against the static first touch policy with the same HBM capacity.

Figure 6.2 shows the average performance (IPC) of each static guidance strategy over all of the benchmarks relative to the static first touch approach. These results allow us to make several interesting observations: 1) Each static guidance strategy significantly outperforms the unguided first touch, 2) While guidance collected using the same (*ref*) program input as the evaluation run tends to perform better than that collected using the different (*train*) input, the difference is relatively small for most of the benchmarks. 3) The size of the knapsack or hotset has a significant impact on program performance. If the knapsack size is too small, this strategy may not utilize the available HBM capacity. Alternatively, if the hotset size is much larger than the actual upper-level capacity, relatively cold data from the hot arena that happens to be touched earlier may crowd out critical data accessed later in the run. 4) The best hotset strategies consistently outperform the best knapsack strategies. This outcome confirms our intuition that being too conservative in cases where an allocation

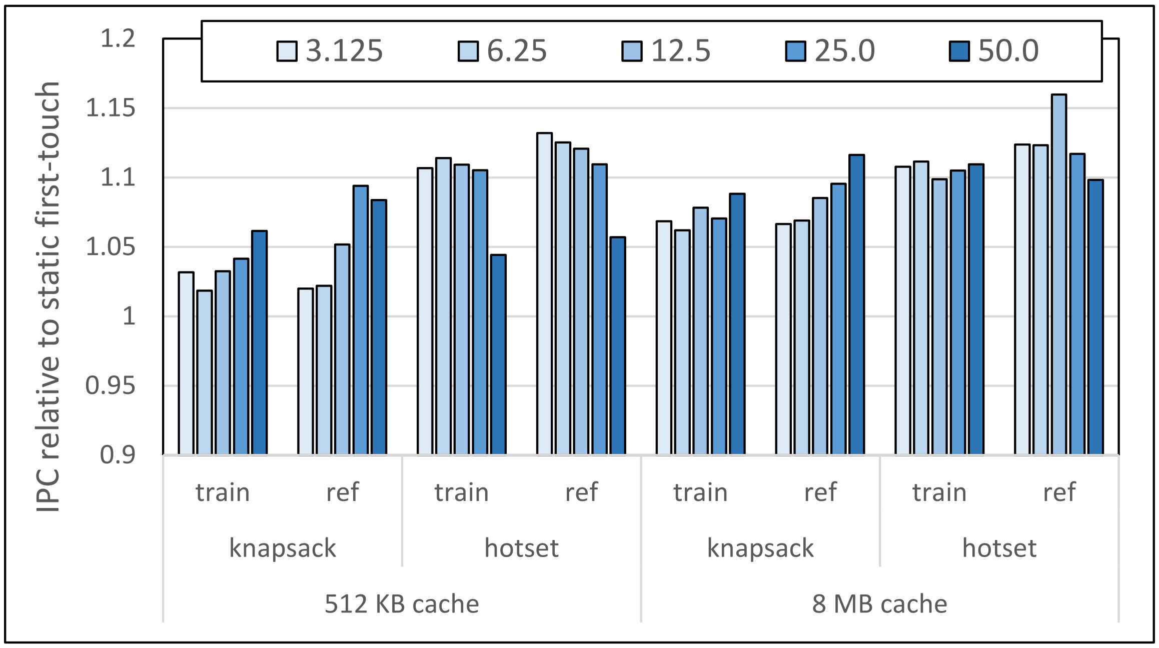

Figure 6.2: Average performance (IPC) improvement of static allocation site guidance with varying knapsack and hotset capacities relative to static first touch.

site with very hot data does not fit entirely in the upper tier is less effective than allowing a portion of the site's data to map to the faster device. Due to this finding, we focus the studies in the remainder of this section only on the hotset guidance-based strategies.

Figure 6.3 shows the performance (IPC) of the benchmarks with the best static hotset guidance approaches relative to static first touch. On average, the best *train-hotset* improves performance by 11.4% and 11.1% compared to first touch with the small and large cache configurations, respectively, while the best *ref-hotset* achieves average performance gains of 13.2% and 16.0%, respectively. Surprisingly, some individual benchmarks, such as *hmmer* and *leslie3d* (with the 8MB cache) perform even better on the hybrid memory system with static guidance than on an HBM-only configuration. One possible reason for this result is that the lower (DDR3) tier provides some additional bandwidth in the hybrid memory configuration, and thus, enables more overall system bandwidth than the single-tier HBM configuration. Agarwal et al. [1] had recognized this phenomenon and proposed policies to exploit the additional bandwidth in the lower memory tiers. Another likely contributor is that altering the heap layout by separating hot and cold objects into distinct arenas improves

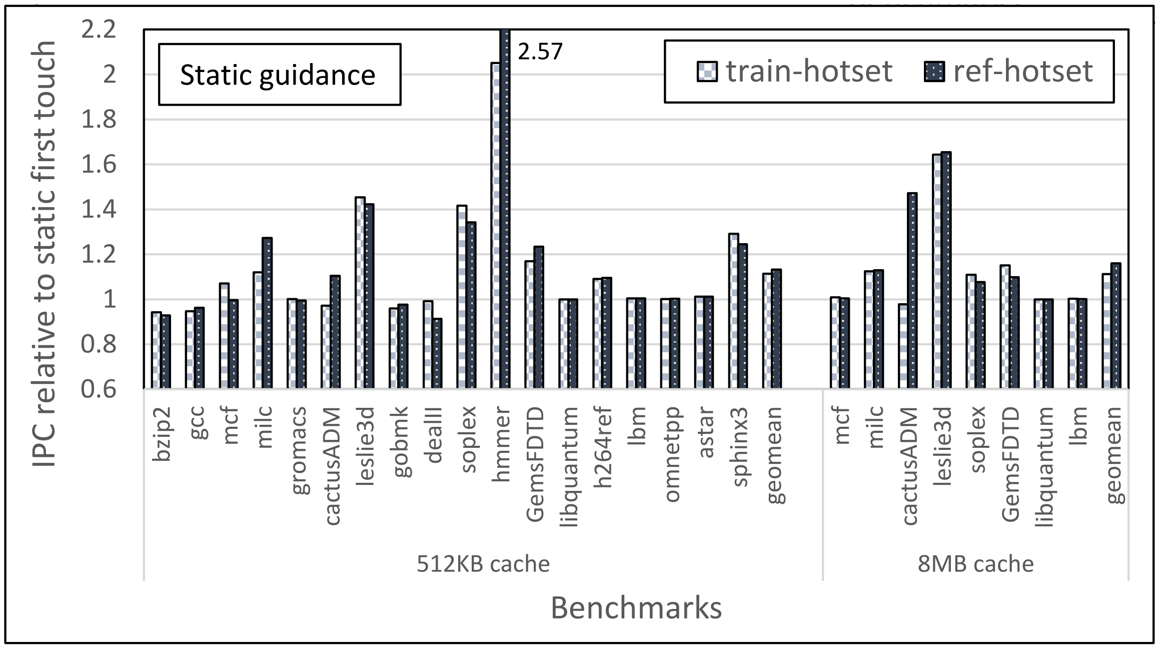

Figure 6.3: Performance (IPC) with best static hotset guidance relative to static first touch.

data locality and increases efficiency for some benchmarks. We investigate this possibility and explore its potential benefits in Section 6.4.

6.1.3 Performance Potential of Per-Phase Application Guidance for Hybrid Memory Systems

The experiments in the previous section generate and apply the same application guidance over the entire benchmark execution, without regard to potentially different usage behavior in each program phase. This section aims to evaluate the potential of using guidance tailored for each program phase to remap application data at phase change boundaries. For each benchmark, we collected memory usage profiles of each program phase with the *ref* evaluation input and constructed per-phase allocation site hotsets with varying capacities (3.125%, 6.25%, 12.5%, 25.0%, and 50.0%). We then ran each benchmark with the original *train* hotsets using the static arena allocation strategy (as in Figure 6.3) as well as with each set of per-phase *ref* hotsets using the per-phase arena allocation strategy. In these runs, we configure the hybrid memory simulator to unmap all the live data in each tier at

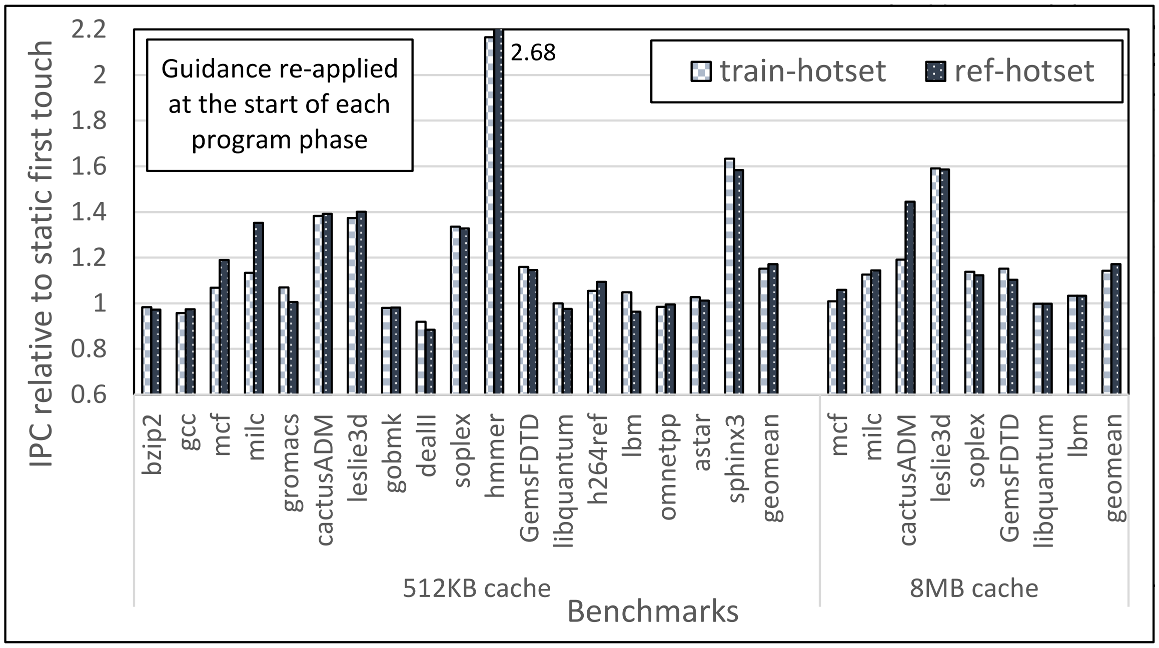

Figure 6.4: Performance (IPC) with allocation site guidance re-applied at the beginning of each program phase.

the start of each program phase. Then, as pages are accessed in each phase, the manager remaps the pages to the appropriate tier according to the per-phase guidance. Similar to our other experiments, all configurations use the same upper-level capacity of 12.5%, regardless of the hotset capacity. Since our intention is to determine the potential benefits of applying per-phase guidance to direct page placement, these experiments ignore remapping penalties.

Figure 6.4 shows the performance (IPC) of the best train-hotset and the best per-phase ref-hotset configurations with guidance re-applied at the start of each program phase relative to the static first touch configuration. Comparing these results to Figure 6.3 allows us to isolate the impact of re-applying guidance at the start of each program phase. Thus, while this approach yields significant improvements for a few benchmarks, such as *mcf*, *cactusADM*, and *sphinx3*, the rest experience little or no performance benefits. The best train-hotset with remapping at the start of each phase improved by 3.4% (512 KB cache) and 2.8% (8 MB cache), on average, over the best train-hotset with static placement. Similarly, the best per-phase ref-hotset improves by 3.5% and 1.0%, on average, compared to the best static ref-hotset. These results indicate that the static policies are able to capture most of the performance critical data in the upper tier without requiring any remapping or data migration.

6.2 Assessing the Overhead of Allocation Site Identification and Partitioning

In this section, we run our allocation site identification system outside of the simulation components to measure the impact on program performance our framework imposes. For these experiments, we generate program guidance, and partition the allocation sites into arenas, but on a real system without heterogeneous memory, so that we can isolate the effect of framework overhead. These experiments were run using the same servers as our simulation experiments.

6.2.1 Time Overhead of Arena Allocation

First, we evaluate the impact of using the `backtrace` and `backtrace_symbols` system calls to identify allocation sites, as well as the extra time spent associating those allocation sites with a particular arena. To measure this overhead, we utilize three configurations. The 'Guided' configuration identifies the allocation site, and provides feedback with regards to which arena that site should be allocated. Next, the 'Ignored' configuration computes the same information but then the arena placement recommendations are ignored and all pages are placed in a single arena. Finally, a baseline configuration which disables the allocation site identification facility entirely in favor of placing all pages in a single arena is used to evaluate the other two configurations.

Figure 6.5 shows the total run time (averaged over five iterations) for each benchmark, relative to the baseline configuration where the allocation site identification and partitioning system is deactivated. The same guidance information was used in all three configurations. We find that, although allocation site identification incurs very little overhead for most programs, some benchmarks, such as GCC and Omnetpp, incur substantial performance

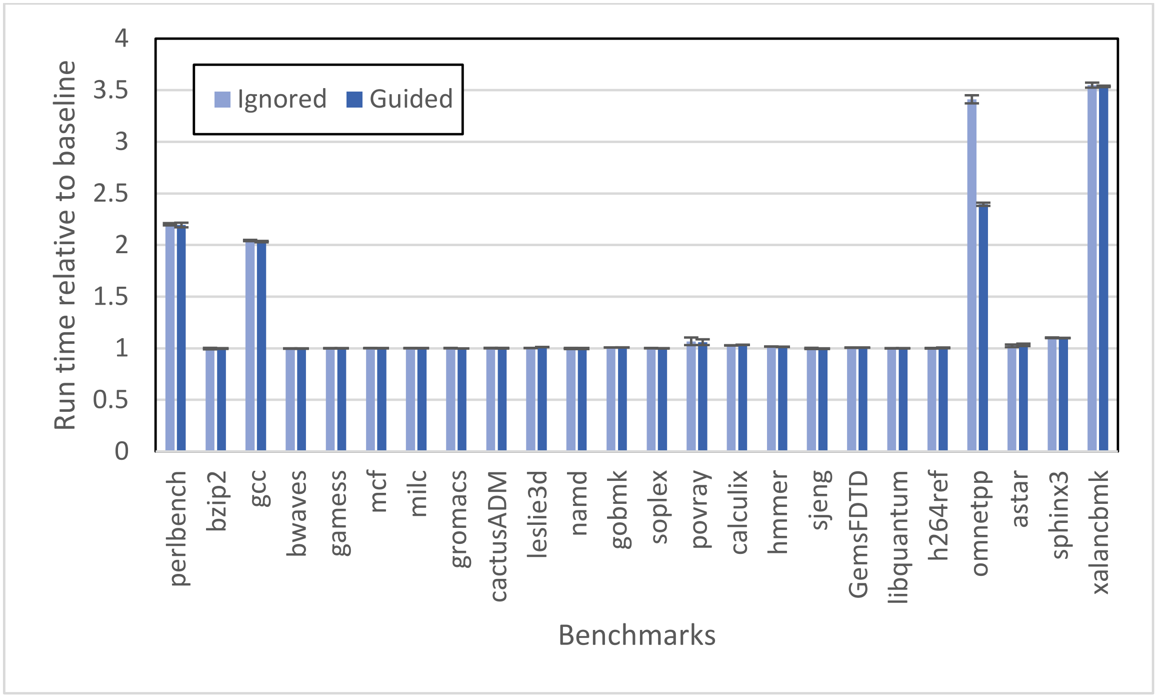

Figure 6.5: Average application run time presented relative to a baseline where allocation site detection and guidance is disabled.

losses. Not surprisingly, these losses are directly proportional to the number of allocation requests during execution, which are shown in Table 5.2.

Strong similarities between the two configurations for most benchmarks indicate that most if not all of the overhead for many applications could be mitigated by replacing our system-call heavy approach with compiler-based methods like method-cloning to identify the allocation sites ahead of time. Interestingly, in the case of Omnetpp especially, differences between the Guided and Ignored configurations can be observed. In these cases we were able to determine that the partitioning of objects into arenas based on their usage patterns is able to affect the efficiency of the DRAM Row Buffer. This result is discussed in detail in Section 6.4.

6.2.2 Space Overhead of Arena Allocation

The results from a previous section shows that some applications may benefit from per-phase specialization of guidance, however, partitioning the allocation sites into multiple arenas to support such a strategy may carry a potential space overhead. In a program with n phases, up to 2^n distinct arenas may be required in the worst case, which could significantly increase fragmentation and bloat the capacity requirements.

Figure 6.6 shows the peak resident set size for each benchmark with the per-phase ref-hotset guidance with 12.5% capacity for up to 10 program phases. Additionally, markers plotted on the right y-axis show the number of arenas created by each benchmark with the per-phase guidance. The space results are shown relative to a configuration that uses the default jemalloc allocator with a single arena.

Thus, the number of arenas required by our profile-guided arena allocation scheme is typically much less than the worst case scenario. However, this scheme does increase capacity requirements for some benchmarks. In most cases, these increases are relatively small, and on average, the per-phase guidance requires less than 5% additional space. One benchmark, *omnetpp*, actually decreases its space requirements because the alternative heap layout reduces the number of pages that are actually touched during the run. Not surprisingly, the static guidance configurations that use only one arena for each memory tier require less additional space overhead than the per-phase guidance configurations. The static guidance schemes only require 1.4% to 1.7% additional memory capacity compared to the default configuration, on average.

6.3 Comparing Application Guidance to State-of-the-Art Hybrid Memory Strategies

To evaluate our application guidance-based strategies in the context of other state-of-the-art hybrid memory strategies, we adopt and implement the work of Meswani et al. [50]. Their work proposes an OS/architectural strategy that relies on per-page access counters in the hardware to identify physical pages that should be promoted to the upper-level memory. Since Meswani et al.'s approach adaptively migrates data into and out of the upper-level memory, we need to update our framework to accurately model the the cost of data migration. For this purpose, we extended our simulator to include penalties for page faults, TLB shootdowns, and page migration as described in [50]. Page faults and TLB

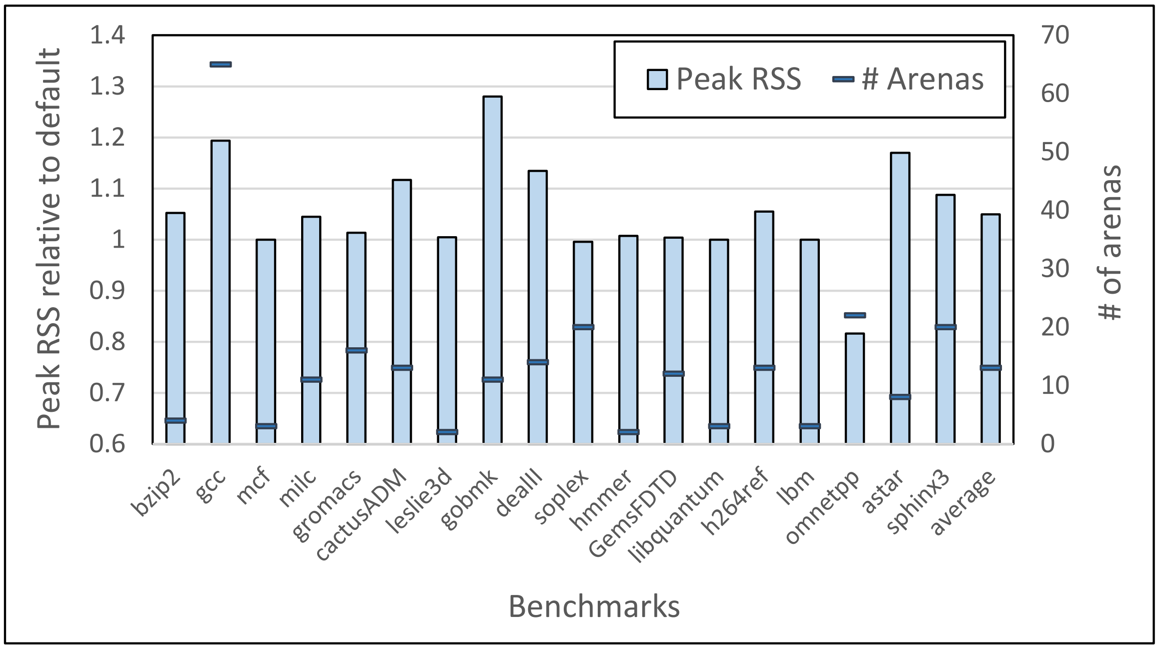

Figure 6.6: Peak resident set size and number of arenas created with per-phase allocation site guidance.

shootdowns incur fixed penalties of 5μs and 3μs, respectively, whenever one is required. To model page migration, we simply insert the necessary memory read and write instructions into the Ramulator instruction stream. Additionally, rather than simulate up to 10 non-consecutive slices of program execution, the experiments in this section simulate a single, large, contiguous slice of 64B program instructions. With Ramulator's execution model, this amount of instructions corresponds to at least 5 full seconds of benchmark execution time (measured in CPU cycles), and a typical execution time of 20 to 30 seconds.

To illustrate how data migration costs can impact performance, let us consider an adaptive variant of the static first touch approach. The adaptive first touch policy aims to keep more frequently accessed data in the upper-level memory by periodically invalidating all of the pages in the application's page tables. Then, as pages are accessed, the system will fault the invalid pages into the upper-level memory on a first touch basis until it is full. If the faulting page is currently on the lower tier, it will need to be migrated in to the upper tier, and may potentially require the selection of a victim page to migrate out to the lower tier. Any subsequent accesses to the page will proceed as normal (i.e. without a page fault) until it is invalidated again at the start of the next interval.

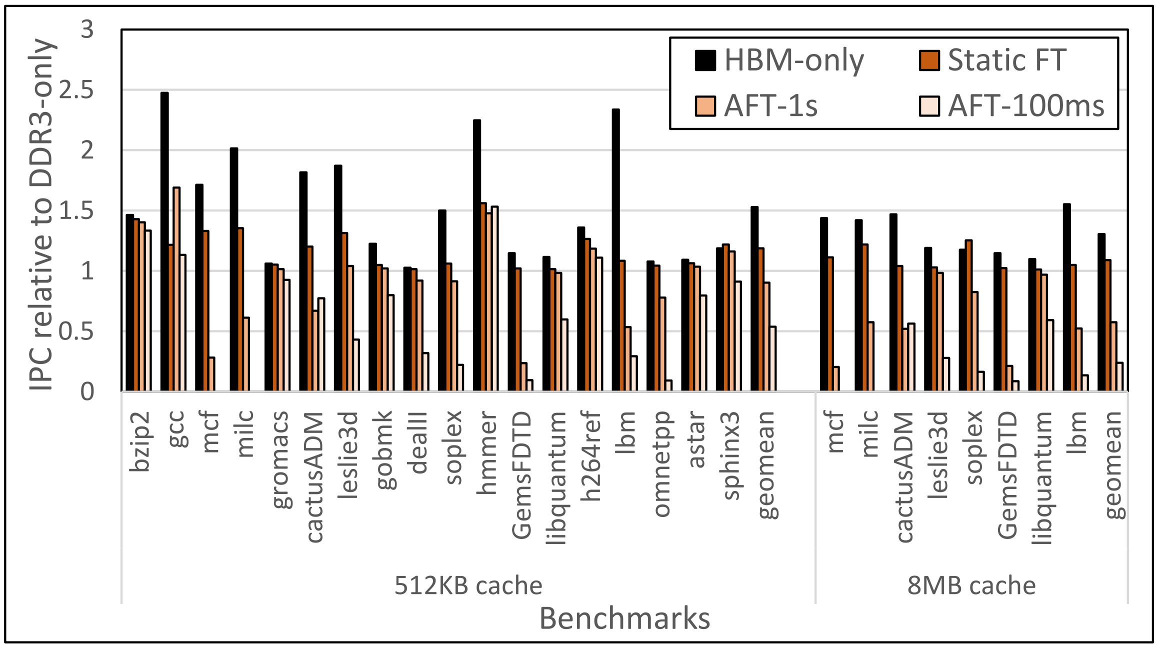

Figure 6.7: Performance (IPC) with HBM-only and static first touch and adaptive first touch policies.

Using our extended experimental framework, we evaluated the original static first touch policy as well as the adaptive first touch policy with interval lengths of 1s and 100ms on an HBM-DDR3 hybrid memory system with an HBM capacity of 12.5%. Figure 6.7 shows the performance (IPC) of the HBM-only, static first touch, and adaptive first touch configurations relative to the performance with only a single tier of DDR3 memory.[1] In most cases, the static first touch policy clearly outperforms the adaptive first touch policy with either interval configuration. The reason is because the adaptive first touch configurations incur about 1 to 2 orders of magnitude more page faults than the static first touch configuration.On average, the adaptive first touch strategies with 1s and 100ms epochs generate about 12x and 70x more page faults, respectively, than the static first touch approach (which requires about 140,000 faults, on average, for each benchmark). Thus, any potential performance advantages that the adaptive first touch policies may have gained by mapping more frequently accessed data into the faster memory tier are lost due to the overhead of the page faults.

[1]For most benchmarks, simulating the adaptive first touch policy with 100ms intervals required a few hours to several days of compute time. *mcf* and *milc* did not complete with this configuration even after 10 days of compute time, so we omit these from our results.

6.3.1 Reactive Profiling to Reduce Migration Costs

Meswani et al. proposed an epoch-based, reactive profiling approach that aims to map frequently accessed data into the faster memory tier, while also limiting the overheads of data migration. It relies on hardware counters to keep track of the number of reads and writes to each physical memory page. At the start of each epoch, the OS selects the pages to promote to the upper-level memory based on the values of the counters, and then performs the necessary data migration. Before resuming the application, the OS resets the access counters, and validates all of the process's page table entries to avoid incurring unnecessary faults between epoch boundaries.

We implement two variations of the above approach to compare with our guidance-based strategies. The first, named *history*, sorts all of the pages by their access counts at the end of each epoch, and then selects the top n pages responsible for the most memory traffic to place in the faster memory tier, where n is the capacity of the fast memory. While the history approach always ensures accurate partitioning of program data, it requires invasive architectural changes that are not likely to be adopted in modern hardware. The second variation, named *first touch hot page (FTHP)*, is more feasible. In this approach, rather than maintain per-page access counts, the hardware sets a "hot bit" on each page when it detects the page has been accessed more than some threshold number of times. Pages with their hot bit set then become eligible for placement in the upper tier at the next epoch boundary. [2] We evaluated both the history and FTHP policies with epoch lengths of 1s and 100ms on the HBM-DDR3 configuration with 12.5% capacity in the HBM tier. For FTHP, we selected an initial threshold value of 32 and used the dynamic threshold adjustment strategy described in [50]. Figure 6.8 shows the performance (IPC) of each policy using 100ms epoch lengths relative to the static first touch approach. To simplify presentation, we omit detailed results for the 1s epoch length, except to note that we found very little performance difference with the longer epoch lengths for either policy.[3] Thus, the reactive profiling-based strategies achieve significant speedups over static first touch. Not surprisingly, the history

[2] Meswani et al. propose adding access counters to the TLB to detect when a page has reached the hotness threshold. Since our simulation environment does not include a TLB, we simply approximate the FTHP policy using per-page access counts.

[3] On average, the longer epoch length improves history by 0.6%, and degrades FTHP by 0.3%.

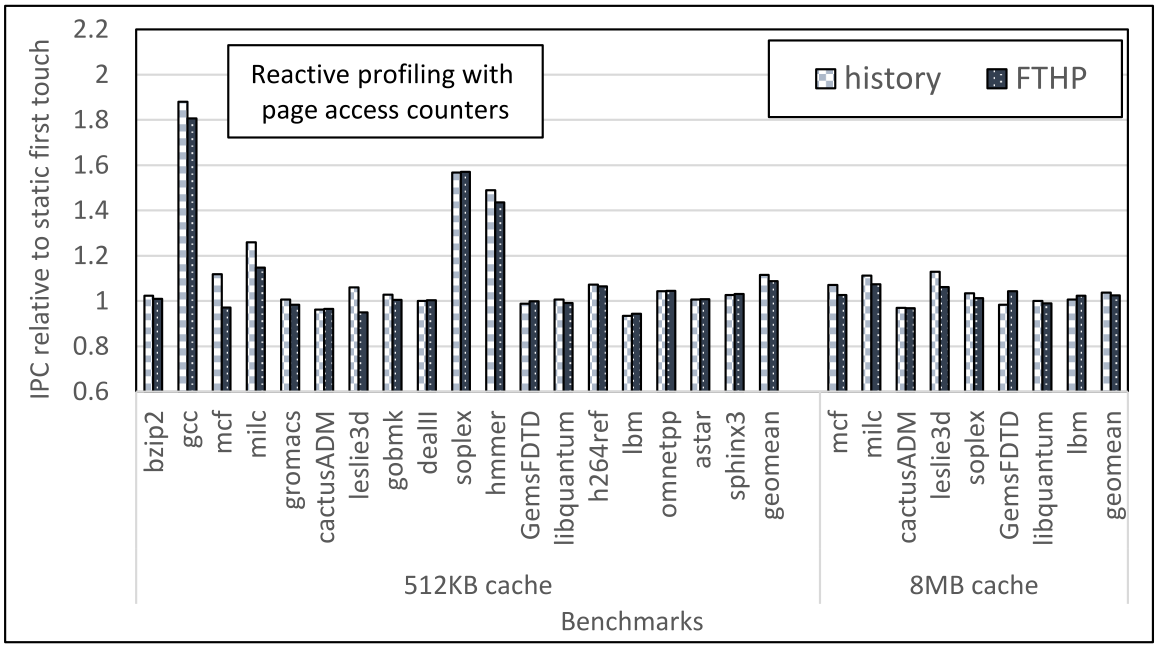

Figure 6.8: Performance (IPC) with reactive profiling strategies from [50] relative to static first touch.

policy outperforms FTHP by 2.8% with the smaller cache and 1.2% with the larger cache, on average. In most cases, FTHP is able to achieve similar or only slightly worse performance than the more accurate history approach.

6.3.2 Comparison of Application Guidance and Reactive Profiling Approaches

Our next set of experiments directly compares our application guidance-based strategies to the OS/architectural approach described in the previous subsection.For this comparison, we prepared two configurations to use with our extended hybrid memory framework: *train-hotset* and *ref-hotset*. Similar to our experiments in Section 6.1, the train-hotset configuration collects memory usage profiling during a separate *train* input run, and uses it to apply static hotset-based guidance during the evaluation run. The ref-hotset configuration computes program phase classifications offline, and adaptively applies guidance tailored to each program phase. As with our earlier per-phase guidance experiments, the ref-hotset configuration profiles the memory behavior of each phase using the *ref* input, computes hotsets for each phase, and uses the hotsets during the evaluation run to apply the per-phase

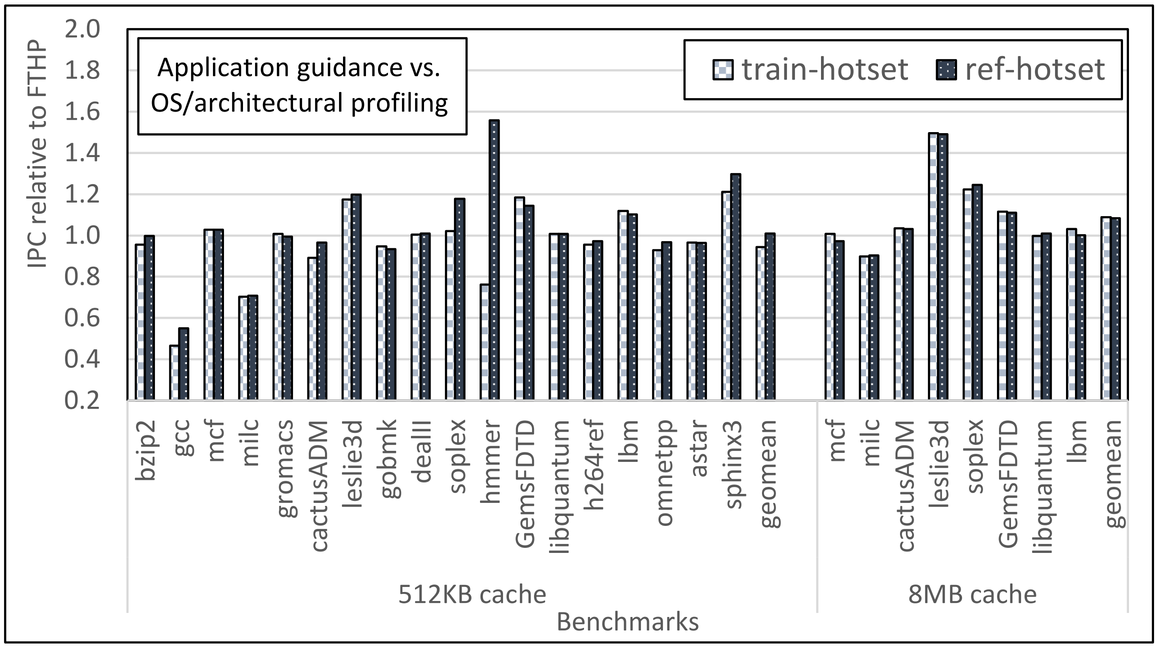

Figure 6.9: Performance (IPC) with static and adaptive allocation site guidance on HBM-DDR3 hybrid memory relative to [50]'s reactive profiling approach.

arena allocation strategy. Additionally, we use Simpoints to classify each 1B instruction slice in the evaluation (*ref* input) run into one of up to 10 program phases. Then, for these longer simulations (of 64B instructions), we configure our framework to detect and adjust arena guidance when the application enters a new program phase. At each phase change boundary, the memory manager interrupts the application, and attempts to migrate data to the appropriate tier using the guidance for the next program phase. In this way, the ref-hotset configuration allows the system to *proactively* adjust its memory map according to expected usage behavior.

For our comparisons, we select the FTHP policy (with 100ms epoch) as our baseline since it is more feasible to implement in modern hardware than the history approach. Figure 6.9 shows the performance (IPC) of the static train-hotset and adaptive ref-hotset configurations relative to FTHP.While there are some benchmarks, such as *gcc* and *milc*, where train-hotset and ref-hotset are not as effective as FTHP, in most cases, the guidance-based approaches either meet or exceed the performance of FTHP. On average, the train-hotset approach with the small cache configuration is 5.7% slower than FTHP, while the ref-hotset approach is

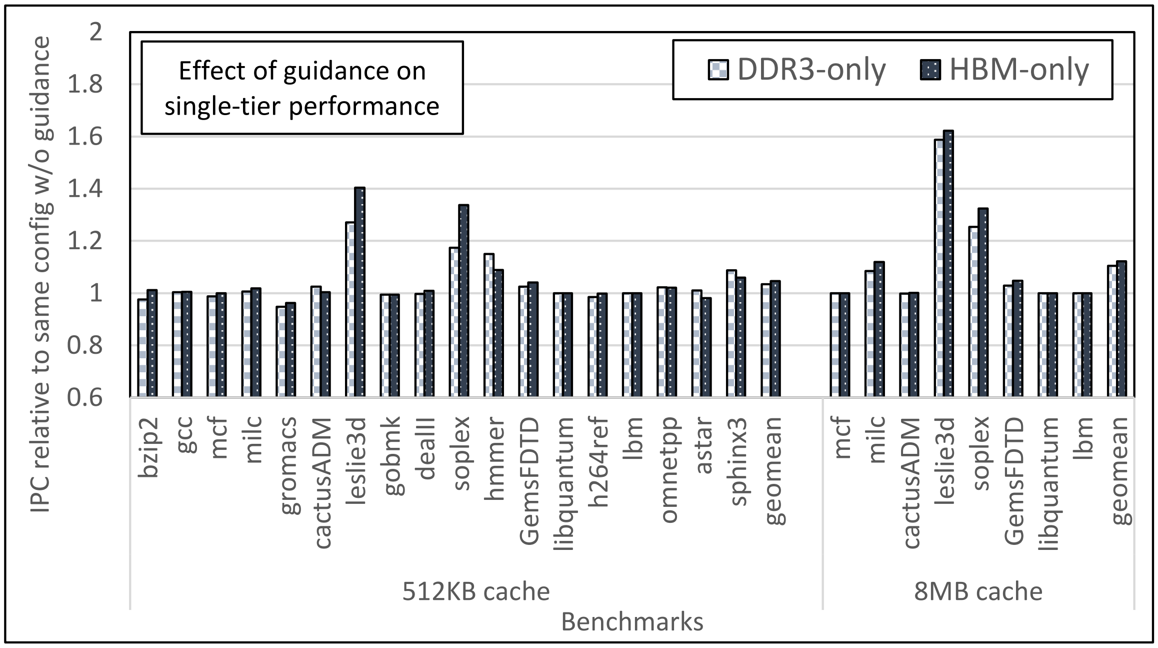

Figure 6.10: Performance (IPC) with guidance relative to the same single-tier configuration with no guidance.

about 1.0% faster. With the larger cache, both the train-hotset and the ref-hotset outperform FTHP by 8.8% and 8.3%, respectively.

We also find that our approach attains this high level of performance with relatively little data migration. Table 6.1 shows the total amount of data migration (in GB) between the upper and lower level memory tiers alongside the total number of migration intervals (epoch ticks or phase changes) for each adaptive management strategy. Thus, despite reaching migration intervals about 3x more often, the ref-hotset approach migrates only about 10% more data than the FTHP approach with a 1s epoch. We plan to explore and optimize the relationship between phase length, data migration, and program performance in future work.

6.4 Impact of Guidance on DRAM Efficiency

Interestingly, as with our experiments in Section 6.1, some benchmarks exhibit even better performance with the static and adaptive guidance-based approaches on the hybrid memory configuration than on a single, all-HBM tier. One possible reason for this outcome is that co-locating hot objects in the same arena potentially increases data locality – either raising

Benchmark	FTHP-1s		FTHP-100ms		ref-hotset	
	GB	MI	GB	MI	GB	MI
bzip2	0.98	55	2.54	581	0.85	252
gcc	3.67	95	20.96	821	3.47	180
mcf	6.34	33	65.03	382	0.00	11
milc	0.42	16	2.54	181	0.37	6
gromacs	0.00	6	0.05	61	0.02	59
cactusADM	0.66	14	12.78	153	0.52	63
leslie3d	0.22	32	2.60	334	0.00	63
gobmk	0.04	27	0.78	294	0.02	183
dealII	0.10	5	0.63	59	0.04	15
soplex	0.10	27	1.58	262	0.01	32
hmmer	0.01	30	0.18	261	0.00	105
GemsFDTD	2.83	35	59.05	375	7.96	62
libquantum	0.02	11	0.02	114	0.00	51
h264ref	0.02	17	0.28	175	0.02	112
lbm	0.51	27	23.24	293	4.62	50
omnetpp	0.11	6	1.20	63	0.22	36
astar	0.00	14	0.95	147	0.07	35
sphinx3	0.02	15	0.15	155	0.04	21
average	0.89	25	10.81	261	1.01	74

Table 6.1: Data migration (in GB) and number of migration intervals (epoch ticks or phase changes) with adaptive hybrid memory strategies (512 KB cache).

cache hit rates, or making memory accesses more efficient, or both. Upon investigation, we were somewhat surprised to find that guidance-based arena allocation strategies have almost no impact on the hit rate of the processor caches. However, in many cases, our approach significantly increases the hit rate for the DRAM row buffer. Intuitively, this result makes sense because packing hot program objects together in an arena will increase the likelihood of consecutive accesses to the same memory page, and hence, to the same row buffer.

This finding indicates that our guidance-based approach may not only benefit hybrid memory configurations, but also any memory architecture that relies on efficient row buffer utilization to achieve low latencies. To evaluate the potential of our approach for standard memory architectures, we setup experiments to use the guidance-based arena allocation strategy on the single-tier DDR3-only and HBM-only configurations. Figure 6.10 shows the performance of the per-phase arena allocation strategy with ref-hotset guidance applied with the DDR3-only and HBM-only configurations. For each configuration, the results are shown relative to the same single-tier configuration with no guidance.

Hence, the guidance-based arena allocation strategy significantly improves performance even for the standard single-tier memory architecture. In both configurations, 4 (out of 18) benchmarks yield $> 5\%$ speedups with the smaller cache, while 3 (of 8) exhibit similar improvements with the larger cache. Only 1 benchmark, *gromacs*, incurs notable performance

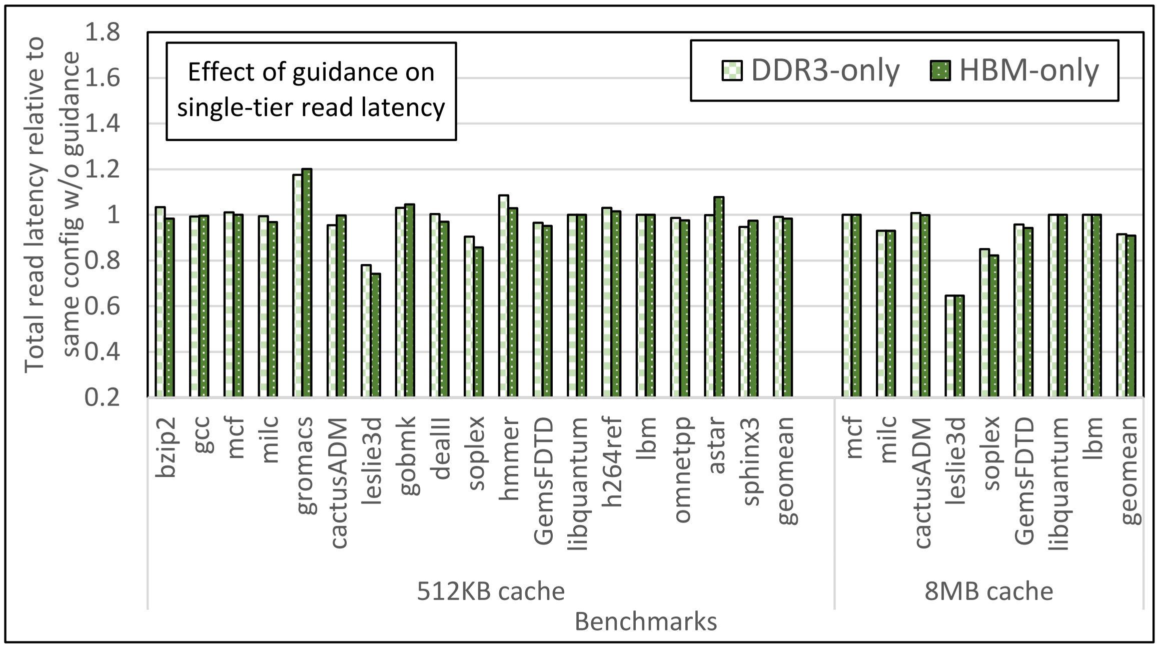

Figure 6.11: Total read latency with application guidance relative to the same configuration with no guidance.

losses (of 5.2% with DDR3 and 3.8% with HBM). As expected, these performance effects coincide with significant differences in the observed row buffer efficiency and memory access latency, which is shown in Figure 6.11.

Lastly, we investigate the potential benefits of combining our arena allocation strategies with the OS/architectural techniques described in Section 6.3.1. In addition to reducing latencies, application-guided arena allocation may also concentrate memory traffic to a smaller portion of application pages, enabling the reactive approach to steer traffic to the upper-level with greater efficiency. Figure 6.12 shows the performance of the history and FTHP policies with application-guided arena allocation on an HBM-DDR3 configuration relative to the same configuration with no allocation guidance. Thus, the combined approaches significantly improve performance over the unmodified OS/architectural-based strategies. For most benchmarks, these improvements are driven by the same reductions in memory latency that we had seen with the single tier configurations. In some cases, especially with the FTHP policy, the proportion of traffic to the upper-level memory also increases. On average, combining application guidance with the FTHP policy achieves speedups of 6% and 15.4%, with the 512 KB and 8 MB cache, respectively, over FTHP alone, and speedups

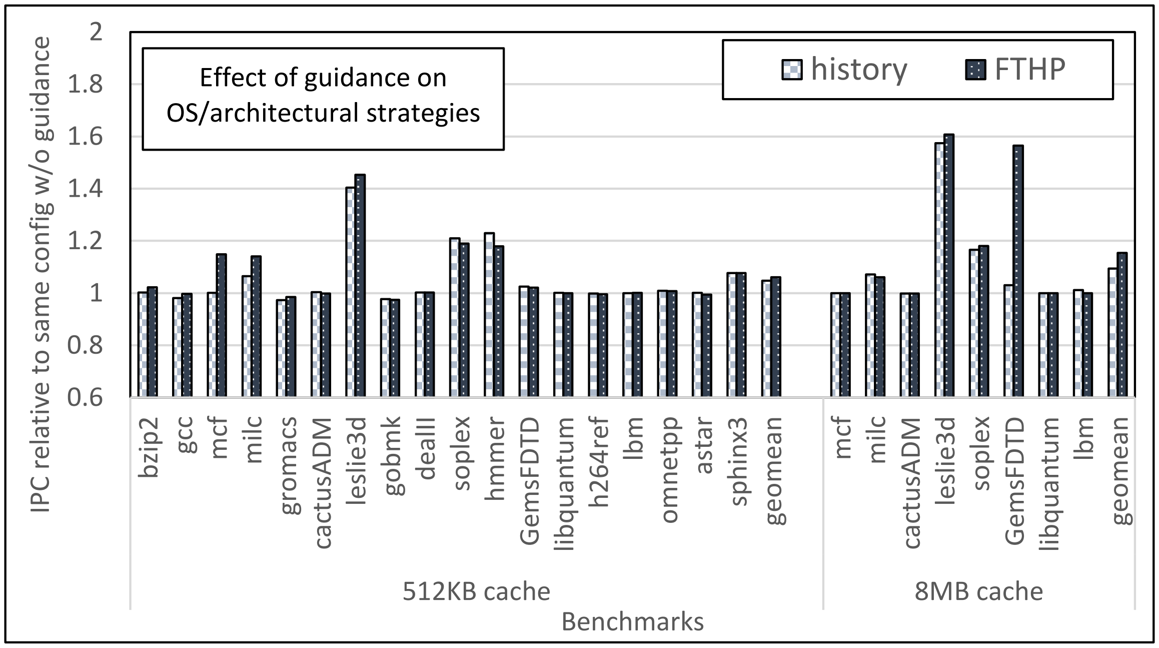

Figure 6.12: Performance (IPC) of OS/architectural strategies with allocation site guidance relative to the same configuration with no allocation site guidance.

of 5% and 7.9% compared to the best adaptive ref-hotset strategy without reactive profiling.

Chapter 7

Future Work

This work lays the foundation for a wide range of interesting and important research in hybrid memory systems just as a number of promising new memory devices are beginning to hit the market. In the immediate future, we plan to adapt our automated guidance framework for use with real heterogeneous memory hardware, including configurations with both high capacity storage class memories [29] as well as high bandwidth on-package DRAM [12]. Towards this objective, we will integrate online allocation site detection with compiler support to reduce or eliminate the performance overhead of our current technique. We also plan to update our framework with more sophisticated profiling and classification strategies, such as the approach proposed by Dulloor et al. [16], to enable automated tier assignments for program data with different access patterns and latency requirements.

Multiple findings from this study warrant additional investigation. For instance, we found that using a different input for the profile run and tailoring application guidance to each program phase both have a relatively small impact on program performance. Further research is necessary to understand how these factors affect program performance, and whether this result is specific to our selected benchmarks and experimental configuration, or if it reflects a more fundamental property of hybrid memory systems. While investigating these issues, we also plan to explore the feasibility of using pure static analysis, without program profiling, to guide hybrid memory management. Additionally, there is clear potential for profile-guided arena allocation to improve DRAM efficiency by increasing row buffer utilization. We plan to study this effect on real DRAM hardware, and evaluate its impact in single- and multi-core computing environments.

Chapter 8

Conclusion

Emerging memory technologies are forcing systems to alter their memory management strategies to take advantage of the different capabilities and performance provided by the new types of hardware. Current strategies rely on source code modifications and/or architectural changes to map memory traffic and access patterns to heterogeneous memory devices. In this work, we describe the first fully-automatic framework for enabling applications to guide data management across a multi-tier memory hierarchy. Our design is flexible enough to collect and apply application guidance from different program inputs or tailored to specific program phases with no additional recompilation. The evaluation demonstrates that this approach can significantly improve program performance, even compared to a state-of-the-art OS/architectural strategy that uses non-standard hardware. Overall, this work makes significant progress towards meeting the challenges posed by hybrid memories and paves the way for automatic adaptation of application behavior to multi-tier memory architectures.

Bibliography

[1] Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., and Keckler, S. W. (2015). Page placement strategies for gpus within heterogeneous memory systems. *SIGPLAN Not.*, 50(4):607–618. 3, 5, 25

[2] Badam, A. and Pai, V. S. (2011). Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Berkeley, CA, USA. USENIX Association. 5

[3] Beg, M. and van Beek, P. (2010). A graph theoretic approach to cache-conscious placement of data for direct mapped caches. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 113–120, New York, NY, USA. ACM. 5

[4] Ben-Asher, Y. and Rotem, N. (2010). Automatic memory partitioning: Increasing memory parallelism via data structure partitioning. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 155–162, New York, NY, USA. ACM. 5

[5] Calder, B., Krintz, C., John, S., and Austin, T. (1998). Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149. 5

[6] Cantalupo, C., Venkatesan, V., and Hammond, J. R. (2015). User extensible heap manager for heterogeneous memory platforms and mixed memory policies. 3, 5

[7] Carlson, J. L. (2013). *Redis in Action*. Manning Publications Co., Greenwich, CT, USA. 1

[8] Chang, K. K., Kashyap, A., Hassan, H., Ghose, S., Hsieh, K., Lee, D., Li, T., Pekhimenko, G., Khan, S., and Mutlu, O. (2016). Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. *SIGMETRICS Perform. Eval. Rev.*, 44(1):323–336. 1

[9] Cherem, S. and Rugina, R. (2004). Region analysis and transformation for java programs. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 85–96, New York, NY, USA. ACM. 5

[10] Chilimbi, T. M. and Shaham, R. (2006). Cache-conscious coallocation of hot data streams. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 252–262, New York, NY, USA. ACM. 5

[11] Chou, C., Jaleel, A., and Qureshi, M. K. (2014). Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. IEEE Computer Society. 5

[12] Consortium, H. M. C. (2014). Hmc specification 2.1. 40

[13] Dean, J. and Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56:74–80. 1

[14] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P. (2010). Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society. 2, 5

[15] Dr. Michael Jantz (2017). CORSys Research Group. http://web.eecs.utk.edu/ mrjantz/corsys.html. 4

[16] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. (2016). Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM. 3, 5, 40

[17] El-Nacouzi, M., Atta, I., Papadopoulou, M., Zebchuk, J., Jerger, N. E., and Moshovos, A. (2013). A dual grain hit-miss detector for large die-stacked dram caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 89–92. EDA Consortium. 2, 5

[18] Evans, J. (2006). A scalable concurrent malloc (3) implementation for freebsd. 13

[19] Giardino, M., Doshi, K., and Ferri, B. H. (2016). Soft2lm: Application guided heterogeneous memory management. In *IEEE International Conference on Networking, Architecture and Storage (NAS), Long Beach, CA, USA, August 8-10, 2016*, pages 1–10. 5

[20] Guo, R., Liao, X., Jin, H., Yue, J., and Tan, G. (2015). Nightwatch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 307–318. 5

[21] Hamerly, G., Perelman, E., Lau, J., and Calder, B. (2005). Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28. 19

[22] Hassan, H., Pekhimenko, G., Vijaykumar, N., Seshadri, V., Lee, D., Ergin, O., and Mutlu, O. (2016). Chargecache: Reducing dram latency by exploiting row access locality. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 581–593. IEEE. 1

[23] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17. 19

[24] Hirzel, M. (2007). Data layouts for object-oriented programs. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 265–276. 5

[25] Hirzel, M., Diwan, A., and Hertz, M. (2003). Connectivity-based garbage collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 359–373. 5

[26] Ho, Y., Huang, G. M., and Li, P. (2009). Nonvolatile memristor memory: device characteristics and design implications. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 485–490. ACM. 1

[27] Hoelzle, U. and Barroso, L. A. (2009). *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition. 1

[28] Hundt, R., Mannarswamy, S., and Chakrabarti, D. (2006). Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 233–244, Washington, DC, USA. IEEE Computer Society. 5

[29] Intel (2016). 3d xpoint. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html. 1, 40

[30] Jantz, M. R., Robinson, F. J., Kulkarni, P. A., and Doshi, K. A. (2015). Cross-layer memory management for managed language applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 488–504, New York, NY, USA. ACM. 8

[31] Jantz, M. R., Strickland, C., Kumar, K., Dimitrov, M., and Doshi, K. A. (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, pages 155–166. 5, 9

[32] Jula, A. and Rauchwerger, L. (2009). Two memory allocators that use hints to improve locality. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 109–118. 5

[33] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. (2015). Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3):158–169. 1

[Kim et al.] Kim, Y., Yang, W., and Mutlu, O. Ramulator: A fast and extensible dram simulator. viii, 4, 14, 19

[35] Kleen, A. (2004). A numa api for linux. *SUSE Labs white paper*. 9

[36] Kristian Vatto, I. C. and Smith, R. (2015). Analyzing intel-micron 3d xpoint: The next generation non-volatile memory. http://www.anandtech.com/show/9470/intel-and-micron-announce-3d-xpoint-nonvolatile-memory-technology-1000x-higher-performance-endurance-than-nand. 2

[37] Kültürsay, E., Kandemir, M., Sivasubramaniam, A., and Mutlu, O. (2013). Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE. 1

[38] Lai, S. (2003). Current status of the phase change memory and its future. In *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*, pages 10–1. IEEE. 1

[39] Lattner, C. and Adve, V. (2005). Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 129–142, New York, NY, USA. ACM. 5

[40] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. (2009). Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM. 1

[41] Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., and Keller, T. W. (2003). Energy management for commercial servers. *Computer*, 36(12):39–48. 1

[42] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. (2009). Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA. ACM. 1

[43] Loh, G. H. and Hill, M. D. (2011). Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 454–464. ACM. 2, 5

[44] Loh, G. H., Jayasena, N., McGrath, K., OConnor, M., Reinhardt, S., and Chung, J. (2012). Challenges in heterogeneous die-stacked and off-chip memory systems. In *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*. 2, 5

[45] Lu, S.-L., Lin, Y.-C., and Yang, C.-L. (2015). Improving dram latency with dynamic asymmetric subarray. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 255–266, New York, NY, USA. ACM. 1

[46] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005a). Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200. 11

[47] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005b). Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM. 15

[48] Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA. 8

[49] McKee, S. A. (2004). Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA. ACM. 1

[50] Meswani, M., Blagodurov, S., Roberts, D., Slice, J., Ignatowski, M., and Loh, G. (2015). Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136. x, 3, 5, 22, 30, 33, 34, 35

[51] Meza, J., Chang, J., Yoon, H., Mutlu, O., and Ranganathan, P. (2012). Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Computer Architecture Letters*, 11(2):61–64. 2, 5

[52] Mittal, S. and Vetter, J. S. (2016). A survey of techniques for architecting dram caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863. 2

[53] Mutlu, O. and Subramanian, L. (2014). Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, 1(3):19. 1

[54] Nethercote, N. (2008). SPEC2006 zeusmp and dealII on Valgrind 3.3.0. 20

[55] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA. ACM. 20

[56] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL. USENIX. 1

[57] Oskin, M. and Loh, G. H. (2015). A software-managed approach to die-stacked dram. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 188–200. IEEE. 5

[58] Pivotal (2016). Pivotal gemfire. Web page at https://pivotal.io/big-data/pivotal-gemfire. 1

[59] Plattner, H. and Zeier, A. (2011). *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer Publishing Company, Incorporated, 1st edition. 1

[60] Reed, D. A. and Dongarra, J. (2015). Exascale computing and big data. *Commun. ACM*, 58(7):56–68. 1

[61] Seidl, M. L. and Zorn, B. G. (1998). Segregating heap objects by reference behavior and lifetime. *SIGPLAN Not.*, 33(11):12–23. 5

[62] Sodani, A. (2015). Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE. 2

[63] Son, Y. H., Seongil, O., Ro, Y., Lee, J. W., and Ahn, J. H. (2013). Reducing memory access latency with asymmetric dram bank organizations. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 380–391, New York, NY, USA. ACM. 1

[64] Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., and Davis, A. (2010). Micro-pages: Increasing dram efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38(1):219–230. 5, 6

[65] Tolentino, M. E., Turner, J., and Cameron, K. W. (2009). Memory miser: Improving main memory energy efficiency in servers. *IEEE Transactions on Computers*, 58(3):336–350. 1

[66] Vetter, J. S. and Mittal, S. (2015). Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Computing in Science & Engineering*, 17(2):73–82. 1

[67] Wang, Z., Wu, C., and Yew, P.-C. (2010). On improving heap memory layout by dynamic pool allocation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 92–100, New York, NY, USA. ACM. 5

[68] Ware, M., Rajamani, K., Floyd, M., Brock, B., Rubio, J. C., Rawson, F., and Carter, J. B. (2010). Architecting for power management: The ibm® power7 approach. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11. IEEE. 1

[69] Wong, H.-S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., Asheghi, M., and Goodson, K. E. (2010). Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227. 1

[70] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on*

Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association. 1

[71] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948. 1

Vita

Adam Howard was born in Charlotte, NC, to John and Kathy Howard. His family moved frequently around North Carolina following his father's work until it landed them in Hendersonville, TN. There, Adam attended Jack Anderson Elementary school, Knox Doss Middle School, and finally Station Camp High School, where he participated in marching band and graduated with honors. After high school, Adam was accepted into the College of Engineering at the University of Tennessee, Knoxville.

Originally majoring in Electrical Engineering, Adam quickly discovered a passion for computer science and programming, and switched his major to Computer Engineering. During his undergraduate program, he worked as a student intern for the National Institute for Computational Science, as a member of UT's first Student Cluster Challenge team which competed at Supercomputing 2013, and as a teaching assistant helping students just learning to program. Adam obtained a Bachelor of Science degree from The University of Tennessee in Computer Engineering in May 2015, and graduated Magna Cum Laude. After finishing his bachelors, Adam accepted a graduate research assistantship and began work on his Master of Science degree in Computer Science with Dr. Michael Jantz's CORSYS research group.