On Automated Feedback-Driven Data Placement in Multi-tiered Memory

T. Chad Effler[1], Adam P. Howard[1], Tong Zhou[1], Michael R. Jantz[1], Kshitij A. Doshi[2], and Prasad A. Kulkarni[3]

[1] University of Tennessee, {teffler,ahoward,tzhou9,mrjantz}@utk.edu
[2] Intel Corporation, kshitij.a.doshi@intel.com
[3] University of Kansas, kulkarni@ittc.ku.edu

Abstract. Recent emergence of systems with multiple performance and capacity tiers of memory invites a fresh consideration of strategies for optimal placement of data into the various tiers. This work explores a variety of cross-layer strategies for managing application data in multi-tiered memory. We propose new profiling techniques based on the automatic classification of program allocation sites, with the goal of using those classifications to guide memory tier assignments.We evaluate our approach with different profiling inputs and application strategies, and show that it outperforms other state-of-the-art management techniques.

1 Introduction

Systems with multiple tiers of memory that are directly accessible via processor-memory buses are emerging. These tiers include (i) a limited capacity high-performance MCDRAM or HBM tier, (ii) a traditional DDR3/4 DRAM tier, and 3) a large capacity (∼terabytes) tier [1] whose performance may lag current DDR technologies by only a small factor. For a virtuous blend of capacity and performance from the multiple tiers, memory allocation needs to match different categories of data to the performance characteristics of the tiers into which they are placed, within the capacity constraints of each tier.

One approach is to exercise the faster, lower capacity tier(s) as a large, hardware-managed cache. While this approach has the immediate advantage of being backwards compatible and software transparent, it is not flexible and imposes unpalatable architectural costs that are difficult to scale in line with capacity increases [2]. An alternative approach is for application and-or operating system (OS) software to assign data into different memory tiers with facilities to allow migration of data between those tiers as needed. Monitoring of per-page accesses has been proposed recently [3, 4] with the goal of letting an OS (re)assign tiers. While this approach preserves application transparency, it is strictly reactive and relies on non-standard hardware. A third approach is annotation of source code [5–7] by which developers take control of, and coordinate tier assignments at the finer-grain of program objects. This approach requires expert knowledge, manual modifications to source code, and risks making such guidance stale as programs and configurations evolve.

Our work aims to combine the power and control of profile-guided and application-directed management with the transparency of OS-based approaches without relying on non-standard hardware. Allocation code paths are grouped into various sets on the basis of prior profiling, and the sets are preference-tied to different tiers in the underlying memory hardware. During execution, these preferences guide the placement of data. This approach does not require source code modifications and permits adapting to memory usage guidance for different program inputs and alternating phases of execution. In this paper, we describe the design and implementation of our automated application guidance framework, and then compare its performance to other hardware- and software-based hybrid memory management strategies using SPEC CPU2006 as workload.

This work makes the following important contributions: 1) We propose, design, implement and evaluate a multi-tiered allocation strategy that uses prior information to group sites for automatic tier selection, 2) We build an open-source simulation-based framework for instrumenting and evaluating it, including a custom Pin binary instrumentation tool [8], as well as extensions to the jemalloc arena allocator [9] and to Ramulator [10], 3) We show that a guidance-based approach has the potential, even when guidance has some inaccuracy, to outperform precise information based reactive placement of data, and 4) We find that adapting to individual program phases has limited benefit, suggesting that a simpler, static policy based on prior profiling is likely to be good enough.

2 Related Work

New frameworks, techniques, and strategies for managing heterogeneous memory systems [5, 3, 7, 6, 11–13, 4, 14] have emerged recently. Of these, several works [7, 6, 13, 14] employ profiling to find frequently accessed data structures, and translate their findings into tiering hints that can be inserted into program source code. Our approach combines runtime allocation site detection with a custom arena allocator to enable memory usage guidance without altering source, and is the first to explore impacts from variation in profiling inputs and from adapting tiering to individual program phases.

Cross-layer management techniques have also been used to optimize data placement across NUMA or other parts of the (single-tier) memory hierarchy [15–18]. Some of these works [17, 18] rely on program profiling and analysis to guide placement decisions. While this work employs similar techniques, the goals, application, and effects of our proposed management strategies are very different.

3 Feedback-Driven Data Placement for Hybrid Memories

Our approach uses a capacity normalized access metric to generate guidance: informally, it seeks to favor placing smaller and hotter objects for allocation into a higher performance tier. This metric is generated on the basis of prior profiling, and is associated with the code paths (also called allocation sites[4]) by

[4] This allocation site-based strategy for optimizing accesses-per-byte is designed to obviate tracing or sampling on an object-by-object basis.

which objects are allocated. Since the number of allocation sites can be much larger than the number of memory tiers, allocation sites are further grouped into sets (as described shortly in Sec. 3.2) and this partitioning guides the placements at run time.[5] To bound evaluation scope, each application is assumed to execute within a container with fixed upper tier capacity.

3.1 Allocation Site Partitioning: We propose two simple alternatives for partitioning program sites into groups. The first alternative is called knapsack. Inspired by [17], it uses a classical 0/1 knapsack formulation to produce groupings that collectively fit into a knapsack (of any capacity by which we want to represent the upper tier) while maximizing aggregate access into it. The second alternative is called hotset. It avoids a weakness of knapsack, namely, that knapsack may exclude an allocation site based on the raw capacity of that site, even if allocations from it exhibit a high access count. In the hotset approach, we sort allocation sites by accesses-per-byte scores and then select those with highest scores until we exceed an alternate soft capacity. The limiting capacity for inducing the hot-cold split in each case is taken as a fraction of an applications total dynamic footprint, D. Thus, if this fraction is 12.5%, then the knapsack approach selects allocation sites such that the aggregate size is just below $D/8$, while the hotset approaches stops after $D/8$ is crossed.

3.2 Profile-Guided Management: During a guided run, the application address space is divided into arenas, each of which is page-aligned and therefore can be independently assigned to a memory tier. Using a system interface, such as the NUMA API or memory coloring [16], the application or runtime can instruct the OS memory manager about preferred arena-to-tier assignments. Our framework supports two schemes for using prior guidance: *static arena allocation* and *per-phase arena allocation*. In the static scheme, the application creates two arenas: hot and cold, and guides allocations from the hotset/knapsack partitions into the hot (and all else into the cold) arena, with the guidance remaining fixed across the run. The per-phase scheme is designed to adjust with changes of behavior during the run. It uses per-phase guidance for grouping sites into arenas such that phase by phase, an arena can (optionally) swap tiers, but may never be in more than one tier at a time.

Figure 1 illustrates the per-phase scheme. Program execution is divided into phases. For N phases, an N bit vector per allocation site describes the sites hot/cold classification phase by phase. For instance, if a site has a vector '10100', the vector indicates it is hot in phases 3 and 5, and cold in phases 1, 2, and 4, across 5 phases. The total number of unique vectors determines the total number of arenas created at application startup time. During execution, sites with matching bit vectors allocate data to the same arena. Upon a phase transition[6],

[5] The primary goal of this work is to study the potential benefits of automated application guidance. While our simulation-based evaluation neglects overhead of profiling, Sec. 4 covers how in practice, allocation site based guidance can be generated (either online or offline) and applied in direct execution with negligible overhead.

[6] Phase transitions may be detected online by several means, including through models of instruction and data access behaviors, hardware event ratios, etc.

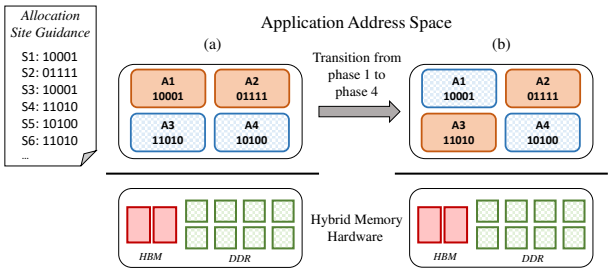

Fig. 1. Per-phase strategy for managing hybrid memories. (a) In phase 1, A1 and A2 correspond to hot allocation sites and are originally mapped to the HBM tier. (b) On transition to phase 4, the guidance indicates A3 will become hot, and A1 is now cold. The application communicates this guidance to the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

the OS adjusts the hot/cold tier classification of each arena and migrates data accordingly. This arena-based coarse-grained remapping of tiers to virtual ranges permits efficiently amortized and application transparent migration.

4 Implementation Details

4.1 Associating Memory Usage Profiles with Program Allocation Sites: To collect access profiles, we instrumented using the Pin framework (v. 2.14) [8]. Our custom Pintool[7] intercepts all of the applications allocation and deallocation requests (specifically, all calls to `malloc`, `calloc`, `realloc`, and `free`). The arguments to these routines are then used to build a shadow structure mapping each allocated region to its allocation site with context. The tool captures the estimated capacity (in peak resident set size, accounting for dynamic allocations and unmaps) allocated at each allocation site; and, it computes an estimated aggregate post-cache memory access counts over those allocations by filtering the accessed addresses through an in-band cache simulator. At the end of application execution, the tool outputs the allocation site profiles to a file.

4.2 Hybrid Memory Management: Evaluation requires two major components: 1) an allocator that uses the above profiles to partition allocation sites into arenas, and 2) a manager that models the effect of, and which applies, guidance-based management strategies.

4.2.1 Arena Allocation: We employ shared library preloading to dynamically link each evaluation run to a custom allocator that overrides allocation requests with our own arena allocation routines based off of jemalloc [9]. Some calls to `realloc` may request a different arena from that used for the original data, and for those, the overriding call transfers the resized data into the new arena.

[7] For direct execution, an alternative to Pin based instrumentation is to use LLVM inserted wrappers (as described in Sec. 4.2.1), and to sample access rates through hardware-based counters (e.g., using the PEBS facility on modern Intel processors).

Table 1. Benchmarks with usage statistics.

Benchmark	MB	Sites	Allocs	LLCPKI	
				512 KB	8 MB
bzip2	853	10	174	15.43	-
gcc	901	19.6K	28.46M	32.18	-
mcf	1,683	5	6	95.26	46.17
milc	711	56	6.52M	47.77	23.72
cactusADM	668	5.3K	0.13M	15.46	5.07
leslie3d	146	101	0.31M	65.23	22.59
gobmk	39	175	0.66M	4.27	-
soplex	604	363	0.31M	57.30	22.07
hmmer	45	188	2.47M	46.31	-
GemsFDTD	884	509	0.75M	31.42	17.26
libquantum	105	10	180	40.95	29.06
h264ref	83	260	0.18M	7.39	-
lbm	415	4	5	66.72	38.75
sphinx3	72	281	14.22M	18.18	-
Average	514	1.9K	3.39M	38.85	25.59

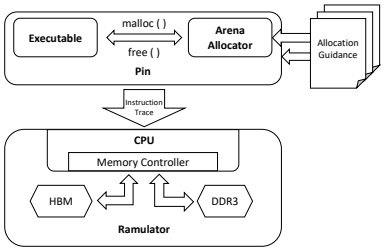

Fig. 2. Framework for simulating hybrid memory management.

To identify allocation sites during execution, our evaluation framework currently collects up to seven layers of call stack context using the `backtrace` routine from the C standard library. While straightforward and easy to implement, this approach can incur substantial overhead if there are too many allocation requests. In a set of native execution runs on an Intel Xeon-based server machine, we found that using backtrace for context detection incurs an average overhead of 3.6% for the 14 benchmarks listed in Table 1, with a maximum slowdown of more than 40% for *gcc*.

To eliminate these overheads, we developed a static compilation pass in the LLVM compiler infrastructure [19] that automatically creates a separate code path for each hot call site and its context. Preliminary tests show that this static pass completely eliminates the overhead of context detection for our benchmark set, and is still able to identify the same set of hot data as the `backtrace` technique.[8] Since the primary goal of this work is to study the potential benefits of automated application guidance during hybrid memory management, we leave full evaluation of the accuracy and performance of static context detection as future work. The simulation-based experiments in Sec. 6 assume no additional overhead for context detection.

4.2.2 Simulation of Hybrid Memory Architectures: Our framework for modeling the behavior and performance of hybrid memory systems adopts and extends the Ramulator DRAM simulator [10]. 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. For this work, we modified Ramulator's memory controller to support multiple tiers with distinct DRAM standards simultaneously. This extended simulator maintains a map of which physical pages correspond to each tier, and sends each request to the appropriate DRAM model

[8] Other, more compact encodings of the allocation sites may also be employed – e.g., a low-overhead approximate method in direct execution is to use a hash over (call-return) last branch records (LBR) recorded by a processor's monitoring unit.

depending on its address. It 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 2 illustrates our approach. At startup, the application connects to a custom Pintool, which filters each load/store through an online cache model and emits a post-cache instruction trace into the extended Ramulator. At the same time, the custom allocator automatically partitions the allocation sites into arenas according to the pre-computed guidance files, and the Pintool inserts the preferred tier into the trace. Ramulator interprets the trace, one request at a time, mapping new data to the appropriate memory tier, until completion.

5 Experimental Framework

5.1 Simulation Platform: Ramulators execution model includes a 3.2 GHz, 4-wide issue CPU with 128-entry re-order buffer, and assumes one cycle for each non-memory instruction. To estimate the impact of various hybrid memory strategies we simulated with two processor cache configurations: 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.

We added logic to Ramulator for simulating a hybrid memory architecture with two tiers: a high-performance tier with configurable, limited capacity, and a slower tier with no capacity bound. We experimented with a range of capacities for the upper tier, and opted to use 12.5% of peak resident set size (RSS) (i.e., 1:8 ratio across tiers) in our evaluations. The choice of 1:8 reasonably approximates the expected capacity ratios of typical (current [20] and expected [1]) hybrid memory systems.

All experiments use the (unmodified) HBM standard included with Ramulator to simulate the fast tier, and use either the DDR3 or DDR4 standard to simulate the slow tier. Detailed statistics about each standard, including rate, timing, bus width, and bandwidth, are listed in Table 4 of [10]. Although we evaluate all of the proposed strategies with an HBM-DDR4 configuration, our detailed experimental results use HBM-DDR3 to model a wider asymmetry between the upper and lower tiers. A summary of our performance results for both platform configurations is presented in Sec. 6.5.

Some of our studies include migration of data between memory tiers. To model the cost of data movement, we folded penalties for migration into our simulations experiments as described in [3], which are as follows. Page faults and TLB shootdowns incur fixed penalties of 5μs and 3μs, respectively. The experimental framework further adds execution time for data migrations, which is a function of the bandwidth of the lower tier.

For a faithful reflection of the effects that guidance-based strategies have on allocation behavior and heap layout, each experiment executes the entire program run from start to finish. However, detailed cache and memory simulations

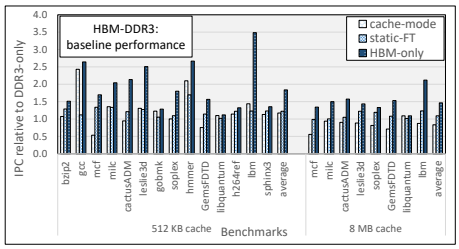

Fig. 3. Performance (IPC) of baseline configurations relative to DDR3-only.

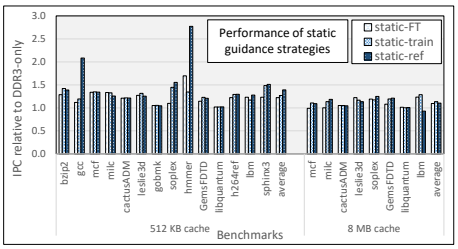

Fig. 4. Performance (IPC) with static guidance strategies relative to DDR3-only.

are limited to only a representative portion of the run using Simpoints [21]. Unless stated otherwise, all of the experiments simulate a single, large, contiguous slice of 64 billion program instructions. With our simulation framework, this volume of instructions corresponds to at least 5 full seconds of execution time (measured in CPU cycles), and a typical execution time of 20 to 30 seconds.

5.2 Benchmarks Description: For our evaluation, we used the standard SPEC CPU2006 benchmark suite [22]. We compiled the benchmarks using gcc (version 4.8.5) with -O2. Profile 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.

To identify applications where efficient utilization of the upper-level memory can have a significant impact on program performance, we conducted pilot measurements using shorter simulations of up to 10 program phases and 1 billion instructions per phase for each program run.[9] For these experiments, we evaluated each benchmark against the two cache configurations and against Ramulators default memory model with 1) a single-tier of (unlimited capacity) HBM (HBM-only) and 2) a single-tier of DDR3 (DDR3-only).

The results of the pilot measurements showed that there is significant potential to improve performance with HBM. 14 (of 28) benchmarks exhibited more than 10% IPC improvement with HBM relative to DDR3 with the 512 KB cache, while 8 benchmarks show similar improvements with the 8 MB cache. The remainder of this work focuses on this limited set of benchmark-cache pairs. Table 1 lists our selected benchmarks along with their memory usage information.

6 Evaluation

6.1 Baseline Configurations: For baseline comparison, we implement two strategies that have been common in hybrid memory systems. The first uses the upper tier as a large direct-mapped cache to hold data brought in from an even larger lower tier [20]. We refer to this type of hardware based tiering

[9] We had to omit *zeusmp* due to an incompatibility with our adopted basic block vector collection tool [23].

as *cache mode* (not to be confused with processor caches). The other baseline strategy is the *static first touch (FT)* [3] policy. Under static FT, when a page is first touched, it is instantiated in HBM if possible and in DDR (the lower tier) otherwise; and, remains there until unmapped.

Figure 3 shows the performance (IPC) of the two baseline policies– cache mode and static FT in a hybrid HBM-DDR3 system where the capacity of the HBM tier is 12.5% of the DDR3 tier. For each benchmark, the IPC in Fig. 3 is shown relative to the IPC of the DDR3-only configuration. Hence, while cache mode outperforms static FT for a few benchmarks (e.g., *gcc* and *hmmer*) static FT is the superior choice. On average, static FT allocation improves IPC (over DDR3-only) by 22% and 9% for the 512 KB and 8 MB CPU cache sizes, respectively. In cache mode, the average IPC change is 17% better for the 512 KB CPU cache but 17% worse for the 8MB CPU cache. Our simulation diagnostics show that the degradation in cache mode occurs due to a high miss rate (over 67% for the 8MB cache) resulting in higher overheads for memory traffic. A third bar in Fig. 3 also shows that in the idealized HBM-only case, the average IPC is better by 61.9% and 30.1% respectively for the small and large CPU caches.

6.2 Static Application Guidance: We next introduce a *static guidance* hybrid management policy that uses prior profiling to partition allocation sites into hot and cold subsets, and then applies the static arena allocation scheme to separate hot and cold data in the evaluation run. The hot space places data in the HBM tier on a first touch basis, while cold data is always assigned to DDR.

We conducted an initial set of shorter simulations (10 phases, 1B instructions per phase) to assess the impact of different strategies for selecting hot subsets. For these experiments, we compute profiling with the *ref* and *train* program inputs and construct hot subsets using the knapsack and hotset strategies with capacities of 3.125%, 6.25%, 12.5%, 25.0%, and 50.0%. We find that the best size for each approach varies depending on the benchmark and profile input. Knapsack achieves its best performance with the largest capacity (50.0%), while hotset does best with sizes similar or smaller than the upper tier capacity limit (of 12.5%). Across all benchmarks, the best hotset outperforms the best knapsack by 4.4% with the *train* profile and by 4.2% with the *ref* profile, on average. This outcome lends strength to the idea that being too conservative in cases where an allocation site with very hot data does not fit entirely in the upper tier is less effective than allowing a portion of the sites data to map to the faster device. We therefore continue using only the hotset strategy and select the hotset capacity that performs best on average, as follows: 12.5% for *train* and 6.25% for *ref* with the smaller cache, and 25% for both *train* and *ref* with the larger cache.

Figure 4 shows the IPC of the benchmarks with the static hotset guidance approaches with *train* and *ref* profiling inputs (respectively labeled as *static-train* and *static-ref*) relative to single-tier DDR3. Thus, application guidance, whether based on profiles of the *train* or *ref* input, does better than static FT during the evaluation run. On average, the more accurate *ref* profile enables static-ref to outperform static-train by more than 12%, when the CPU cache is small (512 KB), but the difference is negligible when the cache is larger (8MB).

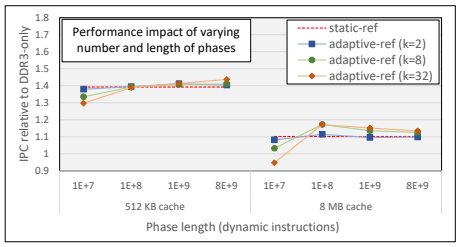

Fig. 5. Performance (IPC) of adaptive-ref with different # and length of phases.

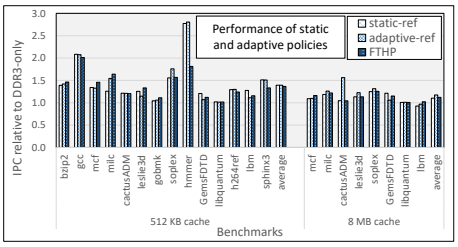

Fig. 6. Performance (IPC) with adaptive strategies on relative to DDR3-only.

Surprisingly, with the 8 MB cache, static-train performs slightly better due to a skewed result from the *lbm* benchmark. Further analysis shows that *lbm* produces about the same amount of traffic into the upper tier with both static-ref and static-train, but the disparity is primarily due to an effect on spatial locality caused by different data layouts. We plan to fully evaluate the impact of our technique on spatial locality in future work.

6.3 Adaptive Application Guidance: We next examine the potential benefit of adapting guidance to changing access patterns, rather than being locked into static guidance policies. To study this effect, we design and evaluate the *adaptive-ref* policy, described below.

For each benchmark, we use Simpoints to divide the evaluation (*ref* input) run into slices of dynamic instructions (of length l), and then classify each slice into one of up to k program phases. We then conduct profiling and compute hot-sets for each program phase, and use this guidance to apply the per-phase arena allocation scheme during the evaluation run. When the application enters a new phase, it suspends execution, and attempts to migrate data in each arena to the appropriate tier using the guidance for the *upcoming* program phase. Since our goal is to investigate the potential advantages of this approach, our experiments assume the application is always able to detect program phase changes accurately and immediately.

We conducted a series of experiments with the adaptive-ref policy varying the length (l) and maximum number of program phases (k). For k values of 2, 8, and 32, Fig. 5 plots a line showing the average performance (IPC) of the adaptive-ref policy, relative to DDR3-only, with phase lengths of 10 million (M), 100M, 1 billion (B), and 8B dynamic instructions. Additionally, we plot dashed lines to show the average IPC of the static-ref policy with each cache size. Detailed results comparing the static-ref and best adaptive-ref configuration (with $k = 8$, l =100M) for each benchmark are shown in Fig. 6.

In most cases, we find that adaptive-ref exhibits similar performance as static-ref, even with the idealistic assumption of accurate phase detection. Different phase lengths have little impact on the proportion of accesses to the HBM tier, but selecting a length that is too short will incur significant data migration overheads, and result in worse overall performance. For a few workloads, such as *milc* and *cactusADM*, increasing the number of program phases does produce benefits, but on average the impact is small. Additionally, we find that the best

Table 2. % of accesses to upper-level memory and data migrated (in GB).

Benchmark	% of accesses to upper-level memory						GBs migrated		
	cache-mode	static FT	static train	static ref	adaptive ref	FTHP 1s	cache-mode	FTHP 1s	adaptive ref
512 KB cache									
bzip2	80.92	69.03	77.08	70.59	81.06	94.09	207.44	0.99	16.94
gcc	96.17	16.78	24.61	80.57	70.14	69.24	68.70	3.73	16.99
mcf	20.04	48.80	48.50	48.48	48.50	63.39	256.59	6.15	0.00
milc	80.97	31.58	32.54	27.16	53.80	56.98	55.36	0.50	19.75
cactusADM	79.27	46.89	46.70	46.69	46.54	44.37	57.24	0.66	0.00
leslie3d	76.12	28.21	27.83	28.76	18.10	37.17	106.71	0.15	0.00
gobmk	95.39	27.84	13.78	14.90	14.18	37.53	13.19	0.04	0.00
soplex	59.65	15.87	46.37	52.92	61.66	51.52	137.88	0.12	0.13
hmmer	96.04	63.31	34.41	75.00	75.79	71.08	59.20	0.01	0.00
GemsFDTD	11.05	13.46	15.56	21.55	12.93	17.70	79.66	2.02	63.95
libquantum	99.53	11.55	11.54	11.54	11.54	13.53	1.04	0.02	0.00
h264ref	95.63	72.13	85.55	86.84	88.21	77.83	32.04	0.02	0.75
lbm	94.97	12.77	12.72	12.72	12.51	10.64	145.82	0.55	29.81
sphinx3	62.65	45.66	59.33	61.83	57.25	69.18	29.74	0.01	0.17
Average	74.88	35.99	38.32	45.68	46.59	51.02	89.33	1.07	10.61
8 MB cache									
mcf	17.77	24.78	24.88	25.22	24.88	43.24	137.19	5.60	0.00
milc	57.89	15.48	16.09	21.58	32.42	29.44	43.87	0.58	20.92
cactusADM	30.58	29.74	29.74	29.69	32.66	27.31	13.69	0.52	0.00
leslie3d	43.66	20.81	20.96	20.75	20.04	14.74	52.86	0.10	0.00
soplex	43.93	30.85	19.64	30.39	35.40	30.16	80.07	0.11	2.17
GemsFDTD	4.72	14.40	15.00	14.99	11.87	9.60	62.14	1.38	61.97
libquantum	99.97	12.50	0.00	12.49	12.49	11.33	0.03	0.01	0.00
lbm	84.00	12.71	12.79	12.79	12.47	11.73	50.37	0.10	30.01
Average	47.82	20.16	17.39	20.99	22.78	22.19	55.03	1.05	14.38

adaptive-ref configuration drives only slightly ($< 2\%$) more traffic to the HBM tier than the static approach, as shown in Table 2. Thus, static-ref makes nearly as-good placement decisions across phases as adaptive-ref, even though it is not capable of adapting to the individual program phases.

6.4 Comparison with OS/Architectural Reactive Profiling: Using our simulation framework, we implemented the *first-touch-hot-page (FTHP)* reactive profiling approach from Meswani et al. [3]. FTHP uses non-standard page access counters in hardware to identify recently hot physical pages and migrate them at epoch boundaries. We evaluate FTHP using two epoch lengths of 1s and 100ms. For our comparisons, we chose the 1s epoch because it achieves slightly (1.2%) better performance with our benchmarks.

Referring again to Fig. 6, a third bar shows the results for FTHP relative to DDR3-only, on an identical HBM-DDR3 platform with 12.5% HBM capacity. Thus, even though they do not have the benefit of dynamic feedback from specialized hardware, the application guidance policies often achieve similar performance as FTHP. With the 512 KB cache, static-ref and adaptive-ref respectively outperform FTHP by 2.8% and 2.9%, while with the 8 MB cache, static-ref performs slightly (1.9%) worse, and adaptive-ref performs 5.3% better. Even static-train (shown in Fig. 4) performs slightly (1.5%) better than FTHP with the larger cache, but does exhibit some slowdown (9.6%) with the smaller cache.

Both adaptive-ref and FTHP limit the frequency of data migration to amortize the cost of page faults and TLB synchronization. The final three columns of Table 2 show the amount of data migrated (in GB) for each adaptive policy.

Note also that the amount of migration for both FTHP and adaptive-ref depends on the length of each epoch/phase. For instance, compared to the adaptive-ref configuration in the table (with k =8, l =100M), adaptive-ref with l =10M migrates almost $2.4x$ more data over the course of each run, on average. Considering these results with the performance results in Fig. 6 and HBM traffic comparison in Table 2, we conclude that, although more frequent migration can steer a higher portion of traffic to the HBM, the additional costs often outweigh the performance benefits for our selected benchmarks.

6.5 Performance Summary: Table 3 presents the average IPC of all of the management policies for both HBM-DDR3 and HBM-DDR4 platforms with 12.5% capacity in the HBM tier. Each set of results uses the corresponding DDR3/4-only configuration as its baseline. As expected, the policies on the HBM-DDR4 platform exhibit similar performance trends as on the HBM-DDR3 platform. On average, the application-guided policies achieve the best performance on HBM-DDR4, boosting performance with the small and large caches by more than 15% and 20% compared to cache mode, by 16% and 7% compared to static FT, and by 5% and 3% compared to FTHP.

Table 3. Performance (IPC) summary of different allocation strategies.

Policy	512 KB cache		8 MB cache	
	HBM-DDR3	HBM-DDR4	HBM-DDR3	HBM-DDR4
cache-mode	1.173	1.173	0.833	0.907
static-FT	1.223	1.165	1.094	1.045
static-train	1.269	1.226	1.136	1.115
static-ref	1.393	1.325	1.102	1.113
adaptive-ref	1.393	1.323	1.173	1.099
FTHP	1.347	1.272	1.107	1.084
HBM-only	1.838	1.568	1.466	1.255

7 Conclusions & Future Work

This work demonstrates that emerging hybrid memory systems will not be able to rely solely on conventional hardware-based caching or coarse-grained software approaches, such as static NUMA assignments, and stand to benefit greatly from fine-grained, application-level guidance. The results point to a need for developing new source, binary, and run-time capabilities to make application guided memory tiering practical. While the current evaluation uses simulation, our goal is to adapt our automated guidance framework for direct execution on real hybrid memory hardware. The immediate next steps include development of hardware-based sampling to profile memory accesses during native execution and low-overhead context detection techniques as described in Sec. 4.

Other findings in this study warrant additional investigation. In many cases, we found that using tailoring application guidance to each program phase has a relatively small impact on program performance. Further research is necessary to understand the relationship between program phases and memory behavior, 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. Finally, we plan to evaluate the potential of extending guidance to other parts of the memory hierarchy, such as caching or prefetching.

References

1. Intel: 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html (2016)
2. Mittal, S., Vetter, J.S.: A survey of techniques for architecting dram caches. IEEE Transactions on Parallel and Distributed Systems **27**(6) (2016) 1852–1863
3. Meswani, M., Blagodurov, S., Roberts, D., Slice, J., Ignatowski, M., Loh, G.: Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In: HPCA, 2015. (Feb 2015)
4. Li, Y., Ghose, S., Choi, J., Sun, J., Wang, H., Mutlu, O.: Utility-based hybrid memory management. In: IEEE CLUSTER. (Sept 2017)
5. Cantalupo, C., Venkatesan, V., Hammond, J.R.: User extensible heap manager for heterogeneous memory platforms and mixed memory policies. "http://memkind.github.io/memkind/memkind_arch_20150318.pdf" (2015)
6. Dulloor, S.R., et al.: Data tiering in heterogeneous memory systems. In: Eleventh European Conference on Computer Systems, ACM (2016) 15
7. Agarwal, N., et al.: Page placement strategies for GPUs within heterogeneous memory systems. SIGPLAN Not. **50**(4) (March 2015) 607–618
8. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. **40**(6) (2005) 190–200
9. Evans, J.: A scalable concurrent malloc (3) implementation for freebsd. (2006)
10. Kim, Y., et al.: Ramulator: A fast and extensible dram simulator. (2015)
11. Giardino, M., Doshi, K., Ferri, B.H.: Soft2lm: Application guided heterogeneous memory management. In: IEEE International Conference on Networking, Architecture and Storage (NAS), USA, 2016. (2016) 1–10
12. Agarwal, N., Wenisch, T.F.: Thermostat: Application-transparent page management for two-tiered main memory. In: ASPLOS. ASPLOS '17, New York, NY, USA, ACM (2017) 631–644
13. Peng, I.B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., Markidis, S.: RTHMS: A tool for data placement on hybrid memory system. In: ISMM 2017
14. Servat, H., Pea, A.J., Llort, G., Mercadal, E., Hoppe, H., Labarta, J.: Automating the application data placement in hybrid memory systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). (Sept 2017)
15. Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., Roth, M.: Traffic management: a holistic approach to memory placement on numa systems. SIGPLAN Not., 48(4) **48**(4) (2013) 381–394
16. Jantz, M.R., et al.: A framework for application guidance in virtual memory systems. In: Virtual Execution Environments. VEE '13 (2013) 155–166
17. Jantz, M.R., et al.: Cross-layer memory management for managed language applications. In: ACM/SIGPLAN OOPSLA, New York, NY, USA, ACM (2015)
18. Guo, R., Liao, X., Jin, H., Yue, J., Tan, G.: Nightwatch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). (2015) 307–318
19. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization. (2004)
20. Sodani, A.: Knights Landing (knl): 2nd generation Intel® Xeon Phi processor. In: Hot Chips 27 Symposium (HCS), 2015 IEEE, IEEE (2015) 1–24
21. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0. JILP **7**(4) (2005)
22. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH (2006)
23. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI. (2007)