# Evaluating the Effectiveness of Program Data Features for Guiding Memory Management

T. Chad Effler
Brandon Kammerdiener
Michael R. Jantz
{teffler,bkammerd}@vols.utk.edu,mrjantz@utk.edu
University of Tennessee
Knoxville, Tennessee, USA

Saikat Sengupta
Prasad A. Kulkarni
saikatsengupta92@ku.edu
kulkarni@ittc.ku.edu
University of Kansas
Lawrence, Kansas, USA

Kshitij A. Doshi
kshitij.a.doshi@intel.com
Intel® Corporation
Chandler, Arizona, USA

Terry Jones
trjones@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

## ABSTRACT

Recent trends have led to the adoption of larger and more complex memory systems, often with multiple tiers of memory performance within the same platform. To utilize complex memory systems efficiently, current data management strategies must be altered to map usage demands to the underlying hardware. Applications, as the generators of memory accesses, are well-suited to guide this process, but building and maintaining separate source code versions for different memory systems is not feasible in most cases. One potential solution is to employ automated program profiling and analysis to facilitate the production of application-based guidance. By attaching memory usage information to static or lightweight program features, compilers and runtime systems can generate fine-grained guidance without additional efforts from users or developers. Recent works have employed this approach with some success, but it is not clear which program features are most useful for guiding data management.

This work evaluates the effectiveness of using different program data features to predict memory usage and guide memory management. It employs a custom set of simulation tools, based in the Intel® Pin framework, to collect and analyze the usage characteristics of application data associated with common program data features, such as allocation sites, types, and context. The results show that even relatively simple features, such as object size, are useful for selecting data with similar usage properties, but finer-grained features, such as the instructions that access a particular object, are often much more effective. Additionally, this work evaluates the performance of using different data features and different program inputs to guide data placement on a heterogeneous memory platform with a limited amount of high performance memory.


*MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA*

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357537

## CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → *Runtime environments.*

## KEYWORDS

Profiling, analysis, heterogeneous memory, performance

**ACM Reference Format:**
T. Chad Effler, Brandon Kammerdiener, Michael R. Jantz, Saikat Sengupta, Prasad A. Kulkarni, Kshitij A. Doshi, and Terry Jones. 2019. Evaluating the Effectiveness of Program Data Features for Guiding Memory Management. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357526.3357537

## 1 INTRODUCTION

Recent years have witnessed the emergence of several new memory technologies, each with their own advantages and drawbacks. For example, "on-package" or "die-stacked" memories [6, 17] enable higher bandwidth, but are only available in limited capacity, while storage class memories (SCMs) [19, 33] can provide non-volatile memory storage with much larger capacity, but have less bandwidth and longer latencies for reads and writes [16]. As a result, many high-end computer systems have begun to incorporate multiple types of memory devices in separate storage tiers. Such *heterogeneous* memory systems sacrifice simplicity and uniform access for the potential to achieve the benefits of multiple memory technologies simultaneously.

Existing data management strategies need to be altered to utilize heterogeneous memories efficiently, but doing so remains a significant research challenge. One common approach empowers application software with facilities to assign and migrate data objects into different device tiers as needed [1, 12, 38, 46]. These fine-grained controls allow developers to coordinate tier assignments with data allocation and usage patterns, and can potentially expose new and powerful efficiencies. However, such software-directed approaches require expert knowledge and source code modifications to adapt each application's data usage to the available memory hardware.

Previous works have sought to automate the production of memory management guidance through the use of program profiling

and analysis. Due to the large volume of memory addresses and accesses generated by most applications, data profiles are typically combined and associated with some program features that are either static or easy to detect during execution. For example, the application might combine guidance for all objects of the same size or allocated from the same source code instruction. In addition to summarizing memory behavior, this approach can potentially make guidance more efficient if the selected features accurately distinguish data with similar usage. Specifically, if the application is able to determine that new or recently allocated data has similar features as data that has already been profiled, it could apply guidance to the new data prior to its first use. Indeed, many recent works have deployed this strategy on both traditional and heterogeneous memory systems for a variety of purposes, including to: allocate high bandwidth data into a capacity-constrained high performance tier [23, 48, 51], steer writes to memory devices with better endurance [5], control memory power consumption [36, 47], reduce garbage collection (GC) effort [9, 13, 29], as well as for several other purposes described in Section 2.

While there are many potential benefits to guiding memory management with application intelligence, guidance that mispredicts memory usage can reduce or completely negate the effectiveness of this approach. Constructing accurate guidance for feedback-directed optimizations (FDOs) in memory systems is particularly challenging due to the need to exert fine-grained control over a large amount of unstructured and often transient program data. Linking memory usage to data features can help address these challenges, but certain features are more useful than others for guiding memory management. Some features might be very useful for distinguishing certain behaviors, but are too expensive to determine during execution or are only associated with a small amount of data. Other features might be easier to profile, but overlap data with dissimilar usage. For example, it might be intuitive to aggregate profiles of objects allocated as the same data type, but objects of the same type may or may not exhibit similar access intensities or utilize caches with the same efficiency.

This work investigates the effectiveness of using different program data features to characterize memory usage and guide memory management. It employs detailed simulations of full-scale applications to collect usage statistics for individual data objects, including: physical capacity requirements, data reads and writes, cache utilization, and data lifetime. Objects and their usage profiles are then categorized and aggregated across a variety of data features, including: sizes, types, execution contexts, allocation sites, as well as others. Using this framework, we evaluate the effectiveness of each feature for characterizing different memory behavior when: 1) the profiled execution matches closely with guided execution, and 2) the profiled execution uses a different (smaller) input than the guided execution. Additionally, we employ this approach to study the impact of using different data features to guide data tiering on a heterogeneous memory platform with a tier of high-performance memory with limited capacity.

The main contributions of this work are as follows:

1. We design and implement a simulation framework, based in the Intel® Pin framework, to evaluate the accuracy and utility of using different program data features to characterize and guide memory usage behavior.
2. We find that accurate profiles of even very simple features, such as object sizes, can characterize memory usage behavior effectively, but more complex features, such as the set of instructions that access an object, significantly improve the accuracy of memory usage characterization.
3. We show that profile input has a substantial impact on the effectiveness of feature-based guidance, and that many features that work well for summarizing accurate profiles of the same program input are much less effective for predicting behavior across different program inputs.
4. We find that allocation context is the most effective feature for use with software-guided data tiering when the profiled and guided execution use different program inputs. In the system configuration we analyzed, this approach correctly classifies objects corresponding to 93% and 87% of the capacity and bandwidth generated by an application, respectively, compared to the ideal approach with accurate knowledge of the behavior of individual data objects.

## 2 RELATED WORK

### 2.1 Application Guided Data Management

Many prior works have successfully used program profiling and analysis to improve data management across the cache and memory hierarchies. Some researchers have proposed static techniques with offline profiling and/or source code analysis to allocate hot fields and objects closer together, thereby improving caching efficiency [11, 31, 37, 41, 50]. Others have combined online profiling with high-level language features and services, such as object indirection and garbage collection, to enable similar benefits transparently, and in an adaptive runtime environment [10, 14, 15, 24, 27, 28, 30, 55, 63, 64, 68].

A number of other techniques integrate application-level guidance with physical data management in the operating system and hardware. Several previous efforts have employed these cross-layer approaches with automated collection of high-level guidance to address a variety of issues, including: DRAM energy efficiency [36, 47], cache pollution [26], traffic congestion for non-uniform memories [20], and data movement costs for non-uniform caches [43, 59].

In recent years, application guidance has also become a useful tool for adapting existing programs to emerging heterogeneous memory platforms. For example, the memkind API provides an option to bind all data objects larger than a specified size to a particular memory tier [12]. Some other projects allow applications to tag and profile certain program data, and then use classification heuristics to assign data to the appropriate tier [4, 5, 22, 49]. Several works have also proposed static analyses and runtime support to associate tiering guidance with program phases and allocation sites to help further automate this approach [23, 48, 51, 60, 65].

All of these works highlight the power and importance of application guidance for addressing a broad range of data management challenges for modern programs. In contrast to these prior efforts, this work does not attempt to create novel memory management FDOs to increase program performance or efficiency. Rather, it aims to evaluate the effectiveness of different data features for summarizing and characterizing memory usage behavior in order to increase understanding and facilitate further research in this area.

## 2.2 Accuracy and Effectiveness of Profiling

Researchers have also proposed strategies to improve the quality of program profiling. Some works have examined how the rate and pattern of sampling-based profiling affects its accuracy [7, 44]. Several others have developed online profiling tools that employ under-utilized computing cores or other architectural features to reduce the overhead of online profiling [32, 42, 62, 67, 69]. While these works also investigate issues that can limit the effectiveness of profile-based guidance, their focus and goals are different from those of this work; we aim to understand and describe the benefits and drawbacks of associating usage profiles with different data features, even when the usage profiles can be collected accurately.

Other works have investigated the impact of program input on optimization behavior with the goal of improving FDO in ahead-of-time compilers [8, 61]. Another study explored challenges related to summarizing and structuring profile data that can limit its effectiveness for guiding certain VM tasks [35]. This research is particularly relevant because it investigates issues that can also limit the performance of memory management FDOs. However, our work is the first to study how and whether aggregating usage profiles with different data features addresses these issues effectively.

# 3 METHODOLOGY

This study employs dynamic binary instrumentation (DBI) to simulate and profile application memory behavior. For each program run, the DBI outputs memory usage statistics, as well as feature identifiers, for each data object, aggregated over its entire lifetime. This information is used to model the memory behavior of larger sets of program data by combining the usage profiles of specific program objects or of objects that share common features.

In addition to enabling static comparisons of different sets of program data, this approach provides knowledge and control of the usage associated with individual program objects. Since most applications allocate and manage data at the object level, this information can be used to model different data placement decisions. While the static profiles are not able to distinguish usage behavior during different portions of an object's lifetime, many modern runtime systems, especially those used with non-managed languages such as C and C++, do not attempt to migrate program data after its initial placement due to the high costs associated with data movement. Thus, the DBI profiles are an effective means to study the impact of realistic data placement and management policies.

The remainder of this section presents an overview of the design of our DBI-based toolset for simulating and profiling memory usage associated with program data features. Later, it describes the features we selected for this study, provides details about how we detected each feature in simulation, and discusses potential challenges for detecting each feature in non-simulation environments.

## 3.1 Memory Usage Simulation and Profiling

Our memory usage simulation and profiling tool, which we call *memtracer*, is implemented as a Pintool in the Intel® Pin framework. Figure 1 illustrates the main components of `memtracer`. When an application binary image is loaded, `memtracer` inserts callbacks to invoke custom instrumentation at each (a) basic block entry point, (b) memory access instruction (e.g., memory load or store), and (c) object allocation and deallocation routine (e.g., `malloc` or `free`).

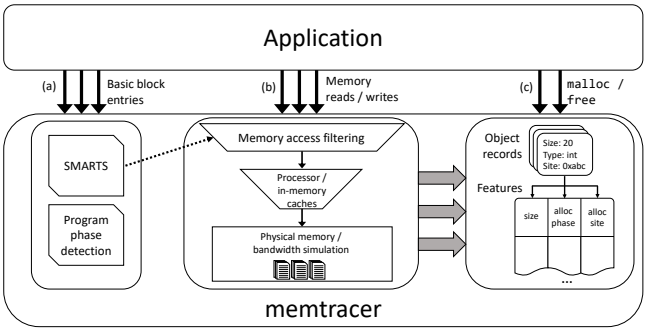


**Figure 1: `memtracer`: dynamic binary instrumentation for memory usage simulation and profiling. `memtracer` includes three instrumentation points: (a) basic block entry to select representative program intervals for accelerated simulation and to track program phase behavior, (b) memory access instructions to simulate cache and memory usage, and (c) object allocation / deallocation routines to associate memory usage with program data objects and features.**

The basic block instrumentation implements the SMARTS strategy for accelerating microarchitectural simulation [66]. SMARTS divides program execution into a set of regular intervals, and selects only a small portion of instructions within each interval for detailed simulation. `memtracer` employs this approach to limit memory usage simulation to a subset of dynamic instructions that are representative of the entire application. Specifically, `memtracer` ignores memory access instructions if they occur outside a simulation interval. Otherwise, it converts each accessed virtual address to its corresponding physical address on the host platform (using the Linux `pagemap` facility [2]), and filters the physical address through a set of built-in cache simulators. Further accounting keeps track of the number and type of accesses and bandwidth to each physical memory page and cache line.

To relate the simulated memory behavior to program data objects and features, `memtracer` maintains a shadow structure that maps the virtual addresses of each heap object to an internal record. Each object record includes several fields to keep track of various usage behaviors, including: counters for data accesses of different types and at different levels of the cache and memory hierarchy, a set of physical addresses corresponding to the cache lines accessed by the object, and dynamic instruction counts denoting the beginning and end of the life of the object. Additionally, the records store identifiers and other information to describe features associated with the object, which are described in the next subsection.

`memtracer` creates a record for each new object at the time it is allocated, and updates its usage and features until the object is deallocated or until the application terminates. When an object reaches the end of its life, `memtracer` finalizes its usage and feature information, removes its record from the shadow structure, and outputs its record to a file on disk. To ensure the object usage profiles are both complete and accurate, `memtracer` instruments *every* object allocation and deallocation event, whether it occurs within a SMARTS simulation interval or not.

| Feature | Code | Online detection? | Distinguishes objects ... |
|---|---|---|---|
| application | app | yes | in the same application |
| size | sz | yes | allocated with exactly the same size |
| size bucket | sb | yes | allocated with similar, but not necessarily identical, sizes |
| type | ty | maybe | with the same data type |
| allocation phase | aph | maybe | allocated during the same program phase |
| phase signature | phs | no | alive during the same set of program phases |
| access signature | acc | no | accessed by the same set of program instructions |
| allocation site | st | yes | allocated from the same program instruction |
| allocation context | st + c$n$ | maybe | allocated from the same call stack context (up to depth $n$) |

**Table 1: Program Data Features. For each feature, the columns show: a short code used to refer to the feature, whether it is feasible to detect the feature during execution, and a short description of the objects that share each feature.**

## 3.2 Program Data Features

This study identifies nine features for summarizing memory usage profiling, which are shown in Table 1, and described below.

*Application:* One option for summarizing memory usage is to not use any features to distinguish program data, but rather employ a single coarse-grain profile for the entire application. This approach is trivial to implement because it does not attempt to distinguish data with different properties.

*Size and Size Bucket:* The size feature distinguishes data objects with the same allocated size (in bytes). In both simulation and non-simulation environments, object sizes can easily be detected in the data allocator, which uses the size to allocate the proper amount of space for the object. In some cases, the exact size of an object is less important than whether the difference between sizes is relatively large or small. The size bucket feature distinguishes objects with sizes that fall within some pre-defined range of values. For this work, the ranges are defined as follows: sizes under 10 bytes are all distinct, sizes 10 through 99 use ranges of size 10 (e.g., 10-19, 20-29, etc.), sizes 100 through 999 use ranges of size 100 (e.g., 100-199, 200-299, etc.), and so on. Similar to the size feature, size buckets are easy to detect in both simulation and non-simulation environments.

*Type:* The type feature distinguishes objects with the same data type. Many modern high-level languages, such as Java and Python, support type introspection for arbitrary data objects. However, some older yet still popular languages, such as C and Fortran, do not have any way to determine object types during execution. Since many of the applications we use for this study are written in such languages, we created an alternative mechanism to detect objects with different data types.

Our approach employs a custom LLVM compiler pass to instrument each cast instruction in the LLVM IR with a callback to `memtracer`. Using this instrumentation, `memtracer` records the set of types to which each object is cast, and uses this set to identify the type of the object. While this approach is useful for distinguishing data used as different types, it has some limitations, specifically: 1) it incurs high overheads not suitable for non-simulation environments, 2) since types are constructed dynamically, the full type of a data object may not be known until after its use, and 3) the LLVM cast instruction does not distinguish pointer types from array types. Languages with native support for type introspection can address all of these limitations, and could potentially improve the effectiveness of this approach.

*Allocation Phase and Phase Signature:* Previous studies have shown that some program behaviors (e.g., instructions per cycle, branch prediction, etc.) are correlated with distinct and repeatable phases of program execution [25, 52, 53]. For this work, we aim to evaluate the potential of using program phases to distinguish application data with similar behavior. Specifically, this study includes two features that use program phases to categorize program data: 1) the allocation phase distinguishes data allocated during the same program phase, and 2) the phase signature distinguishes data alive during the same set of program phases.

Both features rely on an online phase tracking component to categorize recent execution intervals into distinct program phases. Several previous works have proposed techniques to detect program phases with low overhead, but such strategies often require runtime or hardware support that are not commonly available [21, 45, 54]. Given such a capability, allocation phases can distinguish program data prior to its first use, but phase signatures are only useful for offline data classification.

To implement online phase detection, we extended `memtracer` with the architectural phase tracking technique proposed by Sherwood et al. [54]. The adopted approach divides the application's instruction stream into regular length intervals, computes the execution frequency of each basic block within each interval, and stores the normalized frequencies in a *basic block vector (BBV)* for each interval. At the end of each program interval, the online component computes the distance between the current BBV and all previous BBVs that correspond to a unique program phase. If the distance to the closest BBV is larger than some threshold value, the current BBV is categorized as a new and unique program phase, and otherwise, as a re-occurrence of the phase corresponding to the closest BBV. For this study, we configured `memtracer` to use intervals of length $2^{26} \approx 67$M instructions and a unique phase threshold of $2^{19} \approx 524$K for all program phase tracking.

*Access Signature:* The access signature feature distinguishes data objects that are accessed by the same set of program instructions. Some offline profiling and debugging tools use the locations of memory access instructions to track and identify data with similar usage [3, 18]. However, access signatures are not often used in feedback-directed optimizations due to their high detection cost and because they cannot be determined until after the data has been used. To construct access signatures for each data object, `memtracer` inserts the address of each instruction that accesses a data object into a set on the corresponding object record. The set

| Benchmark | Objects | Accessed Lines (MB) | Accesses (millions) | RD/WR Accesses | PC BW (MB) | PC BW on Heap | PC Hit Rate | MC BW (MB) | MC Hit Rate | Dyn. Inst. (trillions) | Sim. Hours |
|---|---|---|---|---|---|---|---|---|---|---|---|
| perlbench | 803,173 | 1,084.4 | 5,997 | 4.03 | 761.8 | 0.954 | 0.998 | 1,247.8 | 0.3693 | 9.565 | 100.5 |
| gcc | 4,776,195 | 11,650.4 | 3,008 | 4.65 | 7,798.8 | 0.905 | 0.958 | 11,096.5 | 0.5811 | 8.441 | 113.3 |
| mcf | 117,316 | 1,051.3 | 4,127 | 3.03 | 20,627.9 | 0.998 | 0.918 | 32,630.4 | 0.4252 | 1.243 | 17.0 |
| omnetpp | 40,226,296 | 8,282.3 | 3,111 | 3.33 | 13,198.6 | 0.960 | 0.939 | 19,136.6 | 0.5356 | 20.481 | 137.1 |
| xalancbmk | 3,826,285 | 2,466.1 | 5,130 | 10.37 | 1,583.5 | 0.793 | 0.996 | 2,010.4 | 0.7247 | 8.030 | 71.5 |
| x264 | 3,791 | 272.3 | 6,625 | 22.54 | 2,734.9 | 0.996 | 0.993 | 4,601.5 | 0.3231 | 2.225 | 16.0 |
| leela | 478,038 | 1952.2 | 866 | 1.89 | 133.8 | 0.683 | 0.997 | 242.2 | 0.2347 | 4.078 | 33.7 |
| xz | 84 | 533.1 | 5,834 | 7.50 | 3,206.6 | 0.995 | 0.991 | 5,167.6 | 0.4028 | 2.001 | 14.8 |
| bwaves | 4,979 | 12,892.2 | 14,241 | 7.40 | 59,346.9 | 0.998 | 0.931 | 102,929.6 | 0.2719 | 5.775 | 93.5 |
| cactuBSSN | 5,038 | 758.9 | 3,234 | 12.57 | 12,284.6 | 0.997 | 0.937 | 19,125.7 | 0.4292 | 2.460 | 40.0 |
| namd | 4,756 | 149.0 | 9,012 | 4.78 | 1,678.4 | 0.999 | 0.997 | 2,725.6 | 0.3744 | 2.348 | 20.7 |
| parest | 12,459,446 | 10,020.4 | 17,134 | 30.06 | 17,257.7 | 0.992 | 0.984 | 26,755.8 | 0.4349 | 11.385 | 93.2 |
| wrf | 830,597 | 18,932.2 | 15,389 | 4.72 | 22,813.0 | 0.944 | 0.975 | 31,121.4 | 0.6542 | 19.930 | 171.8 |
| blender | 97,873 | 455.6 | 2,916 | 110.01 | 2,203.2 | 0.980 | 0.988 | 3,019.2 | 0.6388 | 2.390 | 22.9 |
| cam4 | 605,068 | 12,005.0 | 3,186 | 3.84 | 8,256.4 | 0.881 | 0.957 | 12,534.9 | 0.4577 | 17.861 | 144.9 |
| imagick | 179 | 428.5 | 11,093 | 19.95 | 287.5 | 0.992 | 1.000 | 488.4 | 0.2901 | 5.912 | 33.8 |
| nab | 263,603 | 561.5 | 3,958 | 5.26 | 966.0 | 0.994 | 0.996 | 1,721.7 | 0.2186 | 2.249 | 20.7 |
| fotonik3d | 1,213 | 845.0 | 16,554 | 5.74 | 59,427.4 | 1.000 | 0.941 | 109,729.7 | 0.1536 | 8.124 | 84.5 |
| roms | 1,557,582 | 26,984.1 | 13,225 | 4.15 | 38,656.1 | 0.976 | 0.954 | 58,741.6 | 0.4912 | 6.877 | 81.7 |
| mean | 3,476,922 | 5,859.2 | 7,613 | 13.99 | 14,380 | 0.949 | 0.971 | 23,422.5 | 0.4216 | 7.441 | 69.0 |
| median | 263,603 | 1,084.4 | 5,834 | 5.26 | 7,798.8 | 0.992 | 0.984 | 11,096.5 | 0.4252 | 5.912 | 71.5 |

**Table 2: Benchmark statistics. From left to right, the columns show: benchmark name, # of objects accessed in memory, cache lines accessed for all data objects (in MB), total # of memory accesses from the application in millions, read-write ratio of application memory accesses, memory bandwidth with processor cache alone (in MB), portion of (processor cache-only) bandwidth to heap data, processor cache hit rate, memory bandwidth with processor and in-memory caches, in-memory cache hit rate, # of dynamic instructions in trillions, and simulation time in hours.**

of addresses recorded by the end of the life of an object identifies its access signature.

*Allocation Sites and Context:* The allocation site feature distinguishes data that is allocated by the same instruction in the source code. Similarly, the allocation context feature distinguishes data allocated by the same instruction as well as the same function call context up to some length *n*. Allocation sites are particularly useful for memory FDOs because they are easy to detect during execution and immediately prior to the use of each data object. Detecting the allocation context of program data incurs additional overheads, but some compiler [48, 58, 70] or architectural support [39] can help reduce and mostly eliminate these overheads. For this study, `memtracer` employs the `backtrace` routine from the C standard library to detect program context at each allocation instruction.

*Combined Features:* In some cases, it may be useful to combine the basic features in Table 1 to distinguish data that meet multiple criteria. For example, while our mechanism for detecting data types cannot distinguish pointer and array types, we can combine the type and size features to distinguish arrays of the same size and base data type. Alternatively, combining the allocation phase feature with sites or sizes can help distinguish data with similar static features based on when it is allocated. Such combinations could potentially enable more effective data characterization, but may require additional profiling or detection costs.

# 4 EXPERIMENTAL SETUP

*Platform.* All of our simulation experiments were run on a Microway NumberSmasher Xeon server machine with two Intel E5-2620v3 (Haswell-EP) processors and 128GB of DDR4 SDRAM. We installed 64-bit CentOS 6.9 and Linux kernel version 2.6.32-431 as the operating system. We use Intel® Pin v. 3.5 as the base for the `memtracer` tool.

*Memory Simulation Details.* To accelerate memory usage simulations, `memtracer` implements the SMARTS strategy with sampling interval $k = 64$ and a sampling unit size $U = 2^{13}$ instructions. Hence, each experimental run simulates detailed memory usage for $1/64 \approx 1.6\%$ of all program instructions. Additionally, `memtracer` uses a detailed warming size of $W = 2^{13}$ instructions to warm-up the simulated processor and in-memory caches immediately prior to each simulation interval. The simulation does not include any hardware-based prefetcher components.

The processor cache component in `memtracer` simulates a two-level write-back cache with a 32KB 8-way L1 and an 8MB 16-way L2. The usage simulator estimates memory bandwidth both with and without an in-memory caching component. This component models the hardware-managed caching option found on some modern heterogeneous memory systems, such as the "memory mode" option in recent Intel® platforms [34]. To generate and control contention on the in-memory cache, we set its size as ⅛ of the peak resident set size (RSS) of the application at the start of each experimental run. Both the processor and in-memory caches use 64-byte lines, and the direct-mapped in-memory cache employs a simple modulo function to hash each line to its corresponding block.

## 4.1 Benchmarks Description

Our evaluation uses the single-threaded rate (5xx) version of all integer and floating point applications included in the SPEC CPU 2017 benchmark suite [57], with the following exceptions: 1) We omit *povray* and *exchange* because most (> 50%) of the bandwidth they generate is on the stack, and our analysis only considers features associated with heap data objects. 2) We omit *deepsjeng* and

| benchmark | sz | sb | ty | aph | phs | acc | st | st + c1 | st + c2 | st + c4 | st + c8 | ty + sz | aph + sz | aph + st | aph + sz + st |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| perlbench | 12,591 | 57 | 273 | 52 | 107 | 77,912 | 36 | 160 | 459 | 1,324 | 2,550 | 18,142 | 30,524 | 398 | 33,796 |
| gcc | 47,856 | 65 | 2,258 | 486 | 531 | 675,850 | 8 | 599 | 2,143 | 8,910 | 28,024 | 91,693 | 100,356 | 1,270 | 134,771 |
| mcf | 42 | 10 | 13 | 37 | 39 | 58 | 19 | 19 | 19 | 19 | 19 | 47 | 84 | 86 | 112 |
| omnetpp | 292 | 42 | 263 | 4 | 5 | 94,426 | 360 | 524 | 611 | 873 | 1,141 | 593 | 439 | 470 | 1,070 |
| xalancbmk | 8,049 | 48 | 274 | 82 | 84 | 7,985 | 4 | 350 | 583 | 772 | 1,035 | 8,694 | 119,705 | 99 | 119,727 |
| x264 | 44 | 27 | 4 | 2 | 3 | 437 | 8 | 73 | 74 | 91 | 91 | 45 | 46 | 10 | 49 |
| leela | 971 | 46 | 20 | 12 | 17 | 3,649 | 34 | 57 | 87 | 159 | 217 | 1,007 | 4,307 | 134 | 5,107 |
| xz | 25 | 18 | 14 | 8 | 14 | 60 | 18 | 21 | 21 | 24 | 28 | 26 | 57 | 52 | 57 |
| bwaves | 461 | 14 | 1 | 12 | 15 | 213 | 39 | 39 | 39 | 39 | 39 | 461 | 1,145 | 140 | 3,524 |
| cactuBSSN | 131 | 33 | 93 | 28 | 33 | 936 | 286 | 870 | 1,770 | 2,041 | 2,204 | 256 | 339 | 433 | 773 |
| namd | 252 | 22 | 18 | 3 | 6 | 2,510 | 49 | 69 | 69 | 69 | 69 | 257 | 261 | 53 | 434 |
| parest | 970 | 54 | 291 | 95 | 97 | 67,617 | 396 | 780 | 1,131 | 1,861 | 2,701 | 1,597 | 3,530 | 1,908 | 5,000 |
| wrf | 985 | 41 | 10 | 82 | 86 | 14,309 | 1,348 | 1,389 | 1,452 | 1,551 | 1,819 | 1,008 | 4,298 | 3,564 | 20,566 |
| blender | 1,771 | 56 | 16 | 62 | 69 | 10,324 | 29 | 323 | 959 | 2,241 | 2,425 | 1,791 | 2,830 | 196 | 2,981 |
| cam4 | 1,742 | 46 | 19 | 146 | 152 | 16,625 | 688 | 1,770 | 2,099 | 2,486 | 2,780 | 1,779 | 5,828 | 2,803 | 9,200 |
| imagick | 44 | 31 | 21 | 16 | 18 | 109 | 25 | 49 | 77 | 124 | 135 | 51 | 92 | 67 | 98 |
| nab | 2,326 | 40 | 15 | 9 | 18 | 268 | 77 | 104 | 117 | 145 | 145 | 4,370 | 7,399 | 91 | 7,464 |
| fotonik3d | 91 | 21 | 2 | 6 | 6 | 152 | 117 | 117 | 117 | 117 | 117 | 92 | 92 | 117 | 124 |
| roms | 379 | 21 | 2 | 57 | 61 | 3,529 | 308 | 317 | 352 | 531 | 540 | 380 | 3,105 | 883 | 14,690 |
| mean | 4,159 | 36 | 190 | 63 | 72 | 51,419 | 203 | 402 | 641 | 1,230 | 2,425 | 6,963 | 14,970 | 672 | 18,923 |
| median | 461 | 40 | 18 | 28 | 33 | 3,529 | 39 | 160 | 352 | 531 | 540 | 593 | 2,830 | 140 | 3,524 |

**Table 3: Number of unique sets of data objects corresponding to each data feature.**

*lbm* because they each allocate only one or two large heap objects throughout the entire run. Since our evaluation does not attempt to distinguish data below the object level, applications with only a few data objects will not exhibit much distinction between features.

All benchmarks were compiled with the LLVM compiler toolchain v. 4.0.1 with -O3 [40]. We ran each benchmark simulation using both the *train* and *ref* input sets, and report memory usage results for the *ref* input. In cases where the benchmark-input pair requires multiple invocations, we conducted independent runs of each invocation and aggregated the output in post-processing to produce a single set of results for each application.

Table 2 presents our selected benchmarks along with relevant simulation and usage statistics. Thus, the selected benchmarks exhibit a wide range object allocation, capacity, bandwidth, and caching behaviors. Note that, while `memtracer` captures all object allocations, Table 2 and our later experiments only include objects that that are accessed in memory at least once during a detailed simulation interval. Additionally, the "Accessed Lines" column of Table 2 shows the sum of all cache lines accessed in MB for all data objects. The experiments in Section 5.2 use this metric to estimate capacity requirements for each data object.

Let us make a few additional observations: 1) All applications generate fewer writes than reads, but some applications are much less write-intensive than others, 2) The in-memory cache configuration typically generates much more bandwidth than the processor cache alone due to the high miss rate of the direct-mapped caching scheme, and 3) The execution time of each simulation varied from less than 15 hours to over a week of compute time. We found that increasing the simulation sampling interval significantly reduces execution times, but could potentially be less accurate.

# 5 EVALUATION

This section presents multiple studies on the effectiveness of different program data features for characterizing memory behavior and guiding memory management. The first subsection evaluates the accuracy of using different data features to aggregate and summarize memory usage profiling. The following subsection describes a case study that shows how the accuracy of different features impacts the performance of software-guided data tiering for heterogeneous memory systems.

**Algorithm 1** Calculating mean usage error for each data feature.

```
1: procedure FEATURE_MEAN_USAGE_ERROR(ft, stat, objects)
2:     ft_err ← 0
3:     for each object obj ∈ objects do
4:         obj_usage ← get_object_usage (obj, stat)
5:         obj_ft_set ← get_feature_set (obj, ft)
6:         ft_usage ← get_feature_mean (ft, obj_ft_set, stat)
7:         obj_err ← min ( obj_usage, |(ft_usage − obj_usage)| )
8:         ft_err ← ft_err + obj_err
9:     return ft_err
```

## 5.1 Accuracy of Different Data Features for Characterizing Memory Usage Behavior

Our first set of experiments examines the accuracy by which profiles of data with a particular feature may predict the usage of unknown data with the same features. To conduct this study, we used `memtracer` to collect memory usage statistics and detect the features of each program data object as described in Section 3. We then computed the error associated with predicting the memory usage of each program object using the memory behavior of the other data objects with the same features. Specifically, we use the FEATURE_MEAN_USAGE_ERROR procedure in Algorithm 1 to calculate this error for each application.

The FEATURE_MEAN_USAGE_ERROR characterizes how similar the data objects categorized into the same set by some data feature (e.g., size, type, allocation site, etc.) are with respect to some memory usage metric (e.g., cache lines accessed, bandwidth, etc.). Lower error values indicate data in the same feature set exhibit more similar behavior. Specifically, given a data feature type $ft$, and

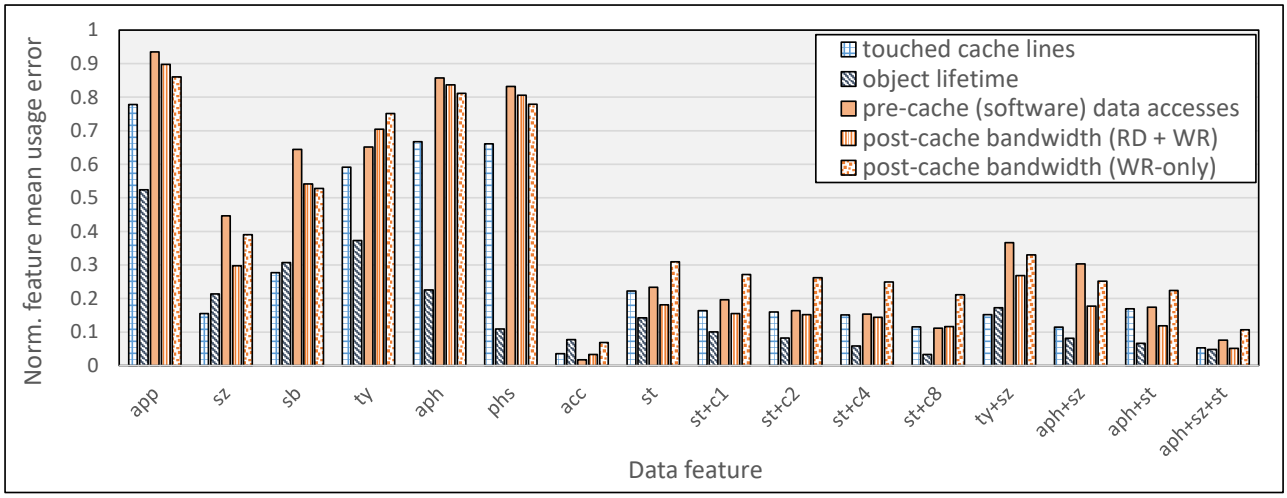


**Figure 2: Normalized feature mean usage error for characterizing number of accessed cache lines, object lifetime, pre-cache (software) data accesses, post-cache RD+WR and WR-only bandwidth (no in-memory cache) for each feature (lower is better).**

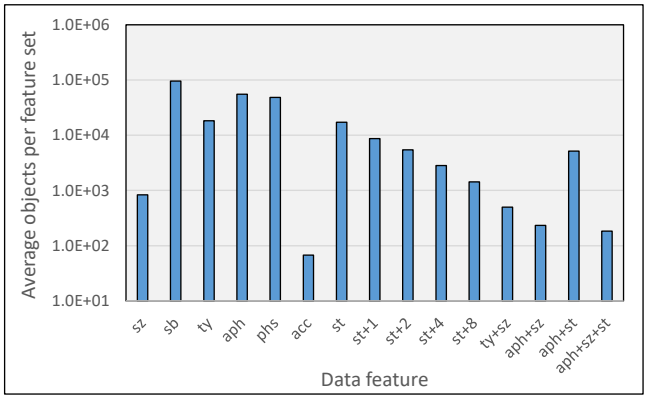


**Figure 3: Average number of objects in each feature set. Higher values indicate each set in the feature summarizes the behavior of more application data, on average.**

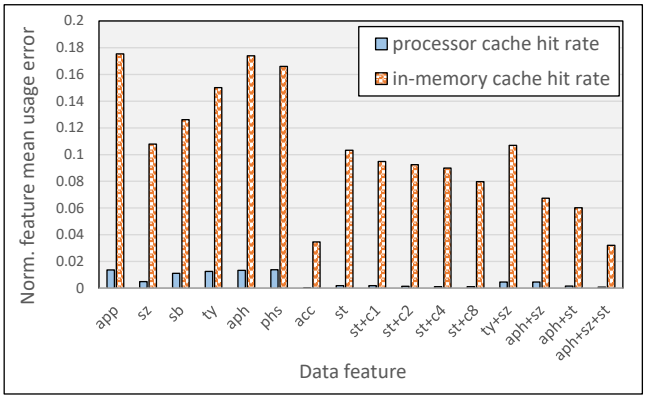


**Figure 4: Normalized feature mean usage error for characterizing processor and in-memory cache hit rate with each program data feature (lower is better).**

a memory usage statistic *stat*, the FEATURE_MEAN_USAGE_ERROR procedure iterates over each program object and computes the difference between the object's usage and the mean usage of other data in the same category for that feature. For example, consider that a given object *o* was allocated with size 24, and generates 10 MB of bandwidth over the course of the run. If the other application objects of size 24 (not including *o*) generate a mean average bandwidth of only 8 MB over the entire run, then *o* contributes 2 MB of bandwidth error for the size feature.

*5.1.1 Objects Per Feature Set.* To provide additional context for this experiment, let us first consider how each feature divides application data objects into different sets. Table 3 displays the number of unique sets of data objects corresponding to each feature for each benchmark application. For example, the table shows that *mcf* generates objects corresponding to 42 distinct sizes, but only 13 distinct types. Note the last four columns of Table 3 show the results for different combinations of the basic feature categories described in Table 1. Figure 3 also helps summarize these results by showing the average number of objects classified in each set for each feature across all of the benchmarks. Additionally, the total number of objects allocated by each application is in Table 2.

Overall, we find that size buckets, allocation phases, and phase signatures have the fewest number of unique sets of objects, and therefore assign the most objects per set, on average. While there is some variation within each benchmark, types and allocation sites (without context) assign the next most objects to each feature set. Not surprisingly, adding context to the allocation site feature, or combining feature criteria, such as types and sizes or allocation sites and phases, further decreases the number of objects corresponding to each unique feature set. Interestingly, access signatures tend to be more unique than the other features, which limits their effectiveness as a tool for summarizing memory usage behavior.

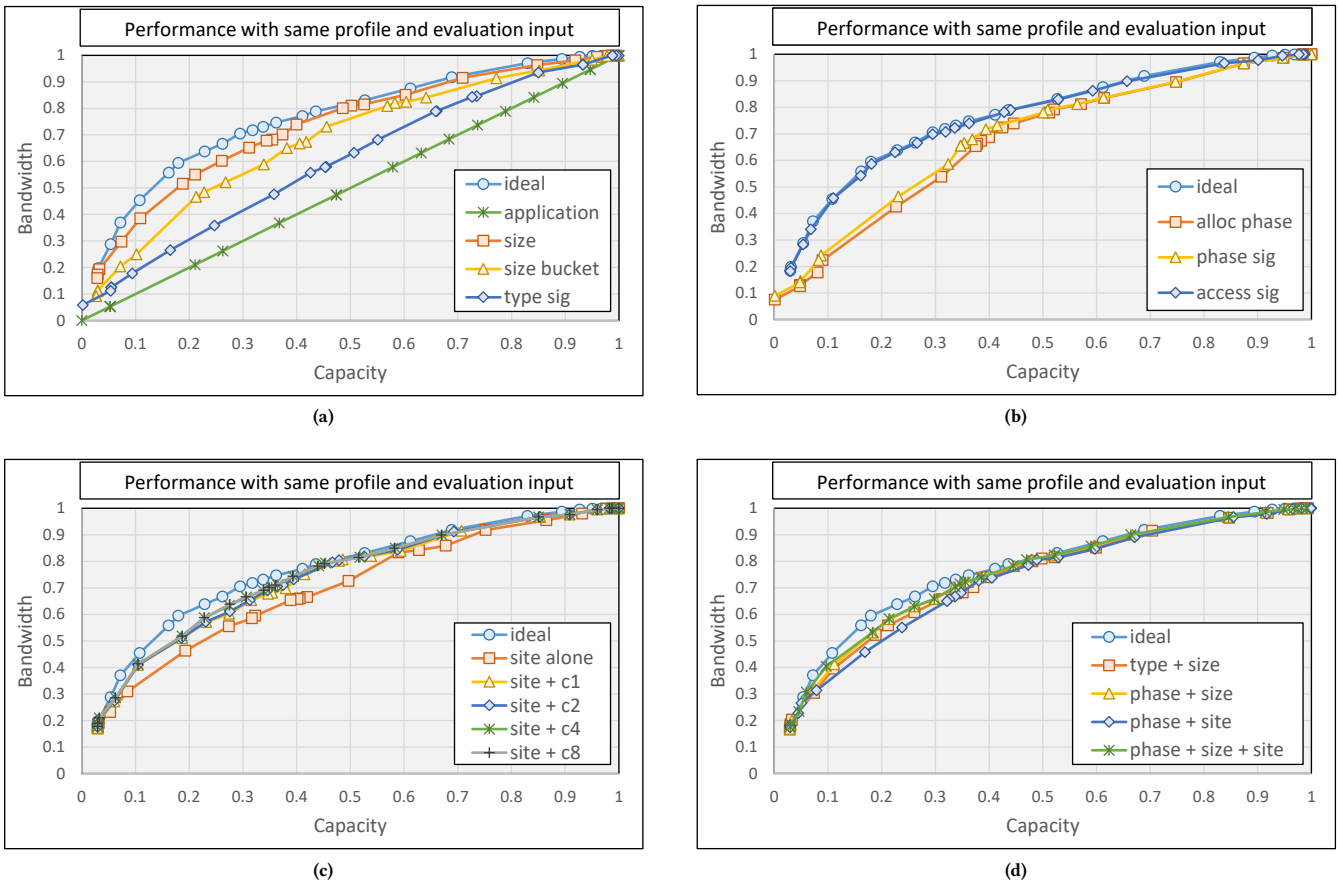


**Figure 5: Portion of bandwidth and capacity assigned to high-performance memory using profiling of different data features to partition application data. For these results, the profiled and guided execution use the same program input.**

*5.1.2 Feature Mean Usage Errors.* Figure 2 presents the feature mean usage error for each data feature, averaged over all benchmarks, for the following simple usage statistics: accessed cache lines, object lifetime, pre-cache (software) data accesses, post-cache RD/WR bandwidth (no in-memory cache), and post-cache WR-only bandwidth. The results are normalized by the maximum possible usage error for each usage statistic (i.e., the sum of all object values for a given statistic). Additionally, Figure 4 shows the feature mean usage errors for the processor and in-memory cache hit rates, again averaged over all benchmarks. For this figure, the usage error is calculated as the total number of cache hits and misses misclassified by the feature mean cache hit rate for all program objects, and is normalized by the total number of data accesses.

The results reveal several interesting findings: 1) Even relatively simple features, such as size and size bucket, characterize usage more effectively than a single profile of the entire application. 2) Features with fewer objects per feature set, such as access signatures and combined features, tend to have smaller classification errors. However, allocation sites significantly outperform other features with similar or fewer objects per set, especially when characterizing memory accesses or bandwidth. 3) It is more challenging to characterize in-memory cache utilization than processor cache utilization due to the relatively high miss rate of the in-memory cache. 4) Adding context to each allocation site does improve classification accuracy for each type of memory usage behavior for some object oriented applications, such as *gcc*, *omnetpp*, and *leela*, which rely heavily on custom allocation pools or external libraries for data allocation. However, additional allocation context has only a minor impact on average because most of the applications in CPU® 2017 are procedural codes with relatively simple allocation patterns. 5) In general, the relative accuracy of each feature is similar for all of the memory usage behaviors, but certain features are more effective for characterizing certain types of usage. For instance, object size exhibits relatively small errors for accessed cache lines, while allocation phases and phase signatures are relatively more effective for characterizing object lifetimes.

## 5.2 Using Data Features to Guide Heterogeneous Memory Management

Our next study aims to evaluate the impact that aggregating memory usage profiles with different data features has on a realistic feedback-directed optimization for heterogeneous memory systems.

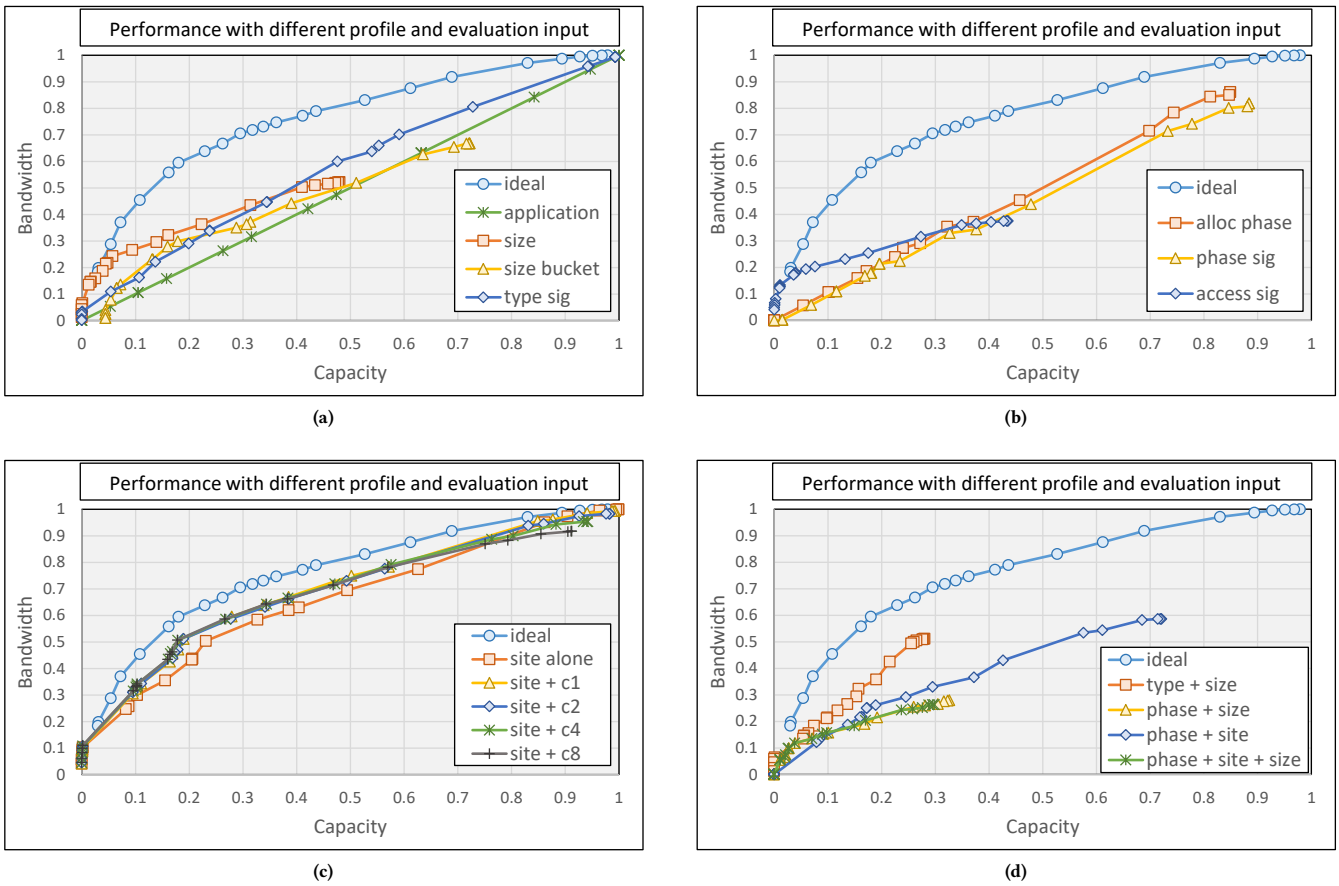


**Figure 6: Portion of bandwidth and capacity assigned to high-performance memory using profiling of each data feature to partition application data. For these results, the profiled and guided execution use different program inputs.**

Many current and emerging memory systems employ a multi-tier architecture where the upper tier exhibits better performance, but has lower capacity than the other tiers. For such architectures, a common memory management task is to steer as much bandwidth to the upper tier as possible within its capacity constraint.

We designed a set of experiments to study the effectiveness of using profiling of different data features to guide this task. The experiments employ the simulation data to estimate the bandwidth and capacity of each individual data object. Specifically, we use the post-cache bandwidth of the processor cache alone (i.e., without the in-memory cache) for object bandwidth, and the number of cache lines accessed multiplied by the cache line size of 64 bytes for object capacity. In contrast to counting the entire allocation size of the object, this approach avoids over-estimating capacity for unused data objects.

For each application and each feature type, we divided the objects into sets corresponding to their individual features. We then computed a ratio of the sum of the bandwidths and capacities of the objects in each set to create a single "bandwidth-per-byte" score for each feature set. Next, we sorted the feature sets by their bandwidth-per-byte scores, and used a threshold value to divide the feature sets into hot (i.e., high-bandwidth, low capacity) and cold groups. The resulting partitions then represent guidance that can be used to divide the application's data into separate memory tiers with different performance and capacity.

*5.2.1 Effectiveness of Guided Memory Management using the Same Program Input for Both the Profiling and Evaluation Runs.* For the first version of this experiment, we use simulations of the *ref* program input to both generate the feature-based guidance and evaluate the effects of that guidance. Additionally, rather than use a single threshold to partition each feature set, we apply a large set of thresholds to produce a range of aggregate bandwidth and capacity effects. This approach allows us to quickly compare the effectiveness of feature-based guidance across system configurations with a range of different capacities available in the upper tier.

The scatter plots in Figure 5 show the aggregate capacity (on the x-axis) and bandwidth (on the y-axis) of the objects assigned to the high performance tier by the guidance at each threshold for each feature type. All results are normalized by the total bandwidth and capacity in the application, and averaged across all 19 benchmarks. For presentation purposes, we plot different lines for each feature across four separate graphs with a small number of lines per graph.

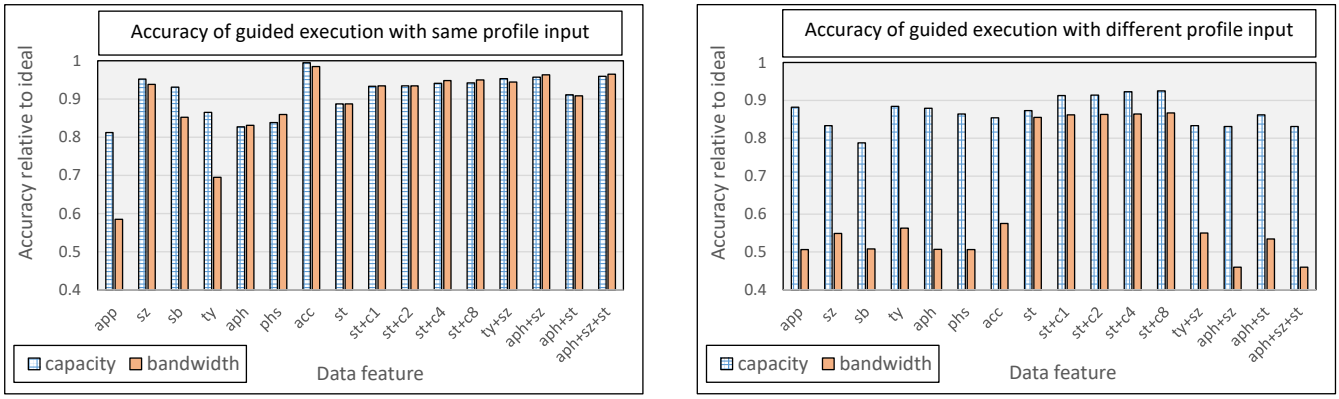


**Figure 7: Portion of capacity and bandwidth assigned to the same set as the ideal configuration with profiles of different features that use (a) the same and (b) different inputs for the profile and evaluation runs.**

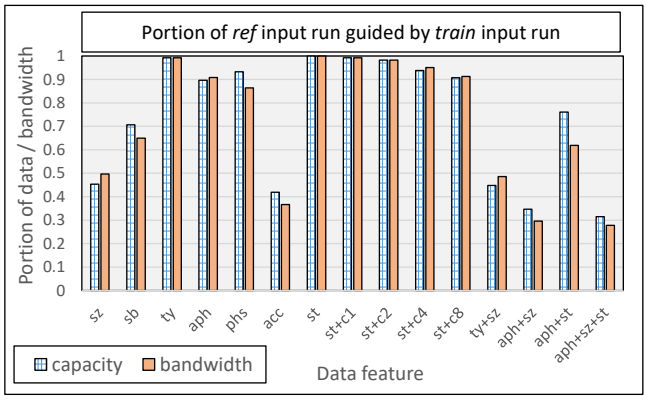


**Figure 8: Portion of data and bandwidth in the *ref* input run corresponding to profile guidance in the *train* input for each data feature.**

To anchor the results and enable comparison of features across figures, each graph includes results for an "ideal" configuration that uses accurate profiles of individual data objects to partition the applications' data. For example, the ideal results show that program data objects corresponding to only 10% of the applications' capacity generate about 45% of their bandwidth. However, guidance associated with the size bucket feature can only distinguish data with about 25% of the bandwidth given a capacity limit of 10%.

Thus, even if the profile run closely matches the evaluation run, using certain data features to aggregate memory usage profiling can still limit the effectiveness of this approach.In particular, profiles associated with *size buckets*, *type signatures*, and *phase-based* features are often too coarse-grained to achieve the full benefits of profile-guided data placement. In contrast, several of the combined features as well as allocation sites, especially with some amount of context, are much more effective at summarizing memory usage for this task. Access signatures achieve performance closest to the ideal scenario, but are often not feasible to deploy in a real FDO.

*5.2.2 Effectiveness of Guided Memory Management using Different Inputs for the Profiling and Evaluation Runs.* For many applications, it is not always practical to collect profiles of execution that use the same program input as is used during guided execution. To evaluate the effectiveness of each data feature for summarizing and applying profiles of different program inputs, we conducted a second experiment that uses profiles of each application with the *train* program input to guide a run with the *ref* program input.

To implement this experiment, we must relate the object usage information from the *ref* input run to profiles of each feature from the *train* input run. For this purpose, we developed an automated script to find the feature profiles from the *train* run that match the data objects in the *ref* run. For example, for the size feature, objects of size 10 in the *ref* run correspond to the profile of objects of size 10 in the *train* run, if it exists. Since the phase-based features track program phases online, we extended `memtracer` to output each basic block vector, and use the original BBVs to match each phase in the *ref* run to its closest phase in the *train* run.

In some cases, there is data in the *ref* run that cannot be mapped to a profile in the *train* run. For instance, if the *ref* run uses data objects of size 256, but the *train* run does not allocate any objects of size 256. Figure 8 shows the portion of capacity and bandwidth corresponding to data in the *ref* run that corresponds to some feature-based guidance in the *train* run. While some features, such as types and allocation sites, generate profiles that are able to guide most of the data in the *ref* run, several other features, including sizes and access signatures, cannot provide guidance for a significant portion of program data. For these cases, our experiment simply assigns data in the *ref* run without corresponding guidance from the *train* run to the cold set (i.e., to the slower memory devices).

Figure 6 shows the aggregate capacity and bandwidth of program data in the *ref* run that is assigned to the high performance tier by guidance from the *train* run for each feature category. Similar to the previous experiments, all of the results are normalized and averaged across all of the benchmarks. We find that some of the features that work very well when the profile run matches guided execution, such as sizes and access signatures, are not nearly as

effective when profiling is conducted using a different program input. In several cases, capacity in the high performance tier is left under-utilized due to the inability to map data in the *ref* run to guidance from the *train* run. Interestingly, Figure 6(c) shows that allocation sites, including with context, are still quite effective at assigning hot data into the high performance tier. Thus, while the performance of some data features is highly dependent on profile input, memory FDOs that use allocation sites and context can still be effective even if the profile run does not match guided execution.

*5.2.3 Accuracy of Feature-Based Guidance.* Lastly, we used the results of the ideal configuration to compute the accuracy by which each feature is able to assign data to the correct memory tier. For this calculation, we selected a bandwidth-per-byte threshold that assigns about 18% of the applications' capacity (and about 60% of the bandwidth) to the high performance memory tier with the ideal configuration. This amount of capacity is similar to the actual portion of high performance memory in many modern heterogeneous memory platforms [34, 56]. We then computed the accuracy by which each feature is able to assign data objects into the same set as the ideal configuration at this threshold.

Figure 7 shows the portion of capacity and bandwidth classified into the same set as the ideal configuration using guidance associated with each data feature, averaged (geometric mean) over all benchmark programs. Figure 7(a) shows the results when the feature profiles are collected from and used to guide the same (*ref*) program input, while 7(b) presents the accuracy when using profiling from the *train* runs to guide execution of a run with the *ref* input. For each feature, we show only the results for the bandwidth-per-byte threshold that generated the most accurate hot and cold sets compared to the ideal approach at the fixed threshold.

Thus, the results mostly confirm our earlier observations regarding the accuracy of associating program profiling with each data feature. In Figure 7(b), we see that the *train* profile-guided approach is relatively accurate at assigning capacity to the correct memory tier, regardless of the feature that is used. Since the ideal approach assigns most (82%) of program capacity to the slower memory tier, features that under-utilize the high performance tier will still classify most application capacity correctly. Indeed, only the allocation site features, including with context, are able to assign the vast majority of both capacity and bandwidth to the correct memory tier with the *train*-based profiles. Overall, allocation sites with 8 layers of additional context are the most effective feature for *train*-guided execution, and are able to assign 93% of capacity and 87% of bandwidth to the correct memory tier.

## 6 FUTURE WORK

There are several avenues for future work. First, we plan to extend this study to evaluate the performance of using different data features to guide other memory management tasks, including: minimizing writes to memory devices with low endurance, identifying data with poor locality to increase cache efficiency, and pre-tenuring objects with long expected lifespans to reduce GC effort. This work also found that access signatures are very effective features for characterizing memory behavior, but are difficult to apply in real FDOs. Our future work will explore and develop static analyses and other mechanisms to associate program data with source code constructs and instructions that use it, but that are easier to detect during execution and more stable across program inputs. Additionally, our simulation tools allow us to correlate detailed memory usage statistics with a wide range of data features. In the future, we will employ machine learning techniques to determine if certain features or combinations of features can predict various usage behaviors regardless of the application itself or its input. Finally, we plan to adapt the knowledge and tools we have developed in this study for use with native applications and system software on real and complex memory hardware.

## 7 CONCLUSIONS

This work employs detailed simulations of full scale applications to evaluate the effectiveness of using different program data features to aggregate and characterize memory usage for guiding memory management. It finds that simple data features, such as allocation sizes, are useful for characterizing the behavior of some program data, but more fine-grained features, such as the set of instructions that access an object, and combinations of features, significantly increase the accuracy of this approach. Additionally, it quantifies the impact of using different data features to guide separation of hot and cold data for a realistic memory management FDO on a heterogeneous memory platform with a limited amount of high performance memory. Overall, it finds that combining allocation sites with call stack context is the most effective means to summarize and incorporate guidance for this task, especially when the profiled and guided execution use different program inputs.

## ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their thoughtful comments and feedback. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, with additional contributions from the National Science Foundation under CCF-1619140 and CCF-1617954, as well as the Software and Services Group (SSG) at Intel® Corporation.

## REFERENCES

[1] [n.d.]. mbind - set memory policy for a memory range, Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/mbind.2.html
[2] [n.d.]. pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[3] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
[4] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. *SIGPLAN Not.* 50, 4 (March 2015), 607–618. https://doi.org/10.1145/2775054.2694381
[5] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392
[6] AMD. 2016. High Bandwidth Memory (HBM) Reinventing Memory Technology. https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.
[7] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. 1997. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions Computer Systems* 15, 4 (Nov. 1997), 357–390. https://doi.org/10.1145/265924.265925

[8] Paul Berube. 2012. University of Alberta Methodologies for Many-input Feedback-directed Optimization.

[9] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[10] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. 2013. Pacman: Program-assisted Cache Management. *SIGPLAN Not.* 48, 11 (June 2013), 39–50. https://doi.org/10.1145/2555670.2466482

[11] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious Data Placement. *SIGPLAN Not.* 33, 11 (Oct. 1998), 139–149. https://doi.org/10.1145/291006.291036

[12] Christopher Cantalupo, Vishwanath Venkatesan, and Jeff R Hammond. 2015. User extensible heap manager for heterogeneous memory platforms and mixed memory policies. (2015).

[13] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational stack collection and profile-driven pretenuring. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM, New York, NY, USA, 162–173. https://doi.org/10.1145/277650.277718

[14] Sigmund Cherem and Radu Rugina. 2004. Region analysis and transformation for Java programs. In *Proceedings of the 4th international symposium on Memory management (ISMM '04)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1029873.1029884

[15] Trishul M. Chilimbi and Ran Shaham. 2006. Cache-conscious Coallocation of Hot Data Streams. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 252–262. https://doi.org/10.1145/1133981.1134011

[16] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[17] Hybrid Memory Cube Consortium. 2014. HMC Specification 2.1.

[18] Intel Corporation. 2009. Technologies for Measuring Software Performance: VTune Analyzers. accessed from http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-white-papers/.

[19] Ian Cutress and Billy Tallis. 2016. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here.

[20] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 381–394.

[21] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and Predicting Program Behavior and its Variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 220–230.

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

[23] T. Chad Effler, Adam P. Howard, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, and Prasad A. Kulkarni. 2018. On Automated Feedback-Driven Data Placement in Hybrid Memories.. In *LNCS International Conference on Architecture of Computing Systems (ARCSâĂŹ18)*.

[24] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 251–265. https://doi.org/10.1145/2908080.2908090

[25] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. 2004. Method-level phase behavior in java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 270–287. https://doi.org/10.1145/1028976.1028999

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

[27] Samuel Z. Guyer and Kathryn S. McKinley. 2004. Finding Your Cronies: Static Analysis for Dynamic Object Colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. 237–250.

[28] Martin Hirzel. 2007. Data Layouts for Object-oriented Programs. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. 265–276.

[29] Wei huang, W. Srisa-an, and J. M. Chang. 2004. Dynamic pretenuring schemes for generational garbage collection. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, Washington, DC, USA, 133–140.

[30] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/1028976.1028983

[31] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. 2006. Practical Structure Layout Optimization and Advice. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, Washington, DC, USA, 233–244. https://doi.org/10.1109/CGO.2006.29

[32] Hiroshi Inoue and Toshio Nakatani. 2009. How a Java VM Can Get More from a Hardware Performance Monitor. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 137–154. https://doi.org/10.1145/1640089.1640100

[33] Intel. 2016. 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[35] Michael R. Jantz, Forrest J. Robinson, and Prasad A. Kulkarni. 2016. Impact of Intrinsic Profiling Limitations on Effectiveness of Adaptive Optimizations. *ACM Trans. Archit. Code Optim.* 13, 4, Article 44 (Dec. 2016), 26 pages. https://doi.org/10.1145/3008661

[36] Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer Memory Management for Managed Language Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 488–504. https://doi.org/10.1145/2814270.2814322

[37] Jinseong Jeon, Keoncheol Shin, and Hwansoo Han. 2007. Layout Transformations for Heap Objects Using Static Access Patterns. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*. Springer-Verlag, Berlin, Heidelberg, 187–201. http://dl.acm.org/citation.cfm?id=1759937.1759954

[38] A. Kleen. 2004. A NUMA API for Linux. *SUSE Labs white paper* (August 2004).

[39] Andi Kleen. 2016. An introduction to last branch records. https://lwn.net/Articles/680985/

[40] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[41] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/1065010.1065027

[42] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 198–208. https://doi.org/10.1109/CGO.2007.35

[43] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving Dynamic Cache Management with Static Data Classification. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 113–127. https://doi.org/10.1145/2872362.2872363

[44] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 187–197. https://doi.org/10.1145/1806596.1806618

[45] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. 2006. Online Phase Detection Algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 111–123. https://doi.org/10.1109/CGO.2006.26

[46] NVIDIA. 2016. GP100 Pascal Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

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

[48] M Ben Olson, Tong Zhou, Michael R Jantz, Kshitij A Doshi, M Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–10.

[49] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 82–91. https://doi.org/10.1145/3092255.3092273

[50] Matthew L. Seidl and Benjamin G. Zorn. 1998. Segregating Heap Objects by Reference Behavior and Lifetime. *SIGPLAN Not.* 33, 11 (Oct. 1998), 12–23. https://doi.org/10.1145/291006.291012

[51] H. Servat, A. J. PeÃśa, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[52] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/1024393.1024414

[53] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[54] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. *SIGARCH Comput. Archit. News* 31, 2 (2003), 336–349. https://doi.org/10.1145/871656.859657

[55] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. 2002. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/582419.582422

[56] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–24.

[57] SPEC. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[58] William N Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2011. Precise calling context encoding. *IEEE Transactions on Software Engineering* 38, 5 (2011), 1160–1177.

[59] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 652–665. https://doi.org/10.1145/3079856.3080214

[60] Gwendolyn Voskuilen, Arun F. Rodrigues, and Simon D. Hammond. 2016. Analyzing Allocation Behavior for Multi-level Memory. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 204–207. https://doi.org/10.1145/2989081.2989116

[61] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

[62] Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 209–220. https://doi.org/10.1109/CGO.2007.37

[63] Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On Improving Heap Memory Layout by Dynamic Pool Allocation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 92–100. https://doi.org/10.1145/1772954.1772969

[64] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. 2012. On-the-fly Structure Splitting for Heap Objects. *ACM Trans. Archit. Code Optim.* 8, 4, Article 26 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086705

[65] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[66] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH computer architecture news*, Vol. 31. ACM, 84–97.

[67] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 170–184. https://doi.org/10.1145/2749469.2750401

[68] Chengliang Zhang and Martin Hirzel. 2008. Online Phase-Adaptive Data Layout Selection. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. 309–334.

[69] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. 2008. Pipa: pipelined profiling and analysis on multi-core systems. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1356058.1356083

[70] Tong Zhou, Michael R. Jantz, Prasad A. Kulkarni, Kshitij A. Doshi, and Vivek Sarkar. 2019. Valence: Variable Length Calling Context Encoding. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/3302516.3307351