Understanding Object-level Memory Access Patterns Across the Spectrum

Xu Ji
Tsinghua University
Qatar Computing Research Institute, HBKU

Chao Wang
Oak Ridge National Laboratory

Nosayba El-Sayed
CSAIL, MIT

Xiaosong Ma
Qatar Computing Research Institute, HBKU

Youngjae Kim
Sogang University

Sudharshan S. Vazhkudai
Oak Ridge National Laboratory

Wei Xue
Tsinghua University

Daniel Sanchez
CSAIL, MIT

ABSTRACT
Memory accesses limit the performance and scalability of countless applications. Many design and optimization efforts will benefit from an in-depth understanding of memory access behavior, which is not offered by extant access tracing and profiling methods.

In this paper, we adopt a holistic memory access profiling approach to enable a better understanding of program-system memory interactions. We have developed a two-pass tool adopting fast online and slow offline profiling, with which we have profiled, at the variable/object level, a collection of 38 representative applications spanning major domains (HPC, personal computing, data analytics, AI, graph processing, and datacenter workloads), at varying problem sizes. We have performed detailed result analysis and code examination. Our findings provide new insights into application memory behavior, including insights on per-object access patterns, adoption of data structures, and memory-access changes at different problem sizes. We find that scientific computation applications exhibit distinct behaviors compared to datacenter workloads, motivating separate memory system design/optimizations.

KEYWORDS
Memory profiling, object access patterns, workload characterization, tracing, data types and structures

ACM Reference format:
DOI: 10.1145/3126908.3126917

1 INTRODUCTION
The memory subsystem is crucial to computational efficiency and scalability. Today’s computer systems offer both high DRAM capacity/performance efficiency and a deep memory cache layer that mixes HBM, DRAM, and one or more NVMs. This trend has enabled in-memory processing in multiple application domains [24, 51, 63]. On the other hand, in recent years there has been a dramatic increase in both the number of cores and the application (VM) concurrency on a physical node, as seen in supercomputers, datacenters, and public/private clouds. As a result, the memory-bandwidth-to-FLOP ratio has been steadily declining, e.g., from 0.85 for the No. 1 supercomputer on Top500 [54] in 1997 to 0.01 for the upcoming projected Exaflop machines [43]. For both commercial and scientific computing, it remains important to optimize programs and systems for efficient memory access.

Such optimizations have to build upon an understanding of the memory access behavior of the program itself, which is highly challenging. Unlike I/O or network requests, which are commonly traced and analyzed, memory access requires temporal and spatial analysis, is a fixed high-speed transaction, and is expensive to perform online analysis. To this end, many optimization techniques adopt offline memory access profiling [16, 48, 59, 62] that collects information during separate profiling runs to guide decision making in future "production runs".

Although much more affordable, existing offline memory access profiling techniques mostly collect high-level statistics (such as total access volume and memory references per thousand instructions) [18, 19, 21, 38] or full access sequences [27, 35] (including complete/sampled traces and derived information such as reuse distance distributions). These properties and data, while useful, are based on logical addresses and are disconnected from program semantics. Also, the same application cannot be expected to have the same memory access pattern for different input problem sizes or input data. Moreover, HPC (scientific computing) applications in particular have received much less attention from existing memory allocation/access characterization studies.

In this work, we address these problems by providing more intuitive profiling results that bridge runtime low-level memory references and their high-level semantics (e.g., data structures that reflect programmers’ view of execution). This means that addresses
need to be mapped back to "objects" allocated at runtime and further mapped to "variables" in the source code. To do this, we design and implement a two-pass profiling tool for variable-level access pattern analysis. Our tool performs the above address-to-object and object-to-variable mapping, which facilitates subsequent online memory trace processing to report object-level access behavior. This two-pass framework provides the option of collecting object allocation/access information at different levels of detail and overhead, with the first (fast) pass available for independent deployment.

Using this tool, we have profiled at the variable/object level, a collection of 38 representative applications spanning major domains (personal computing, HPC, AI, data analytics, graph processing, and datacenter servers), each at three different problem sizes. For each application, we have further identified the major program variables, which are most dominant in memory consumption, and have collected detailed access behavior profiles such as object size and lifetime distribution, spatial density in accesses, read-write ratio, sequentiality, and temporal/spatial locality. For such major variables, we have performed considerable manual inspections to identify the type and purpose of the data structures in the source code, investigating which major data structures are adopted and how they are accessed.

We have performed detailed profiling result analysis, especially focusing on comparing the behavior of scientific vs. commercial/personal computing applications. To the best of our knowledge, our study is the first to perform such a thorough, comparative analysis between these application classes. We organize our findings into 7 key observations, and discuss their practical implications, illustrating how they shed light on the complex heap memory allocation and access activities of modern workloads. Our results reveal that scientific applications possess multiple qualities enabling efficient combination of offline and online profiling (such as fewer, larger, and more long-lived major data structures), and more uniform scaling behavior across variables when computing problems of different sizes. However, they still have dynamic and complex memory access patterns that cannot be properly measured by sampling billion-instruction or shorter episodes even during their stable phases.

2 PROFILING METHODOLOGY

2.1 Objects and Variables

We first define the building blocks that form the basis of our profiling framework, namely objects and variables. Programs statically or dynamically allocate space for data from memory. Conventionally, such a piece of allocated memory is called an "object", instantiating a "variable" for reference in the program. Due to the direct link between variables and program semantics, it is more interesting and less redundant to study allocation/access at the variable level. However, it is not straightforward to build the mapping between variables and objects. For example, objects might be allocated in a utility function, whose addresses are returned to callers performing very different tasks.

In this work, we adopt the methodology of identifying variables by the call-stack, following existing work [4]. For heap objects, we consider the contiguous heap memory region allocated by dynamic memory allocation functions, such as malloc(), calloc(), or realloc() in C/C++ and allocate() in Fortran, as an object. We then define the entity grouping all the objects allocated within the same call-stack as a variable. Here, the call-stack encompasses a series of function names and return addresses, from the main function all the way to the final heap memory allocation call. Therefore, two malloc() calls within the same function are considered allocating for different variables.

To illustrate these definitions, Figure 1 shows sample code from SPEC m11c [53]. Here, eight different objects (tempvec[0], tempvec[1], ..., tempvec[7]) are created by the first calloc() in the for loop (each through a separate call to calloc()). However, since they share exactly the same call-stack, these objects are considered members of the same variable. The object temp_x, on the other hand, is created through another call-stack, and hence belongs to another variable. For static/global variables residing in the data segment, such object-variable mapping is easy and one-to-one.

2.2 Two-Pass Variable/Object-Level Profiling

We now present the design of our variable/object memory behavior profiling framework, shown in Figure 2. The main idea is to perform two-pass profiling to avoid the prohibitive time and space overheads of capturing individual references to all objects.

![Figure 1: Sample code from SPEC m11c](image)

To illustrate these definitions, Figure 1 shows sample code from SPEC m11c [53]. Here, eight different objects (tempvec[0], tempvec[1], ..., tempvec[7]) are created by the first calloc() in the for loop (each through a separate call to calloc()). However, since they share exactly the same call-stack, these objects are considered members of the same variable. The object temp_x, on the other hand, is created through another call-stack, and hence belongs to another variable. For static/global variables residing in the data segment, such object-variable mapping is easy and one-to-one.

![Figure 2: Two-pass profiling workflow](image)

**Fast pass to collect per-object data** We first run the target application and perform a fast pass of profiling to quickly collect per-object information such as object size, allocation/deallocation time, and allocation call-stack. This is done via a custom memory allocation library, a shared library that intercepts every call to malloc, calloc, realloc, free in C/C++ and mmap in Fortran (note that large dynamic memory allocation is managed by mmap in Fortran). We perform online processing of call-stacks, using hashing to find object-to-variable mapping and also to avoid saving redundant call-stacks. The output of the fast pass is the aforementioned per-object information, including the common call-stack for objects identified...
to belong to the same variable. By avoiding detailed memory reference tracing or full call-stack output, this fast pass introduces small to moderate execution slowdown (1.0 × to 42.9 × in our experiments using 30 + applications, with median at 1.2×).\(^1\)

**Offline processing** Next, we process the collected per-object information, such as object lifetime and size, to identify a much smaller subset of target variables offline. In this study, we select up to 10 major variables per application for detailed access behavior tracing and analysis. The criteria for identifying these variables is described in detail in Section 4.

**Slow pass using a custom Pin tool** With the scope of variables reduced to this focus group, we next perform the second, slow-pass profiling, using the popular Intel Pin tool \([35]\) for memory tracing, which we have customized with special online processing. In this pass, we rerun the target application in our customized Pin environment, where this custom Pin tool matches on the fly, the virtual address of each memory reference with virtual address extents of target objects (identified also online through call-stack matching to known major variables). A hash table is used to speedup such online search. However, Pin-based memory tracing alone is costly and the above online search could turn out to be highly expensive when an identified major variable has a large number of member objects. In our experiments the slow-pass profiling incurs slowdown between 30× and 3000× (median at 226×).

This two-pass process enables profiling long-running applications by performing online per-object access characteristics analysis at a fraction of the overhead of traditional tracing, which easily accumulates TBs of trace data within minutes or seconds of execution. Combined with offline analysis, it also allows the second-pass profiling to focus on selected variables/objects of interest to users.

### 3 EXPERIMENT SETUP

#### 3.1 Execution Platforms

Most of our experiments are conducted on a machine with two Intel 12-core E5-2697 v2 CPUs and 256GB memory, running Ubuntu 14.04 with Linux kernel 3.13. For running parallel experiments with multi-threaded or multi-process executions, we use a 10-node cluster with each node consisting of two Intel 12-core E5-2670 v3 CPUs and 128GB memory, running CentOS 7.0 with Linux kernel 3.10.

#### 3.2 Application Workloads

Our goal is to profile the memory-access patterns of workloads that represent a wide range of popular contemporary applications. We therefore sampled from several major categories of computing workloads running on today’s typical platforms (desktops, servers, HPC clusters, and datacenters). The majority of our test applications are from well-known benchmark suites, while the rest are real-world applications, including both C/C++ and Fortran programs.

**SPEC CPU2006**: SPEC CPU2006 \([53]\) is a widely used, industry-standard benchmark suite that includes 31 common end-user applications. We study all 17 SPEC programs written in C/C++ (both integer and floating point).

**Real-world HPC Applications**: We sample three real-world applications, representing different categories of parallel scientific codes. gromacs \([1]\) is a widely used open-source computational chemistry software, performing dynamic bio-molecule simulation. mpiBLAST \([13]\) is a popular parallel implementation of the NCBI BLAST \([29]\) biological sequence alignment tool, routinely used in bioinformatics research. LAMMPS \([32]\) is a classical molecular dynamics code performing parallel particle simulation at the atomic, meso, or continuum scale.

**NPB**: The NAS Parallel Benchmarks (NPB) \([42]\) contain a group of representative HPC codes: the original 5 kernels (IS, EP, CG, MG, and FT), 3 pseudo-applications (BT, SP, LU), and UA, a more recently added benchmark with unstructured adaptive meshes. As we select NPB to study scientific applications’ memory access patterns, we profile 7 of the above members, excluding IS (integer sorting) and EP (almost all-stack accesses, with very small memory footprint).

**PARSEC**: PARSEC (Princeton Application Repository for Shared-Memory Computers) \([8]\) is a well-known benchmark suite for evaluating executions on chip-multiprocessors, containing 12 applications performing tasks such as simulated annealing, data deduplication, and data mining. The benchmark suite covers a wide spectrum of memory behavior, in terms of working set size, locality, data sharing, synchronization, and off-chip traffic. Again we profile all 7 PARSEC applications that are written in C/C++.

**PBBS**: The Problem-Based Benchmark Suite (PBBS) \([52]\) by CMU is a relatively recent collection containing 16 representative data-center tasks. Considering the scope already covered by the above application categories, here we choose to sample from PBBS, two representative graph applications: Breadth First Search (BFS) and Spanning Forest (SF).

**Other Datacenter Workloads**: We further sample two members from the new MIT TailBench suite \([30]\) for request-driven applications. s10 [35] is an in-memory database, running TPC-C with massstree [37] as a building block. dnns is an AI application based on OpenCV [9], using deep neural network-based auto-encoder to identify hand-written characters.

#### 3.3 Profiling with Multiple Problem Sizes

A major goal here is to understand programs’ memory allocation/access behaviors when processing different problem sizes. For all 38 applications, we profile executions under three different problem sizes, referred to as “small”, “medium”, and “large” for brevity, indicating the relative problem sizes in our experiments. Note that standard benchmarks usually come with multiple problem sizes (such as the “test”, “train”, and “ref” sizes for SPEC). For several other applications (such as mpiBLAST and gromacs), we set up three problem sizes based on their documentation and available input datasets. When not studying objective behavior under different problem sizes, we report results only from the “large” run.

### 4 SUMMARY OF PROFILING TARGETS

#### 4.1 Overall Variable/Object Behavior

We begin by providing an overview of variable/object-level characteristics observed in the 38 workloads. Table 1 provides summary statistics on the objects, variables, and memory footprints identified in all the profiled applications. In our experiments, we run every application multiple times while varying the input data size. Due
to space limit we only list measurements from the run with the largest problem size we profiled, e.g., “ref” for SPEC, “native” for PARSEC, and class “B” for NPB.

For each application, Table 1 lists the total number of variables/objects, the maximum number of concurrent objects, and the distribution of object sizes (min, max, and median). Though not shown in the table, we record object lifetime as the elapsed time between memory allocation and de-allocation of a given object (such timestamps are also used in calculating the number of concurrent objects). We also calculate variable lifetime as the average lifetime of all profiled objects that belong to this variable.

4.2 Focused Study of Major Variables

To understand applications’ detailed memory access behaviors, we “zoom in” to a group of representative variables of each application, for which we collect, analyze, and report detailed profiling results.

We select major variables with the top-10 largest per-variable footprint (peak combined memory consumed by its concurrent member objects). We discard variables whose largest object (in the “large” run) does not reach the page size (4KB in our systems). This criterion thus targets the most footprint-consuming variables, by considering their concurrent member objects (sharing the same allocation call-stack) at any given time during program execution.

Such filtering produces 268 major variables in total across all 38 applications, as many of them have fewer than 10 qualifying variables. Note that when re-scoping to major variables, we have lost three of the 38 applications, namely si10, omnetpp, and astar, who do not have any qualifying variables. The last 2 columns in Table 1 summarize the number of major variables and the average object count across them, in each application.

Before investigating more detailed per-variable behavior, we first examine how homogeneous are objects that belong to the same major variable. This analysis serves two main purposes. First, it helps us study the hypothesis that a variable’s member objects behave similarly in terms of access patterns and locality. Second, a practical reason for doing this is that detailed object-level profiling is quite time-consuming. Especially, when there are many runtime objects to be profiled, each memory access has to be compared to the address interval of these target objects, potentially causing 1000× slowdowns. Understanding object homogeneity thus helps us reduce such overhead, by sampling representative objects only.

Table 1 shows that in most cases, each major variable owns fewer member objects, compared to the statistics of all variables in the application. This is particularly the case for real-world HPC applications, mostly with no more than dozens of objects per major variable, and NPB applications, with only one object per variable. Furthermore, SPEC applications performing scientific computing (e.g. milc for quantum chromodynamics and hmpcr for gene sequence search) also possess significantly fewer member objects per major variable, unlike most commercial/desktop applications.

As a side note, this study does show that the NPB programs, as widely used HPC benchmarks, display rather simplistic object allocation behavior (one object per variable) than larger, real-world applications.

To check their behavior consistency, we sampled multiple member objects of each variable and examined their access attributes (to be described later). We observe that in general, objects associated with the same variable are found to possess identical or highly similar behavior, especially in access patterns. So unless otherwise noted, we report results from the largest member object in each major variable (the first one allocated if tied in size).

4.3 Profiling Parallel Applications

Nine programs in our study are multi-process parallel codes (gromacs, mpiBLAST, LAMMPS, and all NPB members except UA). We ran these applications with 4 processes and conducted per-process profiling. We observed in general very minor inter-process memory differences in variable allocation and access behaviors, intuitively due to their SPMD nature and existing optimizations to enhance load balance (lower than 5% difference). The only exception is NPB CG, which uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large sparse symmetric matrix. Its unstructured matrix involved in vector multiplication causes irregular access patterns across processes.

In addition, 16 programs (NPB, PARSEC, and PBBS) support multi-threaded execution. We performed two sets of profiling to (1) compare access patterns among threads and (2) compare process-level aggregate access patterns between single-thread and 4-thread runs. Again, we find allocation and access patterns to be similarly consistent across threads, while the combined multi-thread access patterns are consistent with those from a single-thread run computing the same input problem, except for limited differences in ordering of accesses to shared variables.

Considering the high degree of homogeneity across processes/threads in behavior, for the rest of this paper we focus on studying allocation/access activities within a thread/process.

5 RESULTS AND ANALYSIS

To facilitate comparison, we have classified all individual applications roughly into two groups, mainly by checking their domain/purpose: scientific computing applications (Sci-comp) and commercial/desktop applications (Others). Note that such manual classification is by no means rigid—there are commercial scientific applications, for example. Here, we consider the Sci-comp group containing the programs more likely to run on conventional HPC platforms, such as in-house clusters, cloud-based virtual clusters, and supercomputers. This classification results in 15 programs in the Sci-comp group, including all NPB members tested, all real-world HPC applications, and milc, dealII, soplex, hmpcr, 1libquantum from SPEC in Table 1. The Others group contains the rest of the workloads (23 programs).

5.1 Variable/Object Size and Concurrency

We begin by looking at object/variable counts and sizes. Applications vary widely in the number of objects and variables they use. They allocate from as few as 3 to over 267 million heap objects (median at 78,447 and 90th percentile at 61,217,343). The number of variables is much smaller, ranging from 3 to 49,690 (median at 97 and 90th percentile at 2,777). This indicates that, for most application, the vast majority of heap objects share a much smaller set of

These applications do possess an important class of behavior, with small objects composing larger data structure such as trees. While this work focuses on larger objects due to online profiling cost, we plan to revisit them in future work.
allocation call-stacks. The maximum number of concurrently-live objects for each application runs in the middle, from 3 to over 13 million (median at 643 and 90th percentile at 1,992,716).

Furthermore, our profiling results show that scientific applications tend to have fewer objects per major variable: a median of 1 for Sci-cmp vs. 2579 for Others. Not surprisingly, scientific applications tend to have larger objects, with minimum/median/maximum object sizes at 8.6MB/27.8MB/240MB in the Sci-cmp group and 0.5MB/17.0MB/153MB in Others, respectively.

With respect to concurrent objects, we find that with the exception of dealII, an adaptive finite element method library that creates more than 13 million concurrent objects, the Sci-cmp group members possess significantly lower concurrent object counts. Finally, in scientific applications, each variable appears to have fewer objects. In particular, when we examine only the major variables, there is dramatic contrast between the Sci-cmp and Other's groups in terms of object count per variable: 90th percentile at 24 and maximum at 53 for Sci-cmp, while 90th percentile at 1044 and maximum at over one million for Others.

### Observation 1
Applications exhibit highly diverse patterns in the number and size distributions of their variables and objects. However, in general scientific applications have fewer and larger variables, compared with commercial or desktop workloads. Scientific applications also tend to have significantly fewer number of concurrent objects during their execution.

### Implications
The large number of memory objects created in applications makes tasks like memory allocation and garbage collection performance-critical. Meanwhile, object behavior, both across applications and among variables within the same application, exhibits wide variability. This makes judicious placement (for NUCA [6, 12, 26], NUMA [3, 14], emerging hybrid memory systems spanning multiple levels of heterogeneous memory hardware [16], and out-of-core systems [57]) highly important and profitable. In particular, with fewer and larger objects, scientific computing applications are good candidates to explore runtime memory object placement, as such optimization is likely to be more cost-effective compared to applications with millions of small objects.

<table>
<thead>
<tr>
<th>ID #</th>
<th>Benchmarks</th>
<th>Total # of variables</th>
<th>Total # of objects</th>
<th>Max # of concurrent objects</th>
<th>Distribution of Object Size (Bytes)</th>
<th># of major variables</th>
<th>Avg. object count per major variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>freqmine</td>
<td>9,400</td>
<td>357,415</td>
<td>139,155</td>
<td>Min 8</td>
<td>Median 263</td>
<td>Max 1,024</td>
</tr>
<tr>
<td>27</td>
<td>gromacs</td>
<td>9,433</td>
<td>2,380,375</td>
<td>139,155</td>
<td>Min 8</td>
<td>Median 263</td>
<td>Max 1,024</td>
</tr>
<tr>
<td>28</td>
<td>gromacs</td>
<td>9,433</td>
<td>2,380,375</td>
<td>139,155</td>
<td>Min 8</td>
<td>Median 263</td>
<td>Max 1,024</td>
</tr>
</tbody>
</table>

Table 1: Summary of variable-level and object-level statistics
5.2 Variable Lifetime

We now examine how variable lifetime, i.e., the average lifetime of its member objects, is distributed across application variables, particularly in relation to variable size. Over all applications, we find that the distributions of variable sizes and lifetimes are quite skewed: 97.0% of variables have an average size that is under 1% of the largest variable in the same application, and 54.4% of variables report lifetimes under 1% of their application’s execution time. Additionally, we find no evidence of a strong correlation between variable size and variable lifetime, as indicated by a low Pearson coefficient ($R = 0.089$ for Sci-comp applications; $R = 0.016$ for Others) with high statistical significance ($p-value < 0.001$).

When we limit the examination to major variables only, we find that the lifetime distribution becomes quite disparate between the two groups. Major variables in the scientific applications are much more likely to be long-lived, with 25% percentile and median normalized lifetimes at 0.08 and 0.99, respectively, compared to 0.01 and 0.19 for the Others group.

**Observation 2** We find that the vast majority of objects are short-lived, in both scientific and other applications. However, the majority of footprint-consuming (major) variables in scientific applications are indeed long-lived, while this does not apply to most of the commercial/desktop applications. In addition, there is no strong correlation observed between variable size and lifetime.

**Implications** Scientific applications have long-lived major variables (many with a single object as indicated by earlier results). This again makes these codes appealing for memory allocation/placement optimizations, as most “important variables” live throughout the execution, paying optimization overhead once yet enjoying its benefit for a significant portion of the execution (and potentially for subsequent runs of the same application).

On the flip side, such long-lived and large data structures in scientific applications, once allocated to a faster memory, might tie up precious space there. Applications in the Others group, however, have dramatically higher fractions of short-lived objects, allowing more flexible, interleaved utilization of faster memory layers. As our study also indicates that there is no apparent correlation between object size and lifetime, short-lived major variables in Others are potentially good candidates for exploiting optimized placement at affordable overheads.

5.3 Problem Size Scaling

Running the same workload with different input sizes is common in most computing scenarios, scientific, commercial, or personal. One of our goals in this study is thus to understand the behavior changes of applications under different problem sizes. Throughout our analysis here, we focus on the identified major variables.

We start by examining the simplest attribute, the size of the leading (largest) object of each variable, and categorize their behavior into three groups: **fixed**, where object size stays constant across three problem sizes; **scaling**, where object size grows with increasing problem size; and **irregular**, where object size shrinks or exhibits non-monotonic changes.

Figure 3 plots the per-application distribution of major variables by these three scaling behavior types. The left side shows the 15 programs in the Sci-comp group, while the right side shows 20 from Others (recall that 3 among the Others applications do not have any major variables identified).

We observe that the two groups exhibit highly distinct behaviors. 117 out of 127 total major variables in the Sci-comp group scale with a growing problem size, with 9 fixed and 1 irregular. The Others group shows much more diverse breakdowns. Among all 141 major variables there, 80 are scaling, 57 are fixed, and 4 are irregular. This indicates that typical scientific applications exhibit more uniform tendency to have their major variables grow with input problem size, while for the other applications, there is a significant chance that major variables do not grow when computing a larger problem. Intuitively, not all footprint-consuming data structures (such as buffers or hash tables) are associated with the overall problem size. Plus, for some benchmarks, the given problem sizes may scale in parameters not affecting space consumption of storing the “main subject”, such as computing more steps in chess games (more computation yet not necessarily larger datasets).

To further analyze, we exclude the small set of “irregular” variables and then roughly classify applications into three categories: **fixed**, where all major variables are “fixed”; **scaling**, where all major variables left are “scaling”; and **hybrid**, where major variables exhibit both behaviors. Below we list the applications in each category (35 in total, excluding the three with no major variables identified):

- **Fixed (5)**: sjeng, gobmk, povray, h264ref, mpiBLAST.
- **Scaling (23)**: milc, dealIII, soplex, hammer, libquantum, gromacs, LAMMPS, CG, MG, FT, BT, SP, LU, ferret, perlbench, gcc, mcf, sphinx3, canneal, streamcluster, BFS, SF.
- **Hybrid (7)**: dnn, bzip2, bodytrack, freemine, vips, dedup, xalancbnk.

We find 5 of the 35 applications in the “fixed” group, with all major variables (except very few “irregular” ones) having constant sizes across problem sizes. A closer look reveals that they belong to the aforementioned case where problem size scaling is done by incurring more computation rather than processing larger datasets.

The “scaling” and “hybrid” groups of applications are easier to explain. For these groups, we further examine if their scaling variables grow at consistent speed. We analyzed the growth factor of individual scaling variables across problem sizes, and found that multiple variables are often grouped under several “scaling speeds” (which can be quite different from one another). Four applications have all 10 major variables sharing the same growth factors and no
application has more than 3 distinct speed groups. This behavior is consistent across the Sci-comp and Other's programs. In particular, many of the scientific applications we profiled have 3 variable size growth speeds, corresponding to the x, y, and z dimensions of their simulation subjects. In these applications, major variables belonging to such scaling speed groups often follow identical scaling factors across problem sizes.

**Observation 3** Sci-comp applications exhibit highly uniform scaling behavior, with major variables almost always growing in size when computing larger problems. Other's applications have much weaker consistency in this regard. For both groups, major variables of the same application that scale with problem sizes often cluster into up to three different “scaling speeds”.

**Implications** When we run the same application at a different scale, variable sizes may or may not change significantly. Scientific applications emerge as a rather predictable group here, with their major variables dominantly growing in size when the input problem size is scaled up. Commercial/desktop applications, in contrast, exhibit more diverse behaviors.

This makes optimizations based on offline profiling more challenging. E.g., unlike assumed by X-Mem [16], a significant portion of important variables have fixed size across problem sizes in commercial/desktop applications. Even with scientific applications, nearly uniform scaling behavior does not imply a uniform size growth rate among scaling variables of the same program. Instead, certain amounts of online profiling should be conducted to supplement variable-level memory access characterization results based on past executions.

To this end, our study also reveals potential solutions by finding that objects can be tied to each other in “scaling speed”, while there might be up to three such distinct “speed groups” among the scaling variables in an application. This implies that by sampling a few representative variables, we can rather reliably predict the sizes of major variables. Such capability might be of more interest for commercial/desktop applications, though, which have more short-lived yet footprint-consuming objects.

### 5.4 Object Footprint

We next examine the *total object footprint* (the total size of all active non-stack objects – footprint for short) during each application’s execution, in an attempt to answer the question of “how early does an application typically approach its peak allocation?”

Figure 4 shows the fraction of applications reaching three footprint milestones along the execution timeline. The majority of applications profiled make the bulk of their overall footprint fairly early in their execution: 33 out of the 38 applications reach 80% of peak footprint by relative time of 0.28, 90% by 0.35, and 100% by 0.99; half of them reach 80% by 0.02, 90% by 0.05, and 100% by 0.11. The Sci-comp group, though not individually plotted, have 13 of their 15 members reach peak footprint before the relative execution time of 0.3 (outliers to be explained below).

Compared to the case with reaching the 90% footprint, significantly more applications arrive at their full footprint later in their execution. By inspection, we found that this is due to data structures allocated during the result processing and finalization stage, as demonstrated by the surge near the end of execution in Figure 4. The vertical line in Figure 4 indicates that by 20% time into execution, 32, 28, and 23 applications out of 38 have reached the 80%, 90%, and 100% milestones, respectively.

![Figure 4: CDF of number of applications that reach 80%, 90%, and peak footprint at certain times during execution](image)

**Figure 5: Footprint evolution of 5 outlier applications**

We next take a closer look at the behavior of applications whose peak-memory consumption patterns deviated significantly from the average case. Figure 5 shows the five “late bloomers” identified from our footprint analysis. Interestingly, these outliers showcase different scenarios generating continuous or late footprint growth. For the three Other’s applications: *perlbench* (performing Perl interpreter evaluation with multiple mail processing scripts) allocates ad-hoc objects continuously; *canneal* performs simulated annealing for chip design optimization, which has growing footprint due to gradually involving more elements; *silo* is a TPC-C style in-memory database benchmark, incurring steady growth in memory consumption during the initial database building, which continued at a slower speed in the query processing stage. The Sci-comp outliers are *dealII* and *libquantum*. *dealII* performs adaptive finite elements analysis with error estimation where dynamic refinement gradually increases footprint by deallocating objects (hence the brief drops) and allocating new and larger ones. *libquantum* performs simulation of a quantum computer, with a finalization phase where heap memory consumption grows by 45%.

Finally, we verified that such footprint evolution behavior stays consistent across multiple input problem sizes; we omit the detailed plots here for space purposes.

**Observation 4** Applications typically reach a significant portion (80%) of their peak heap footprint quite early in their execution, and 24 of applications reach their total peak footprint by 20% of execution time. In particular, most scientific applications reach their peak footprint very early during their executions.
**Implications** Our memory footprint study indicates that most applications reach near-peak footprint rather early in their execution, and this behavior persists across problem sizes. Such profiling results may guide cloud/datacenter managers in VM migration and load consolidation.

Also, the early arrival of peak memory allocation in scientific applications (likely a side effect of their long-lived major variables) enables online profiling, if necessary, to be swift. Collected runtime object characteristics can be used for memory allocation and placement optimization, without the need for continued online monitoring and profiling.

### 5.5 Major Variable Data Structure Types

Next, we study the per-variable semantics and access patterns. Fortunately we could carry out the much more detailed monitoring and examination here, after reducing the profiling scope to the identified major variables only. This includes a significant amount of manual code study to identify the data structures of the major variables, categorized into the following data structure types (DSTs):

- **1-D hash table:** Typically identified by variable name or the use of hash function;
- **1-D buffer:** Array for intermediate storage (typically for input/output data) outside the main computation of an application/function, identified by name or lifetime plus processing behavior;
- **1-D index:** Array indexing into another data structure, identified by name or the use of indexing operations (after excluding the above three types);
- **1-D bitmap:** Identified by name or the use of bit-level operations (after excluding the above three types);
- **1-D vector:** Default category for 1-D arrays not identified as the above four types;
- **2-D matrix:** 2-D array storing homogeneous data elements, identified by name plus (x, y)-style reference pattern;
- **2-D collection:** Set of arrays presenting independent objects, such as different attributes of the same data entity, identified by name or element data types.
- **M-D matrix:** Multi-Dimensional array storing data elements, identified by data semantics plus (x, y, z, ...)-style references.

**Figure 6:** Data structure type distribution of the two application groups, by count

![Figure 6](https://example.com/fig6.png)

Figure 6 presents the breakdown of these DSTs across major variables in the Sci-comp and Others groups (127 and 141, respectively), in terms of variable count. Generic arrays (1-D vectors) are a leading DST for both classes of applications, comprising 39% of major variables in the Sci-comp group, and 34% in Others. Interestingly, multi-dimensional matrices (MD-MTX) co-leads the Sci-comp major variables (also at 39%), but it is completely absent in Others. Similarly, DSTs such as hash tables, buffers, bitmaps, and index arrays, are much better represented at the Others' side.

We also repeated the distribution calculation by total object size and by total footprint. Due to space limit we omit related figures and detailed discussion. In summary, we found the types array (1-D, 2-D, and multi-D), index, and buffer to be more space/footprint-consuming. All the other DSTs combine to occupy much smaller footprint (1.4% for Sci-comp and 3.4% for Others).

**Observation 5** Scientific and commercial/desktop applications have highly contrasting distributions of the data structure types defining their major variables: multi-dimensional matrices are a prominent DST for Sci-comp but missing from the Others data structure list, while DSTs such as bitmap and hash table are heavily favored by Others' applications. In addition, certain DSTs, such as array, index, and buffer, are more footprint-consuming.

**Implications** Our results confirm common perceptions that program semantics in scientific computing applications differ from other programs. However, the degree of differences (e.g. regarding multi-dimensional matrices) is somewhat surprising. Such disparate data structure preference may motivate different design or optimization along the memory hierarchy, such as annotation by programmer or compiler for data placement [16, 39], to assist runtime decision making.

### 5.6 Per-object Memory Access Patterns

Next, we study per-variable detailed access patterns, as defined below. We report profiling results on the leading object of each major variable.

**Definitions:** The read (write) ratio of an object is the fraction of the object’s reference volume from read (write) operations, with the two ratios adding up to 1. This metric is particularly important for hybrid memory systems, which incorporate technologies that have asymmetric read/write costs (such as STT-RAM) and/or lack in-place updates plus endurance concerns (such as NAND flash).

For a given reference sequence to an object, we consider a reference sequential if it accesses data adjacent to that referenced in the previous access. Accordingly, we define an object’s access sequential ratio as the fraction of its reference volume that are sequential.

One more attribute critical to memory performance is locality. We consider both temporal and spatial locality and quantify them using Gupta’s method [22]. An object’s temporal locality is calculated as the fraction of references accessing the same cacheline visited within a certain “near future” window over all references. Following another existing study [40], we set the window size to 1,000 instructions in our experiments. Similarly, an object’s spatial locality is calculated as the fraction of references where data in its spatial neighborhood (8 cachelines in our experiments, excluding itself) has been accessed within the previous 1,000 instructions.

Finally we examine the major variables’ per-object access density (density for short), defined as an object’s total reference volume divided by its size. It describes how many times each byte of an object is referenced on average, alluding to their relative importance, which is particularly useful for judicious object placement.

**Summary of Read/Write and Address Distribution** We examined the major variables’ read ratio, sequential ratio, and temporal/spatial locality and compared their distribution between the
two application groups. Overall, the Sci-comp and Others' groups do not show significant difference in these attributes, though the Sci-comp are found to be slightly more read-intensive and have slightly higher locality, both temporally and spatially. The two groups also have a similar ratio of accesses being sequential. Due to space limit, we omit detailed results here.

Multi-Dimensional Matrix Access Patterns Recall that multi-dimensional matrices (MD-MTX) are very common on the Sci-comp side (39% by major variable count and 64% by footprint) and completely absent on the Others' side. As a case study, we looked into these variables (49 from 7 Sci-comp applications) and found that they do possess unique patterns seldom found in Others'.

Table 2 categorizes these 49 MD-MTX variables (all from NPB) into three distinct behavior types, plus one combination type, as described below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Application name (# of variables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse</td>
<td>CG(2), MG(1), FT(1), LU(1), UA(5)</td>
</tr>
<tr>
<td>Sequential</td>
<td>MG(2), FT(2), LU(2), UA(1)</td>
</tr>
<tr>
<td>Random</td>
<td>CG(2), UA(2)</td>
</tr>
<tr>
<td>Mixed (Seq.+Spus.)</td>
<td>BT(10), SP(10), LU(6), UA(2)</td>
</tr>
</tbody>
</table>

Table 2: Categorization of MD-matrices by reference patterns

Sparse stands for typical access patterns in applications with stencil/diagonal computation models (though the matrices are often dense). Here the program updates a matrix element based on a function of certain neighboring elements, or traverse a matrix diagonally. With this type, the object is accessed in small chunks (4 or 8 bytes in our results), with large address offset in between. The offsets are not uniform (i.e., not strictly “strided”), but concentrated at several to dozens of distinct values. 38 major variables from 7 applications possess this behavior, including 26 with hybrid sequential-sparse accesses. By studying the code, we have found that within such variables accesses are predominantly sparse in nature, with sequential accesses during initialization/finalization.

Sequential here stands for dominantly sequential accesses, where long sequential access runs are separated by infrequent strides. The length of such sequential access runs depends on the problem size, but in most cases are significantly larger than the cache line size and often larger than page size.

Random stands for random accesses, where each access is strictly one data element and the address offset values between adjacent accesses are found to be random. There are only 4 variables from 2 applications with this access behavior. Studying the code shows that all of them have accesses dictated by index arrays.

Spatial Density by Application Figure 7 shows the per-application density distribution among major variables by count. Density varies a lot among variables, from very sparse (18 variables under 0.01) to very dense (26 above 100,000). Also, most applications have rather diverse density across their variables, highlighting that peer variables have vastly different access heat levels. Sci-comp applications overall have the distribution biased toward the dense end, with 65% of major variables having density of over 100, compared to 34% among Others'.

Finally, Figure 8 shows the distribution of variance observed with growing problem sizes, by plotting the number of major variables that are either: fixed, where changes in density across problem sizes are within 10%; scaling, where changes in density are above 10% and grow monotonically with problem size; and irregular otherwise. At a first glance, most variables have either fixed (93) or scaling (103) density behavior, while most applications have variables belonging to both groups. Six applications (including most of those with “fixed-only” behavior in object sizes), however, have most of their variables scaling in density, implying that their major data structures carry higher computation complexity or are increasingly reused (such as buffers in streaming applications). Several other applications, including BFS and dnn, have entirely or mostly fixed density across problem sizes, implying stable computation-to-data ratio or stable reuse degree. Sci-comp applications, compared to Others', appear to have more variables with scaling behavior (71% vs. 46%).

Observation 6 Major variables in scientific applications have similar distribution as those in other domains in terms of read/write ratio, sequentiality, and locality. However, multi-dimensional matrices, favored by Sci-comp group only, possess unique “stencil/diagonal” access patterns with large but regular leaps between small access units. Spatial density varies much across variables in the same applications may or may not scale with the problem size. In general, Sci-comp applications’ major variables also tend to have larger access density.

Implications The sparse access pattern very common in accessing MD-MTX variables form interesting multiple sequential streams. Though appearing sparse with fine access granularity, such codes often enjoy high temporal/spatial locality (as confirmed by our measurement) due to their long-term sequential sweeping access
manner, which contrasts with random accesses in commercial applications (e.g., databases and graph).

On a different note, spatial density studied in this paper may be a good indicator of a variable’s “worthiness” to be placed on closer/faster layers like the last-level cache or DRAM. Meanwhile, our results suggest that within an application, both the absolute and relative density can change significantly across problem sizes, so offline profiling for judging heat levels can be misleading.

5.7 Sampling Window Size for Access Profiling

Memory access profiling, trace collection, and trace-driven simulation are expensive. It is common for practitioners to study a partial, yet believed to be representative episode of the execution. Here we analyze the effect of the length of such episodes.

Figures 9(a) and 9(b) shows time-series measurements of two sample per-object attributes, access volume and read ratio, for three major variables in SPEC bzip2 during a full execution. Each data point shows the average value over a window of 1 billion instructions. These plots indicate that (1) variables’ lifetimes and their overlap create variation in a scope much larger than billions of instructions, and (2) each individual variable’s access behavior also fluctuates significantly over the many billion-instruction episodes.

To quantify the variability in these access patterns under different episode lengths, Figures 9(c) and 9(d) plot the coefficient of variation (CV, standard deviation divided by mean) in readings of these two attributes with varying episode length (in number of instructions). The bars plot the median CV value across all major variables in an application group, with error bars showing the minimum and 75th percentile CV values.

As expected, increasing the episode length reduces inter-episode variance. However, such trend stops or slows down significantly after 5 billion instructions, signaling that the inter-episode variance at this point is likely attributed to longer-term phases in execution. Also, Sci -comp programs appear to be more stable in access volume, but less so in read ratio, compared to Other’s programs. This is reasonable considering typical iterative scientific program behavior, where variables are processed at rather predictable speed, but may alternate between read and update passes.

**Observation 7** For both scientific and commercial/personal applications, aggregate and per-variable access behaviors may vary significantly across adjacent billion-instruction episodes. Increasing episode length removes short-term “local” variances, while there remains longer-term behavior variation due to program phase changes.

**Implications** Many prior studies on hardware architecture, memory hierarchy design or program access pattern analysis observe and test small sample segments from program execution, with typical lengths of under 200 million instructions [17, 31, 33, 41] and 200-500 million instructions [20, 25, 36, 46, 50, 58]. Our experiments reveal that one or a few such samples often fail to capture a full picture of the programs’ memory access behavior, even within their “steady phase” (after initialization and before finalization).

Meanwhile, using longer episodes might lose timely adaptation with online monitoring or decision making, and may bring higher profiling/analysis overhead. To this end, “partitioned” per-variable accesses (like those shown in Figure 9(a)) appear to possess clear periodic patterns, though producing much more irregular aggregate access streams when interleaved. This suggests that variable access pattern analysis can be conducted offline, then combined with lightweight parameters and events (such as object allocation and de-allocation) to estimate a program’s overall memory access behavior. The iterative phases and long object lifetimes with scientific applications make such methodology more promising there.

6 RELATED WORK

**Memory Tracing Tools.** There are many projects building memory tracing tools [27, 35, 38, 48] and reducing their runtime performance overhead optimizations [18, 19, 21], including those using dynamic sampling rates to minimize the runtime profiling overhead [21]. Existing tracing tools often require source codes to be recompiled with tracing tools and in particular, may require the use of special heap memory allocation libraries [48]. Meanwhile, several variable and object level profiling techniques [10, 11, 45, 47, 62] have been explored to provide a finer-grained profiler, and provide a clearer picture of a program’s memory behavior. These methods mainly rely on runtime instrumentation. In contrast, our work does not require source code access and focuses on understanding per-object behavior in relation to its variable within a program.

**Memory Access Pattern Characterization.** A large body of memory profiling works in the past had objectives that were limited to understanding the bandwidth and memory footprint requirements of the whole application, instead of per-variable memory access behavior. For example, existing profiling work [27, 35, 38, 48] only reported memory access information at raw memory address level. Not only variable-level patterns are obscured, aggregate access patterns can change in complicated ways when application inputs change. Instead, our work provides an insightful and intuitive
understanding of applications’ memory accesses by associating access patterns to their corresponding objects, especially by assessing behavior changes across varied input problem sizes.

On object-level study, there have been visualization tools to highlight memory accesses, such as HPC Toolkit [2, 34]. Wu et al. [62] proposed an efficient object-relative translation and decomposition technique to retrieve access information within one object. Zorn et al. [4, 7, 49] conducted a series of studies on memory object allocation. More recently, Voskuilen et al. [56] analyzed the behavior of malloc calls in multiple HPC applications to promote application-driven allocation for multi-level memory management. The NVmalloc library [57] allows users to explicitly allocate variables on SSD devices for out-of-core scientific computing, while Hammond et al. [23] presented automated policies to assist user-directed placement. Peng et al. [44] studied applications’ memory behavior on Intel Knights Landing (KNL) and utilized applications’ access patterns to guide data placement on hybrid-memory systems. Our systematic study of object-level access patterns (across 38 applications from diverse domains) supplements these existing approaches and provides practical implications for ongoing architectural, system, and programming language work.

**Domain-specific Characterization.** Many prior studies [5, 28, 60, 61] have analyzed memory access patterns in scientific and commercial applications. E.g., Barroso et al. [5] provides a detailed performance survey of three major business application classes (such as OLTP and web search). Zhang et al. [28] studied data locality in commercial OLTP and Weinberg et al. [60] quantified temporal locality of HPC applications. In particular, one prior study [61] exploits temporal address correlation and stream locality in shared memory accesses for scientific and commercial multiprocessor workloads, respectively. However, our research is the first to systematically compare and analyze general memory access behavior between scientific and other applications.

**Locality Quantification.** Many prior studies [15, 22, 40, 60] have examined temporal and spatial locality. Reuse distance [15, 60] is a popular locality metric, but is costly to calculate and only useful for scenarios with LRU-like replacement. Our approach targets the per-object access locality (plus other patterns) naturally tied to program semantics, which potentially allows application-level reuse distances to be derived by synthesizing per-object access patterns, when supplied with proper runtime parameters.

7 CONCLUSION

In this work, we have developed a two-level profiling framework, and used it to obtain in-depth understanding of memory allocation and access behaviors, while connecting memory objects with program variables as well as computation semantics. We have profiled in detail 38 applications, spanning various domains including AI, data analytics, and HPC, each running with three problem sizes. Our comprehensive analysis of the results produced seven key observations that helped us identify critical differences in scientific applications’ memory behavior, compared to other applications, and the corresponding opportunities/challenges for memory system design or memory management optimization.

Our profiling and analysis results have verified that there are significant differences in variable/object allocation and access behaviors between HPC applications and commercial/desktop ones. These findings can facilitate the design and optimization of HPC-specialized hardware, compilers, and middleware, such as next-generation supercomputer processors or accelerators. At the same time, we need to be careful in applying memory management design choices or recommendations based on experiments using only desktop/personal computing programs. Finally, commonly used HPC benchmarks do not capture the complexity in memory object allocation/access of real-world large applications.

ACKNOWLEDGEMENT

We appreciate the thorough and constructive comments/suggestions from all our reviewers. Especially we thank our shepherd, Simon Hammond, for his guidance and responsiveness. This work was supported in part by the National Key R&D Program of China (Grant No. 2016YFA0602100), National Natural Science Foundation of China (Grant No. 91530323), Qatar Foundation (through a QCRI-MIT collaboration program), and the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MISP) (No. 2015R1C1A1A0152105). It was also supported by, and used the resources of the Oak Ridge Leadership Computing Facility at ORNL, managed by UT Battelle, LLC for the U.S. DOE under the contract No. DE-AC05-00OR22725.

REFERENCES


