MaPHeA: A Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation Framework

Deok-Jae Oh
Seoul National University
South Korea
djoh0967@scale.snu.ac.kr

Yaebin Moon
Seoul National University
South Korea
ybmoon@scale.snu.ac.kr

Eojin Lee
Samsung Electronics
South Korea
eojin29.lee@samsung.com

Tae Jun Ham
Seoul National University
South Korea
taejunham@snu.ac.kr

Yongjun Park
Hanyang University
South Korea
yongjunpark@hanyang.ac.kr

Jae W. Lee
Seoul National University
South Korea
jaewlee@snu.ac.kr

Jung Ho Ahn
Seoul National University
South Korea
gajh@snu.ac.kr

Abstract

Hardware performance monitoring units (PMUs) are a standard feature in modern microprocessors for high-performance computing (HPC) and embedded systems, by providing a rich set of microarchitectural event samplers. Recently, many profile-guided optimization (PGO) frameworks have exploited them to feature much lower profiling overhead than conventional instrumentation-based frameworks. However, existing PGO frameworks mostly focus on optimizing the layout of binaries and do not utilize rich information provided by the PMU about data access behaviors over the memory hierarchy. Thus, we propose MaPHeA, a lightweight **M**emory hierarchy-**a**ware **P**rofile-guided **H**heap **A**llocation framework applicable to both HPC and embedded systems. MaPHeA improves application performance by guiding and applying the optimized allocation of dynamically allocated heap objects with very low profiling overhead and without additional user intervention. To demonstrate the effectiveness of MaPHeA, we apply it to optimizing heap object allocation in an emerging DRAM-NVM heterogeneous memory system (HMS), and to selective huge-page utilization. In an HMS, by identifying and placing frequently accessed heap objects to the fast DRAM region, MaPHeA improves the performance

of memory-intensive graph-processing and Redis workloads by 56.0% on average over the default configuration that uses DRAM as a hardware-managed cache of slow NVM. Also, by identifying large heap objects that cause frequent TLB misses and allocating them to huge pages, MaPHeA increases the performance of read and update operations of Redis by 10.6% over the transparent huge-page implementation of Linux.

CCS Concepts: • Software and its engineering → Allocation / deallocation strategies; Main memory.

Keywords: Profile-guided optimization, heap allocation, heterogeneous memory system, huge page

ACM Reference Format:

Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W. Lee, and Jung Ho Ahn. 2021. MaPHeA: A Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation Framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21)*, June 22, 2021, Virtual, Canada. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3461648.3463844>

1 Introduction

Profile-guided optimization (PGO) [24] is a compiler technique that profiles the runtime behavior of an application and re-compiles it by applying appropriate transformations based on the characteristics of its collected behaviors. To improve the performance of an application effectively, PGO guides an optimization process by taking into account runtime information of the program that cannot be obtained when it is compiled for the first time.

Many PGO techniques have been proposed previously with different profiling methods, targets, and objectives. Most proposals target the binary layout of an application, such as code and data segments [12, 24, 42, 53, 54, 61]. As for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES '21, June 22, 2021, Virtual, Canada

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8472-8/21/06.

<https://doi.org/10.1145/3461648.3463844>

profiling methodology, sampling-based methods exploiting performance monitoring units (PMUs) [12, 53, 54] are gaining popularity due to their low runtime overhead in order to complement the instrumentation-based methods [24].

Moreover, *data address profiling* [4, 6, 25, 28], a recent addition to PMU capabilities for high-performance computing (HPC) and embedded systems, enables the profiler to pinpoint the instruction that causes a specific hardware event (e.g., cache misses) and/or the data it accesses. This feature introduces new promising opportunities for PGO to broadly expand its targets and optimization objectives. Especially, the emergence of big-data workloads [49, 65] and tiered memory systems in HPC [46, 58] and embedded systems [22, 23, 71] introduces new challenges on efficient memory management (e.g., data placement in heterogeneous memory systems (HMS) [2, 19, 57, 63, 67] and frequent TLB misses [36, 51, 55]). Data address profiling can potentially address them by profiling the program behaviors across the memory hierarchy and effectively guiding program optimization.

Based on this observation, previous techniques have exploited PMU to optimize data placement for non-uniform memory access (NUMA) and HMS [14, 45, 63, 66–68], hence improving program performance at a low profiling cost. However, they either incur substantial runtime overhead [14, 67, 68] or require manual programming effort for program transformation [14, 66–68]. Furthermore, they do not demonstrate the effectiveness for a wide range of use cases and/or fully utilize the information provided by recently introduced PMUs.

In this paper, we propose MaPHeA, a lightweight new **Memory hierarchy-aware Profile-guided Heap Allocation** framework applicable to HPC and embedded systems. It is *lightweight* as it profiles the memory access information of an application via hardware-event sampling. MaPHeA is *aware of the memory hierarchy* by utilizing data address profiling, which provides detailed information on individual memory accesses, such as the address and the type of access as well as where the data resides. It then analyzes which heap objects are performance-critical by using the information. For example, MaPHeA can identify the objects that are frequently accessed from a certain level in the memory hierarchy (e.g., LLC or DRAM) or cause frequent TLB misses. Then, it can apply optimizations such as allocating those hot objects to fast memory and allocating the TLB-miss-prone objects to huge pages. To realize this, we modify the intermediate representation of gcc and use a wrapper to the default malloc() to support tiered memory. To show the effectiveness of MaPHeA, we apply it to guiding memory allocation in an HMS and selective huge-page utilization. The HMS used for evaluation employs both Intel Optane DC Persistent Memory Module (DCPMM, slow memory) [41] and DRAM (fast memory). On the given HMS system, we utilize MaPHeA to properly allocate each object to either DRAM or DCPMM to reduce the average main-memory access latency. As a second use case of MaPHeA, we selectively utilize huge

pages on large objects to mitigate the performance impact of TLB misses.

We summarize the contributions of this paper as follows:

- We identify new opportunities for PGO to overcome the memory management challenges of emerging memory systems via optimizing memory allocation. This potential can be harnessed by exploiting the runtime information across the memory hierarchy provided by today’s PMUs.
- To capitalize on these opportunities, we propose MaPHeA, a fully automatic PGO framework that utilizes data access profiles taken from modern PMUs to guide memory allocation on the emerging large memory system.
- We demonstrate the effectiveness of MaPHeA using two use cases. We first show that allocating hot objects to the fast memory (DRAM) improves the performance of graph-processing and Redis [59] by 56.0% on average compared to the baseline that uses the fast memory as a hardware-managed cache of the slow non-volatile memory (NVM). Also, through allocating large heap objects causing frequent TLB misses to huge pages, MaPHeA increases the performance of read and update operations of Redis by 10.6% over the transparent huge-page (THP) implementation of Linux kernel 4.20.0 [39, 51].

2 Background and Motivation

2.1 Compiler-Assisted Profile-Guided Optimization

Profile-guided optimization (PGO) is a compiler technique that optimizes applications by utilizing profiling information. This technique enables the compiler to generate codes whose performance exceeds what can be achieved by exploiting compile-time information only. Thus, many PGO frameworks have been proposed for decades [12, 42, 53, 54, 56, 61], and multiple production-level compilers support PGO [24]. Conventional PGO techniques improve application performance by optimizing the application’s binary layout, such as code/data/stack/heap segments, and often achieve further performance gains through iterative optimization.

A main challenge for a PGO framework design is how to gather runtime information (profiling method). PMU sampling based profiling has become attractive due to its low profiling overhead and detailed information about microarchitectural events it provides. Thus, multiple major Internet service providers have recently adopted PMU sampling-based PGO [12, 13, 54].

2.2 Hardware-Assisted Event-Based Sampling

PMU is a specialized hardware unit inside the processor to analyze system performance. It counts the occurrences of pre-defined hardware events (e.g., instruction cycles, branch mispredictions, and cache misses) by looking up performance

Figure 1. The structure and operation of hardware-event sampling of Intel Processor Event-Based Sampling (PEBS).

monitoring counters (PMCs). The operations of Intel’s Processor Event-Based Sampling (PEBS) are illustrated in Figure 1. At every user-specified sampling period, the information gathered in the PMC is packed into PEBS record format, which contains the architectural state of the processor, including various registers.

The PEBS record is then stored in the memory region called a PEBS buffer. When the PEBS buffer contains more PEBS records than a pre-defined threshold, the PEBS records are finally saved into a file through an interrupt. This interrupt has a small latency (about 200 ns), and the interrupt overhead can be minimized by adjusting the counter overflow threshold and the PEBS buffer size (by default, the Linux perf tool that supports a profiling interface using PEBS has a buffer size of 4 KB and a sampling frequency of 1 KHz). The hardware-assisted event-based sampling technique, such as PEBS, has a much lower profiling overhead than the instrumentation-based method by using special hardware counters in the background. Writing to a file happens much less frequently than updating those counters.

The PMUs of modern microprocessors also provide enhanced functionality for profiling memory accesses, called *data address profiling*. Examples include Intel’s PEBS data linear address (PEBS-DLA) [28], IBM’s sampled data address register (SDAR) [25], AMD’s instruction-based sampling (IBS) [4], and ARM’s statistical profiling extension (SPE) [6]. Data address profiling collects runtime information, such as accessed data address, memory hierarchy level in which the data exists, and data access latency in cycles. This feature enables the profiler to identify which load/store instruction accesses the data in question and where this data resides. Therefore, it can be used for various PGO techniques, from memory allocation optimization to conventional binary layout optimization.

2.3 PMU Sampling-Based PGO

Existing PGO techniques for optimizing the data/stack/heap layouts can be effective for traditional applications to improve their cache utilization [15, 61] and data locality [62]. However, emerging applications demand more than cache optimization for PGO. In particular, big data analytics and machine learning applications have emerged to feature large memory footprints, substantially increasing the pressure to main memory. For such applications, PGO covering the

Figure 2. The trend of DRAM cache miss rates and TLB miss frequency as the number of records stored in Redis increases from 1M to 50M. Redis runs at a HMS that is configured to use Memory Mode.

entire memory hierarchy is necessary to maximize performance. Besides, near-far heterogeneous memory systems (HMSs) are becoming more popular, by utilizing emerging memory technologies such as 3D-stacked DRAM [30, 38], non-volatile memory (NVM) [47], and managed DRAM [37], which further complicate memory management.

As a motivational experiment, we run Redis [59], a popular in-memory key-value store, on an HMS, where the fast DRAM is configured as a hardware-managed cache of the slow NVM (see Section 5.1 for the detailed experimental setup). Figure 2 shows the changes in DRAM cache miss rates and relative increase in TLB misses (normalized to the 1M dataset case) as the dataset size scales. As the size of the dataset stored in memory exceeds the DRAM cache capacity, the DRAM cache miss rates increase significantly. Utilizing a DRAM cache in the HMS achieves high performance if a target application has a small working set. If not, the performance degrades substantially due to increased traffic to the slow memory (NVM), incurring an additional miss penalty. Moreover, the increased frequency of L2 TLB misses causes frequent page table walks, which incur a significant miss penalty for memory accesses. This leads to a substantial application throughput drop (up to 40%).

Applying PGO to optimizing memory management is a promising direction to reduce this memory-related slowdown, especially for those emerging workloads with large memory footprints. Among non-code segments, the heap segment has been identified as one of the major performance bottlenecks as it occupies the largest memory space in those applications [7, 19, 31, 43]. Prior work proposes several approaches to optimize heap management, such as accelerating the memory allocation path [32] and predicting memory object lifetime using machine learning techniques [43].

However, we still see the substantial potential to further improve the efficiency of heap management using PGO. PMU-based sampling is a powerful tool to augment PGO to identify those objects that benefit the most by allocating them to fast memory or huge pages. Modern processors provide a software interface to collect detailed information about an application’s memory accesses with low overhead through PMU sampling and data address profiling. As mentioned

Figure 3. The organization and operations of MaPHeA. MaPHeA (a) profiles heap allocations and memory accesses, (b) analyzes heap accesses to generate an allocation guide map, and (c) controls the heap object allocation automatically using the compiler-assisted allocation guide.

above, several previous works have proposed techniques similar to PGO, which exploit PMU sampling and data address profiling to optimize heap data placement for NUMA and HMSs [14, 45, 63, 66–68]. However, they have limitations such as manual programming effort for program transformation, insufficient usage of PMU’s rich memory hierarchy information, and limited use cases. A key challenge here is how to utilize these features effectively on PGO to optimize heap usage, hence improving the overall system performance.

3 Memory Hierarchy-Aware Profile-Guided Heap Allocation Framework

We propose MaPHeA, a lightweight Memory hierarchy-aware Profile-guided Heap Allocation framework. MaPHeA is a profile-guided optimization (PGO) framework that profiles statistics on heap memory data and automatically optimizes the heap allocation of a given program.

Figure 3 shows the overall structure and operations of MaPHeA. MaPHeA has three key components, which operate as follows. First, the memory profiler of MaPHeA collects architecture-specific heap access statistics (e.g., cache/TLB accesses) as well as information about heap memory allocations (e.g., size of each allocation unit) for the program being profiled (Section 3.1). Second, a proper heap allocation strategy for each allocation call site is determined (e.g., the type of memory/page that the object should be allocated to) by analyzing the profiling information (Section 3.2). Finally, the MaPHeA compiler backend modifies the program so that each heap allocation complies with the specified strategy obtained during the previous step (Section 3.3).

3.1 Profiling Heap Allocations and Accesses

To collect information on heap allocations and accesses, MaPHeA performs a profile run. During this profiling process, MaPHeA utilizes (i) hardware-assisted event-based sampling to collect heap access information and (ii) dynamic logging to collect information regarding heap allocation.

Figure 4. MaPHeA’s process of profiling dynamic allocations and their accesses. Wrapper functions in libmaphea.so call logAllocation() to log the allocation information, which is used to associate the heap data accesses with specific objects.

Collecting heap access information: MaPHeA configures performance monitoring units (PMUs) to monitor memory accesses. At every time interval defined by the sampling frequency (e.g., 1 KHz), the PMU logs a set of information (e.g., virtual address, memory latency, and timestamp) about each memory access happening within the time interval. When the profile run finishes, MaPHeA utilizes the Linux perf [18] tool to retrieve the recorded information. While the events that can be profiled may vary across platforms, we find that major processor vendors such as Intel [28], AMD [4], and IBM [25] support a common set of performance counters, which can be read out by the perf tool. Therefore, though we target Intel architectures in our prototype, we believe that other architectures also can be easily supported with only minimal modifications.

Collecting heap allocation information: To collect information on heap allocations, we implement a simple wrapper library (libmaphea.so) for the existing (de)allocation calls (e.g., malloc(), realloc(), free()) and link it to the target binary. The wrapper library first performs the conventional memory allocation (i.e., malloc()) and identifies the call site with __builtin_return_address() (see Figure 4).

Memory Access Information by Object ID					Memory Access Information by Callsite				
Object ID	...	DRAM access	Avg.lat	TLB miss	Call site	...	DRAM access	Avg.lat	TLB miss
0	...	1,501,617				...			
...	...	361,269				...			
41	...	1,541,427			0	...	3,043,044	539	301
N-1	...				K-1	...			
Sum up						...	52	7	2

Figure 5. The data structures of heap access log

Then `logAllocation(ptr, size, callsite)` is called to log the allocated memory address, the memory object size, and the call site address. The timestamp is also logged using the `clock_gettime()` system call. Other allocators (e.g., `new`) and deallocators (e.g., `free`) are handled similarly to log calls to them. The overhead of this logging mechanism is relatively small (less than 1% of the program runtime for the evaluated workloads). This is because memory allocations/deallocations are usually performed outside the critical loops of the program.

Associating heap accesses with call sites: Timestamps and addresses of the profiled accesses and (de)allocations are used to associate each memory access to a specific heap object and eventually to a call site. For each profiled access, a set of heap objects that contain the address in question are first identified. Then, by using the timestamp of the access, the profiler uniquely identifies the object whose lifetime (i.e., the time between its allocation timestamp to its deallocation timestamp) overlaps with the access timestamp. Figure 4 illustrates an example. The access to address `0x7ffec006e300` is associated with the heap object whose starting address is `0x7ffec00002c0` since the timestamp of the access (36.8) falls within the object's lifetime (from 36.6 to 83.3). Two data structures shown in Figure 5 are created using this mechanism. The first table summarizes access statistics for each *object*, and the second one collects them for each *call site*. These structures are passed to the next stage to determine per-call site allocation strategies (Section 3.2).

3.2 Heap Allocation Strategies

Profile data provide useful information about the application's heap usage. MaPHeA exploits this information to determine a proper allocation strategy for each heap allocation call site. We outline two allocation strategies in this section and further discuss their use cases in Section 4.

Data placement optimization: MaPHeA can apply a data placement optimization strategy for NUMA systems and emerging heterogeneous memory systems (HMSs) composed of two or more different types of memory devices. Emerging memory technologies such as 3D-stacked DRAMs [30, 38] provide much higher access bandwidth than conventional DRAM, whereas other memory technologies such as NVM [47] and managed DRAM [37] have longer access latency and lower access bandwidth than the conventional DRAM (e.g., DDR4 [29]). Thus, in an HMS, heap access latency and bandwidth may be highly different depending on

Figure 6. The operation flow of MaPHeA-augmented gcc. In the auto-profile pass, MaPHeA workflow replaces the default allocation calls with MaPHeA's allocation calls.

where an object is placed. Due to this difference, it is important to judiciously place each object to the right type of memory. MaPHeA aids this process by providing necessary information for all heap objects. For example, a set of heap objects that are frequently accessed and/or performance-critical can be placed in the faster memory. Section 4.1 studies how MaPHeA can be applied for data placement optimization in a real HMS composed of conventional DRAMs and Intel DC Persistent Memory Modules (DCPMM).

Huge-page optimization: MaPHeA can be used to optimize heap objects that frequently incur TLB misses. A TLB miss triggers a page table walk, which can often cause a substantial performance slowdown [8, 9]. One popular solution for minimizing this overhead is to use huge pages. With this mechanism, a single entry in TLB can reach a broader range of the virtual address space, reducing the number of TLB misses. However, naively increasing all page granularities is often suboptimal. For example, using a huge page for sparsely accessed data is inefficient because only a tiny fraction of the page ends up being accessed to waste physical memory (known as *internal fragmentation*). Also, utilizing a huge page for rarely accessed data has little performance impact as the number of TLB misses is small. As a result, it is crucial to carefully select a set of heap objects to utilize huge pages. With MaPHeA, a user can specify the policy to decide whether to use a huge page or not based on the collected information (e.g., TLB miss frequency, access sparsity, and allocation size) for each heap object. A use case of this allocation strategy is further elaborated in Section 4.2.

3.3 MaPHeA Compiler Backend

Once a proper allocation strategy (e.g., which type of memory to utilize, whether to use a huge page or not) is determined for each heap allocation site (Section 3.2), the MaPHeA compiler backend updates each allocation call site with a proper allocation strategy. For this purpose, we modify the Google AutoFDO framework (v0.19) [12]; the modified framework replaces each heap (de)allocation call in the target program with the corresponding custom memory (de)allocator call taking an additional flag of the allocation strategy as a parameter. Below, we describe implementation details.

Using the profiling information (Section 3.1), MaPHeA identifies the corresponding module and line of each call

GCov Header	Allocation Guide Map			
Filename table	file[0]	srcline[0]	guidance[0]	call site[0]
Function table	file[1]	srcline[1]	guidance[1]	call site[1]
...	...	...	...	...
Allocation list	file[N-1]	srcline[N-1]	guidance[N-1]	call site[N-1]

Figure 7. GCov file with MaPHeA’s allocation guide map. The allocation guide map includes file, code line, and guided allocation method for each call site.

site by inspecting the symbol table. Then, this call site information (represented in a (filename, line number) pair) along with the allocation strategy is stored in a table-like data structure and saved as a GCov file (i.e., a GCC-specific file format). This GCov file is read by the MaPHeA backend (i.e., the extended Google AutoFDO tool), which automatically replaces the corresponding allocation calls in the code into a custom allocation call with a strategy flag.

Specifically, we extend the *auto-profile* optimization pass of the Google AutoFDO tool, which is originally used to perform different PGOs (see Figure 6). The following steps are taken to replace memory allocation/deallocation calls with the appropriate flags.

1. MaPHeA allocation and deallocation functions (e.g., `maphea_malloc()`, `maphea_realloc()`) are included and linked to the target program.
2. The tree representation of the GIMPLE code is traversed to find the heap allocation call sites included in the allocation guide map (Figure 7).
3. The specified allocation call sites are replaced with MaPHeA allocation functions with specific flags as directed by the allocation guide map.

We list flags that MaPHeA currently supports as follows.

- **MAPHEA_DEFAULT:** The system-wide default allocation policy is used for this call site.
- **MAPHEA_HOT:** This call site allocates an object to the fast (hot) memory.
- **MAPHEA_HUGEPAGE:** This call site allocates an object to a huge page. Either 2MB or 1GB huge page (for Intel processors) is used depending on the object size.
- **MAPHEA_HOT_HUGEPAGE:** This flag is a combination of both **MAPHEA_HOT** and **MAPHEA_HUGEPAGE**.

Most of the above flags are derived from the available flags from `mmap` or `mbind` system calls, which conventional memory allocators like `malloc` internally utilizes to perform memory allocations. Such flags enable the system to allocate memory on a specific device or utilize a huge page for a particular allocation. Internally, MaPHeA allocator exploits `memkind` [11] to perform the memory allocation based on the designated allocation strategy. Note that the MaPHeA optimizations and the existing AutoFDO optimizations can be composed to apply together. Also, if necessary, it is possible to implement MaPHeA in a separate pass.

Figure 8. (a) HMS’s configurations and (b) examples of huge page allocation according to the properties of objects.

4 MaPHeA Guided Heap Allocation: Two Use Cases

In this section, we apply MaPHeA to two use cases, an *HMS* and *selective huge pages*, to optimize heap allocation. For each case, we first diagnose the sources of inefficiency, followed by how MaPHeA addresses them by controlling heap object allocation.

4.1 Use Case 1: Heterogeneous Memory System

In this section, we apply MaPHeA to a real HMS to optimize memory allocation. Figure 8(a) shows the target HMS with both Intel Optane DC Persistent Memory Modules (DCPMMs) [41] and DDR4 DRAM modules. Then, using MaPHeA, we identify objects to be allocated in the fast memory (DRAM) to control heap object allocation accordingly.

HMS composed of both DRAM and DCPMM: To showcase the optimization on an HMS, we configure an HMS with DCPMM and DDR4 DRAM modules. DCPMM provides substantially higher density than the conventional DRAM. However, it is known to have an over $2\times$ longer latency and $3\times$ lower read bandwidth than DRAM [21]. To utilize DCPMM efficiently, Intel CPU [5] supports two configurations for DCPMM (Figure 8(a)): Memory Mode and App Direct Mode. Memory Mode uses DCPMM as volatile main memory and DRAM as a (software-transparent) cache of DCPMM. App Direct Mode uses both DRAM and DCPMM as separate memory regions. In the rest of this paper, we refer to Memory Mode as *DRAM cache mode* and App Direct Mode as *flat-memory mode*.

The DRAM cache mode is implemented as a direct-mapped write-back cache, which does not require any software modifications [21]. In the DRAM cache mode, a memory request is first directed to the DRAM cache and checks if it hits there. If so, the data is immediately returned to the CPU, taking 80.6 ns in the evaluated system (see Table 1); if not, the request is then sent to the DCPMM. However, it has high DCPMM access latency (242.7 ns) due to the tag lookup overhead in the DRAM cache, so-called *DRAM cache miss penalty*, which leads to performance degradation for applications with a large working set and low locality. This mode performs well for applications that have a small working set and high locality.

Figure 9. The portion of LLC misses and average latency of two specific heap object of the bfs application in the GAP benchmark suite [10] running with a web dataset, and the degree of performance improvement when each of two heap objects is allocated to DRAM instead of DCPMM.

The flat-memory mode uses DCPMM as either persistent storage or NUMA memory node by exposing it to the physical memory space. We can access DCPMM in the flat-memory mode using load/store instructions through direct access (DAX) [27, 60] functions. As DCPMM is directly accessed without going through DRAM, this mode shows lower DCPMM access latency (175.7 ns) than DRAM cache mode.

However, the access latency of DCPMM is still over 2× longer than that of DRAM, which takes almost the same 80.8 ns as the DRAM cache mode. If the data of a program is not carefully allocated, more accesses are directed to DCPMM compared to the DRAM cache mode, leading to a longer average memory access latency. Therefore, the user should characterize the memory access pattern first, and then modify the program to allocate data to the proper memory type. The user can modify the program manually by using API such as memkind [11] and PMDK [27] to allocate each data object to either DRAM and DCPMM. However, this process entails the user effort of manual modifications. Instead, MaPHeA automates this process using a production compiler (gcc).

Applying MaPHeA to HMS: To select data objects to allocate to DRAM, MaPHeA exploits memory access frequency, size, and memory access latency from the access information to the heap segment. It first estimates memory access frequency using LLC miss count, because most of the LLC misses lead to main-memory accesses. Also, to efficiently utilize the limited capacity of fast memory, MaPHeA normalizes this frequency by the size of the heap object to calculate the per-byte access count.

One distinct design decision for priority setting is to take the *average latency* of load instructions into account. Even for accessing data from the same memory hierarchy level, memory access latency can be affected by several factors such as queuing delay. Therefore, among those heap objects having the same main-memory access frequency, it is advantageous to allocate the object whose average memory access latency is longer than the others, into the fast memory. To verify this, we had an experiment with bfs of the GAP benchmark suite (see Figure 9). The two heap objects of pvector:29 and reader:292 have the similar frequency of LLC misses but different average latencies. Thus, we measured

the performance gains when allocating the objects to DRAM over DCPMM. Allocating reader:292 (with a longer average latency) to DRAM results in over 10% higher throughput gains than allocating pvector:29 to DRAM. Therefore, considering the average latency, we devise a heuristic model that prioritizes the data to be allocated to DRAM through the following equation (higher ones being prioritized).

$$Alloc\ priority = \frac{(LLC\ miss\ count)}{(data\ size)} \times (average\ latency) \quad (1)$$

MaPHeA sorts heap objects according to their priority and allocates the ones with the highest priority to the fast memory up to the predefined capacity threshold. The MAPHEA_HOT flag falls back to the slow memory when the available fast memory capacity is not enough.

4.2 Use Case 2: Selective Huge-Page Utilization

Using huge pages can be effective in alleviating the performance penalty due to frequent TLB misses; however, depending on usage patterns of huge page, either the memory use, performance, or both of a target application can be adversely affected. Therefore, we introduce a method that can efficiently optimize TLB usage by applying MaPHeA.

How to use huge pages to mitigate TLB miss penalty:

We can use huge pages either i) manually by specifying the candidate heap object through modifying an application code, or ii) transparently by managing huge pages in an operating system. Transparent Huge Page (THP) support by Linux takes the latter (runtime) approach and works as follows. When a page fault occurs, if there exists an available physical address range spanning the size of a huge page, THP greedily allocates a huge page. If there is no contiguous physical address region available that is large enough to cover a huge page, huge pages are reclaimed by memory compaction either synchronously (the application requesting a huge page is stalled until a huge page is reclaimed) or asynchronously (by calling a background page-reclamation process). This mechanism is effective when allocating an object that causes frequent TLB misses fits in a huge page, such as object A in Figure 8-(b). However, in the case of allocating an object whose size does not fit in a huge page, such as object B in Figure 8-(b), THP might unnecessarily increase memory consumption (footprint) due to internal fragmentation (also called memory bloat). For security, Linux must zero the pages being reclaimed before returning them to user applications, which can cause a latency spike.

The memory bloat and performance problems of THP have been reported in previous studies [36, 55]. Due to these issues, the use of THP is restricted in databases and key-value stores despite the advantages of reducing the TLB miss penalty by exploiting huge pages. To overcome these limitations of THP, a user must analyze memory access patterns and modify the target application to allocate huge pages to the objects that

Figure 10. Comparison of (a) the performance, working set size, and (b) access distribution between *DRAM cache*, *DRAM first-touch*, *OS-migration*, and *MaPHeA* on the multi-programmed GAP and the Redis applications. The results are normalized by those of the *DRAM cache* configuration.

Table 1. System configuration

Processor	Intel Xeon Platinum 8260
Last-level cache	35.75 MB shared, 11-way
Main memory	
HMS	2×(8GB DRAM, 128GB DCPMM)
Selective huge-page	6×16GB DRAM
Operating system	Linux kernel v4.20.0

cause frequent TLB misses but have a low risk of memory bloat. Using MaPHeA, this process can be automated.

Applying MaPHeA to huge-page optimization: To select heap objects to be allocated to huge pages, MaPHeA uses each object’s TLB miss frequency and the allocated size. First, the TLB miss frequency is estimated using the L2 TLB miss count. The miss penalty when an L1 TLB miss is followed by an L2 TLB hit is just several CPU cycles and hence does not significantly affect performance. By contrast, as shown in Figure 8-(b), an L2 TLB miss causes a page table walk, which incurs a significant miss penalty and has a great impact on performance. Therefore, MaPHeA only considers L2 TLB misses. Second, to take advantage of exploiting huge pages maximally, the memory bloat shown in object B of Figure 8-(b) must be minimized. Therefore, we sort the heap objects by the TLB miss frequency normalized to the object size, which is similar to Equation 1, and allocate the ones with the highest return-on-investment to huge pages.

5 Evaluation

5.1 Experimental Setup and Methodology

System configuration: To exemplify the effectiveness of the MaPHeA framework, we configured an evaluation system as shown in Table 1. Each use case uses a different main-memory configuration. For the data placement optimization use case, we used DRAM and DCPMM to construct an HMS. Specifically, to match the working set size provided by the evaluated benchmarks, we scaled down the capacity of the system by installing 8GB DRAM and 128GB DCPMM in each of two integrated memory controllers (iMC), respectively. To use DCPMM as a NUMA node, we applied the

“PMEM NUMA node and hotness accounting/migration” kernel patch [40]. For the selective huge-page utilization use case, we connected a single 16GB DRAM DIMM to each of the six memory channels in the system.

Methodology: We quantified the effectiveness of MaPHeA using the GAP benchmark suite [10] and Redis [59]. To evaluate the performance of Redis, we used YCSB [16]. Among applications and datasets in GAP, we selected Breadth-First Search (bfs), PageRank (pr), and Connected Components (cc) as benchmarks and Kronecker 27 (kron27 [48]), twitter [35], and web [17] as input datasets.

In evaluating data placement optimization, as the HMS with DCPMM targets the workloads requiring large memory capacity, we ran each GAP application in eight processes to form a memory footprint large enough to prevent all data from fitting in DRAM. All the processes execute two threads, and all the threads are assigned to different cores, respectively. In the case of Redis, we loaded 40M key-value pairs to the Redis server and generated 40M read-heavy requests using YCSB. We also used Redis to evaluate selective huge-page utilization. We loaded 6M key-value pairs to the server and generated 10M update heavy requests. All the applications are compiled with -O2 option.

In data placement optimization evaluation, we compared the performance of MaPHeA with DRAM cache mode, which is a hardware-managed scheme, and the software-managed memory optimizer framework [26] developed by Intel. Intel *memory optimizer* is a framework that periodically identifies hot/cold pages used in the user-space (*page scan phase*) and migrates hot pages to the fast DRAM region and cold pages to the slow DCPMM region (*migration phase*). We configured the page scan and migration options in the *memory optimizer* to the default suggested values. In the rest of this paper, we refer to *memory optimizer* as *OS-migration*.

5.2 MaPHeA on Data Placement Optimization

We first evaluated the performance and memory access characteristics of MaPHeA on the real HMS. Specifically, we compared the performance of four different schemes, *DRAM*

Figure 11. Comparison of the average latency and the tail latency between Linux’s *THP* (grey color) and *MaPHeA* (black color) on Redis with the YCSB workloads.

cache, *DRAM first-touch*, *OS-migration*, and *MaPHeA*. *DRAM first-touch* is a policy that allocates data to DRAM as long as its capacity is available and then allocates the rest of the data in DCPMM. Figure 10-(a) shows the relative performance for four evaluated schemes across different workloads and the working set size (WSS) of each workload. As shown in the figure, *MaPHeA* achieves better performance than *DRAM cache*, *DRAM first-touch*, and *OS-migration* on all the evaluated workloads. On average, *MaPHeA* achieves 56.0%, 87.2%, and 52.3% better performance than *DRAM cache*, *DRAM first-touch*, and *OS-migration*, respectively. Using the flat memory mode with a naïve allocation policy (i.e., *DRAM first-touch*) shows lower performance than *DRAM cache* in *cc* and *bfs*. *OS-migration*, which dynamically changes data placement in the flat memory mode, achieves better or at least similar performance compared to *DRAM cache* on all workloads. However, its additional performance gain is limited due to the page table scan and page migration process. In contrast, *MaPHeA* achieves substantially better performance than both *DRAM cache* and *OS-migration* configurations.

Figure 10-(b) shows the portion of DRAM and DCPMM access on *DRAM cache* and *MaPHeA*. *MaPHeA* reduces the total main-memory traffic by 45.2% compared to the *DRAM cache* on average. This is because *DRAM cache* suffers from additional memory traffic due to DRAM cache misses. Specifically, for each DRAM cache miss, two extra memory accesses (read data from DCPMM, write data to DRAM) are necessary. Furthermore, in many of the workloads, *MaPHeA* reduces the number of DCPMM access. This indicates that *MaPHeA* can effectively place hot data to DRAM.

It is also possible to utilize DCPMM as a swap space for the system. In general, the operation of this model is similar to that of the DRAM cache in a high level, except for the case where a large portion of the heap object has a very short lifetime (i.e., data are deallocated before swapped out). However, we could not find such cases in our evaluated workloads, and even if the case occurs, it would likely have a similar performance with the *DRAM cache*.

5.3 MaPHeA on Selective Huge-Page Utilization

Next, we evaluated the effectiveness of *MaPHeA* in selectively exploiting huge pages using Redis, a popular in-memory

key-value store. In particular, we compared the performance of three different methods: 4KB page only implementation (*baseline*), the transparent huge page implementation (*THP*), and *MaPHeA*. Here, the average and tail latency values are normalized to the results of *baseline*.

Figure 11 shows the average and tail latency improvements over the *baseline*. *MaPHeA* reduces the average latency by 12% in the read workload and 14% in the update workload compared to *baseline*, while *THP* reduces the average latency of read and update operations by just 1%. *THP* and *MaPHeA* reduce the number of L2 TLB misses by 30% and 69% via utilizing huge pages, respectively. Therefore, *MaPHeA* shows significant performance gains due to the reduced TLB miss frequency, while *THP* shows rather modest performance improvement. This discrepancy is due to the increased frequency of memory compaction and zeroing operations to reclaim huge pages securely, as described in Section 4.2. The performance impact of this compaction can be seen more clearly in the tail latency results. In *THP*, a latency spike occurs due to compaction, which significantly deteriorates tail latency to offset the performance gain obtained by reducing the TLB miss frequency. By contrast, *MaPHeA* achieves better performance while minimally affecting the tail latency.

We also measured the memory usage on the *baseline*, *THP*, and *MaPHeA*. On *THP*, the greedy allocation policy causes a significant memory space overhead (bloat) of 26% compared to *baseline*, leading to a deficiency of free memory space. By contrast, *MaPHeA* conservatively allocates huge pages to the heap objects, resulting in a memory space overhead of about merely 2%. This difference indicates that *MaPHeA* can improve performance by using huge pages while minimizing the latency spikes and memory bloats of *THP*.

5.4 Sensitivity Studies

We evaluated several factors that could affect the effectiveness of *MaPHeA* in the HMS case study.

Datasets and effective LLC capacity: According to the dataset and hardware environment, the main-memory access characteristics to the heap object can be changed, which causes a statistical error in the access distribution between the profile time and the runtime after recompilation. Therefore, we observed the heap object access distribution on the various datasets and shared LLC capacity that can affect statistical error. To control the effective LLC capacity, we changed the number of available LLC-ways from 3 ways (9.75 MB) to 9 ways (29.25 MB) by using Intel Cache Allocation Technology [28]. As shown in Figure 12, even if the effective LLC capacity decreases according to the number of available LLC ways, the order of hot heap objects is rarely changed in most cases. Also, even in the workloads that show the order changed, such as the *bfs-kron27* workload, the proportion of DRAM access is still higher than that of the DCPMM because the difference in order is not severe. However, there is a difference in access distribution according to datasets.

Figure 12. The main-memory access distribution of heap object and relative LLC miss count according to effective LLC capacity. Data 1 to 8 represent eight heap objects with the largest numbers of memory accesses on each application; the data with slashes represent the data guided to be allocated to DRAM by *MaPHeA*.

Figure 13. Comparison of performance improvement according to differences in the training and test datasets on GAP applications. The training and test datasets applied to the application are denoted as train-test. The results are normalized by performance of the *DRAM* cache scheme.

For example, in *bfs*, data (heap object) 7 is accessed more frequently when using the twitter and web dataset compared to the *kron27* dataset. Therefore, the optimized *bfs* binary using a profiled result with the *kron27* dataset cannot optimally allocate heap objects on the twitter or web datasets. Despite this, because data distribution does not change severely according to the dataset changes, allocation through *MaPHeA* can still produce better performance.

Performance robustness: The difference between the training dataset used at profile time and the test dataset used at run time after recompilation may influence the degree of performance improvement of a program. Therefore, we observed how robust the performance improvement is when applying different datasets for training and test. We applied the train-test input combination of three datasets *kron27*, *twitter*, and *web* (e.g., *twitter-web* and *twitter* become a train dataset, *web* becomes a test dataset) to *cc*, *pr*, and *bfs* applications and compared the performance of the *DRAM* cache scheme. As shown in Figure 13, when each test dataset is applied to a binary optimized for a different training dataset, the performance improvement is almost the same as when applied to a binary optimized for the same train dataset. As shown in Figure 12, the data access distribution of the application is not severely changed according to the

Figure 14. The trend of access distribution and profiling overhead according to the sampling frequency, which decreases from 100 KHz to 1 KHz. Data 1 to 5 represent five heap objects with the largest numbers of memory accesses.

dataset change, and the selected hot objects are the same. Therefore, there is little change in performance improvement according to difference between the training dataset and the test dataset.

Sampling frequency: The precision of PEBS-based profiling improves as the sampling frequency increases, but the performance overhead also increases. Therefore, we observed changes in performance overhead of *MaPHeA*'s profiling and heap object access distribution. Figure 14 shows the results when we decreased sampling frequency from 100 KHz to 1 KHz on *bfs-kron27* and *cc-kron27* workloads. Even if the sampling frequency decreases, the heap object access distribution is almost unchanged. On the other hand, the decrease in sampling frequency from 100 KHz to 1 KHz reduces the performance overhead by 5.8%. Notably, the performance overhead of sampling with 1 KHz frequency is 0.5%. Therefore, we can reduce the performance overhead for profile heap access information without impacting the optimization result by decreasing the sampling frequency of *MaPHeA*.

6 Related Work

The previous PGO researches were analyzed in Section 2. We briefly compare *MaPHeA* with studies targeting the data placement optimization for HMSs and huge-page utilization.

OS-level data management in HMSs: There have been various studies for data placement in HMSs, which is managed by OS. HeteroOS [33], Thermostat [2], and Nimble [69] focused on flat memory and proposed the OS-based methods that analyze page granularity memory access at runtime and migrate pages to an appropriate memory device. It is transparent to applications and can consider the runtime behavior of the systems. However, it has significant runtime overhead due to the process of tracking accesses to each page.

User-level data management in HMSs: There have been numerous application-based studies that optimize data placement in HMSs. LLAMA [20] and Khaldi et al. [34] proposed methods that perform object classification and placement by analyzing instructions at compile time. However, it is difficult to accurately predict the access distribution over the memory hierarchy occurring at runtime. X-mem [19] decides data placement by analyzing the memory access patterns through dynamic instrumentation, which causes significant profiling overhead. MOCA [50] profiles the memory access information through simulation to consider the memory hierarchy, but conducting microarchitectural simulation is slower than using the PMU sampling. Servat et al. [63], ATMem [14], Unimem [67], and Tahoe [68] proposed data allocation or migration at runtime based on PMU sampling results. However, they require designating the target objects to be monitored manually and cause significant performance overhead in situations where the migration occurs frequently.

Software-based solutions to mitigate TLB miss penalty: Many studies have proposed to mitigate the problems of the THP implementation [36, 51, 55], by suggesting ways to trade the degree of memory bloat off against that of performance impact. Our selective huge-page utilization case study is inspired by these studies, using the average/tail latency and memory bloat as evaluation criteria; however, as opposed to our PGO-based MaPHeA, these studies, including Ingens [36] and HawkEye [55], proposed runtime-based huge-page management schemes. Magee et al. [44] predict TLB utilization by analyzing data-reuse patterns at compile time and guides huge-page allocation through heuristics. In contrast, MaPHeA decides heap objects to huge pages based on TLB miss information collected through profiling.

PGO techniques for managed languages: There have been several PGO studies that orchestrate memory resources in managed languages. Using the garbage collector (GC) in Java Virtual Machine (JVM), MS Delta [1] identifies traversal paths where cache misses occur frequently in linked data structures and inserts prefetch instructions through JIT compilation. HCSGC [70] improves cache locality and prefetching efficiency by dynamically relocating objects in the order that mutators access them and separating frequently accessed objects from infrequently accessed objects in GC. Akram et al. [3] proposed a mechanism to minimize the wear-out of NVM by placing write-intensive objects in DRAM and read-intensive objects in NVM. Olson et al. [52] analyzed

the power consumption of DRAM that change according to the data object usage patterns. Based on this analysis, they proposed a data placement method that can improve the energy efficiency of the DRAM. Panthera [64] classified objects according to the characteristics of the data abstraction (e.g. Spark's resilient distributed dataset (RDD)) of a big data application and proposed a data placement method to improve energy efficiency in HMS. These studies do not solve the various problems occurring across the memory hierarchy, and their optimization target is different from that of MaPHeA, which focuses on improving the memory performance of the applications written with unmanaged languages.

7 Conclusion

We have proposed MaPHeA, a lightweight memory hierarchy-aware profile-guided heap allocation framework applicable to high-performance computing (HPC) and embedded systems. To the best of our knowledge, MaPHeA is the first profile-guided optimization technique that optimizes the heap object management across all memory hierarchies with a low-performance overhead and low programming effort. MaPHeA achieves a low profiling overhead by sampling hardware events and logging dynamic allocations. Also, to manipulate the memory allocation based on profile result analysis with minimal user effort, we implemented the functions to transfer allocation guidelines to the compiler and modify the intermediate representation to call the guided memory allocation inside the compiler. As case studies, we applied it to optimizing heap object allocation in an emerging DRAM-NVM heterogeneous memory system for HPC and embedded systems and in selective huge-page utilization. In the first study, by guiding the heap objects suitable to be allocated to DRAM without manual modification, MaPHeA improved the performance of memory-intensive graph-processing and Redis by 56.0% on average over the DRAM cache mode. The second study identified large heap objects with frequent TLB misses and allocated those to huge pages, through which MaPHeA improved the performance of read and write operations of Redis by 10.6% on average over the configuration using the transparent huge-page implementation of Linux.

Acknowledgments

This work was supported by the R&D program of MOTIE/KEIT (10077609) and the Future Semiconductor Device Development Program funded by MOTIE and KSRC (10080613).

References

- [1] A-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. 2004. Prefetch Injection Based on Hardware Monitoring and Object Metadata. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/996841.996873>
- [2] N. Agarwal and T. F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In

- Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/3037697.3037706>
- [3] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/3296979.3192392>
- [4] AMD. 2017. AMD64 Architecture Programmer's Manual Volume 2: System Programming. <https://www.amd.com/system/files/TechDocs/24593.pdf>.
- [5] M. Arafat, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora. 2019. Cascade Lake: Next Generation Intel Xeon Scalable Processor. *IEEE Micro* 39 (2019). <https://doi.org/10.1109/MM.2019.2899330>
- [6] ARM. 2019. ARM® ARM Architecture Reference Manual Armv8, for Armv8-A Architecture Profile. <https://documentation-service.arm.com/static/60119835773bb020e3de6fee?token=>.
- [7] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. 2018. Memory Hierarchy for Web Search. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*. <https://doi.org/10.1109/HPCA.2018.00061>
- [8] T. W. Barr, A. L. Cox, and S. Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. <https://doi.org/10.1145/2024723.2000101>
- [9] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. <https://doi.org/10.1145/2508148.2485943>
- [10] S. Beamer, K. Asanović, and D. Patterson. 2015. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619* (2015).
- [11] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond. 2015. *Memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*. United States. National Nuclear Security Administration.
- [12] D. Chen, T. Moseley, and D. X. Li. 2017. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. <https://doi.org/10.1145/2854038.2854044>
- [13] D. Chen, N. Vachharajani, R. Hundt, S. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. 2013. Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations. *IEEE Trans. Comput.* 62 (2013). <https://doi.org/10.1109/TC.2011.233>
- [14] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren. 2020. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. <https://doi.org/10.1145/3368826.3377922>
- [15] T. Chilimbi, M. D. Hill, and J. R. Larus. 1999. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/301631.301633>
- [16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. <https://doi.org/10.1145/1807128.1807152>
- [17] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* (2011). <https://doi.org/10.1145/2049662.2049663>
- [18] A. C. de Melo. 2009. Performance counters on Linux. In *Linux Plumbers Conference*.
- [19] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. <https://doi.org/10.1145/2901318.2901344>
- [20] D. Greenspan. 2019. LLAMA - Automatic Memory Allocations: An LLVM Pass and Library for Automatically Determining Memory Allocations. In *Proceedings of the International Symposium on Memory Systems*. <https://doi.org/10.1145/3357526.3357534>
- [21] T. Hirofuchi and R. Takano. 2019. The Preliminary Evaluation of a Hypervisor-based Virtualization Mechanism for Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1907.12014* (2019).
- [22] J. Hu, M. Xie, C. Pan, C. J. Xue, Q. Zhuge, and E. H. . Sha. 2015. Low Overhead Software Wear Leveling for Hybrid PCM + DRAM Main Memory on Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23 (2015). <https://doi.org/10.1109/TVLSI.2014.2321571>
- [23] J. Hu, Q. Zhuge, C. J. Xue, W-C. Tseng, and E. H. . Sha. 2013. Software Enabled Wear-Leveling for Hybrid PCM Main Memory on Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. <https://doi.org/10.7873/DATE.2013.131>
- [24] J. Hubicka. 2005. Profile Driven Optimisations in GCC. In *Proceedings of GCC Summit*.
- [25] IBM. 2018. POWER9 Performance Monitor Unit User's Guide. https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf.
- [26] Intel. 2018. Memory Optimizer. <https://github.com/intel/memory-optimizer>.
- [27] Intel. 2018. Persistent Memory Documentation. <https://docs.pmem.io/persistent-memory/>.
- [28] Intel. 2019. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3B: System Programming Guide. <https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3b-system-programming-guide-part-2>.
- [29] JEDEC. 2012. JEDEC Standard: DDR4 SDRAM.
- [30] JEDEC. 2015. High Bandwidth Memory (HBM) DRAM.
- [31] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G-Y. Wei, and D. Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169. <https://doi.org/10.1145/2749469.2750392>
- [32] S. Kanev, S. L. Xi, G-Y. Wei, and D. Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/3037697.3037736>
- [33] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. 2017. Heteroos: Os Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. <https://doi.org/10.1145/3079856.3080245>
- [34] D. Khaldi and B. Chapman. 2016. Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing. In *Third Workshop on the LLVM Compiler Infrastructure in HPC*. <https://doi.org/10.5555/3018869.3018871>
- [35] H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th international conference on World wide web*. <https://doi.org/10.1145/1772690.1772751>
- [36] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. <https://doi.org/10.5555/3026877.3026931>
- [37] S. Lee, B. Jeon, K. Kang, D. Ka, N. Kim, Y. Kim, Y. Hong, M. Kang, J. Min, M. Lee, C. Jeong, K. Kim, D. Lee, J. Shin, Y. Han, Y. Shim, Y. Kim, Y. Kim, H. Kim, J. Yun, B. Kim, S. Han, C. Lee, J. Song, H. Song, I. Park, Y. Kim, J. Chun, and J. Oh. 2019. 23.4 A 512GB 1.1 V Managed DRAM Solution with 16GB ODP and Media Controller. In *Proceedings of the 2019 IEEE International Solid-State Circuits Conference*. <https://doi.org/10.1109/ISSCC.2019.8662367>
- [38] J. Leidel and R. C. Murphy. 2015. Hybrid Memory Cube System Interconnect Directory-Based Cache Coherence Methodology. US Patent App. 14/706,516.

- [39] Linux. 2009. Transparent Hugepages. <https://lwn.net/Articles/359158>.
- [40] Linux. 2018. PMEM NUMA Node and Hotness Accounting/Migration. <https://lkml.org/lkml/2018/12/26/138>.
- [41] L. Looi and J. X. Jianping. 2019. Intel Optane Data Center Persistent Memory. In *2019 IEEE Hot Chips 31 Symposium*.
- [42] C-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. 2004. Ispike: A Post-Link Optimizer for the Intel/spl reg/Itanium/spl reg/Architecture. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*. <https://doi.org/10.1109/CGO.2004.1281660>
- [43] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/3373376.3378525>
- [44] J. Magee and A. Qasem. 2009. A Case for Compiler-Driven Superpage Allocation. In *Proceedings of the 47th Annual Southeast Regional Conference*. <https://doi.org/10.1145/1566445.1566553>
- [45] J. Marathe and F. Mueller. 2006. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. <https://doi.org/10.1145/1122971.1122987>
- [46] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. 2015. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture*. <https://doi.org/10.1109/HPCA.2015.7056027>
- [47] Micron. 2016. 3D XPoint Technology. <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [48] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. 2010. *Introducing the Graph 500*. Technical Report. Sandia National Laboratories.
- [49] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C-Y. Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. <https://doi.org/10.1145/2807591.2807626>
- [50] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. Coskun. 2018. MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*. <https://doi.org/10.1109/IPDPS.2018.00042>
- [51] J. Navarro, S. Iyer, P. Druschel, and A. Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. <https://doi.org/10.1145/844128.844138>
- [52] M. B. Olson, J. T. Teague, D. Rao, M. R. JANTZ, K. A. Doshi, and P. A. Kulkarni. 2018. Cross-Layer Memory Management to Improve DRAM Energy Efficiency. *ACM Transactions on Architecture and Code Optimization* 15 (2018). <https://doi.org/10.1145/3196886>
- [53] G. Ottoni and B. Maher. 2017. Optimizing Function Placement for Large-Scale Data-Center Applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. <https://doi.org/10.5555/3049832.3049858>
- [54] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. 2019. Bolt: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. <https://doi.org/10.5555/3314872.3314876>
- [55] A. Panwar, S. Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/3297858.3304064>
- [56] K. Pettis and R. C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/93542.93550>
- [57] M. K. Qureshi and G. H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. <https://doi.org/10.1109/MICRO.2012.30>
- [58] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. <https://doi.org/10.1145/1555754.1555760>
- [59] Redis. 2020. redis.io. <https://redis.io>.
- [60] A. Rudoff. 2017. Persistent Memory Programming. *Login: The Usenix Magazine* (2017).
- [61] J. Savage and T. M. Jones. 2020. HALO: Post-Link Heap-Layout Optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. <https://doi.org/10.1145/3368826.3377914>
- [62] M. L. Seidl and B. G. Zorn. 1998. Segregating Heap Objects by Reference Behavior and Lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/384265.291012>
- [63] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*. <https://doi.org/10.1109/CLUSTER.2017.50>
- [64] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/3314221.3314650>
- [65] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*. <https://doi.org/10.1109/HPCA.2014.6835958>
- [66] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu. 2018. Profddp: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *Proceedings of the 2018 International Conference on Supercomputing*. <https://doi.org/10.1145/3205289.3205320>
- [67] K. Wu, T. Huang, and D. Li. 2017. Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. <https://doi.org/10.1145/3126908.3126923>
- [68] K. Wu, J. Ren, and D. Li. 2018. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task-Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. <https://doi.org/10.5555/3291656.3291698>
- [69] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. <https://doi.org/10.1145/3297858.3304024>
- [70] A. M. Yang, E. Österlund, and T. Wrigstad. 2020. Improving Program Locality in the GC Using Hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. <https://doi.org/10.1145/3385412.3385977>
- [71] Z. Zhang, Z. Jia, P. Liu, and L. Ju. 2016. Energy Efficient Real-Time Task Scheduling for Embedded Systems with Hybrid Main Memory. *Journal of Signal Processing Systems* 84 (2016). <https://doi.org/10.1109/RTCSA.2014.6910524>