MemGaze: Rapid and Effective Load-Level Memory Trace Analysis

Ozgur O. Kilic
Pacific Northwest National Laboratory
ozgur.kilic@pnnl.gov

Nathan R. Tallent
Pacific Northwest National Laboratory
tallent@pnnl.gov

Yasodha Suriyakumar[§]
Portland State University
yasodhan@pdx.edu

Chenhao Xie
Pacific Northwest National Laboratory
xiechenhao2013@gmail.com

Andrés Marquez
Pacific Northwest National Laboratory
andres.marquez@pnnl.gov

Stephane Eranian
Google Inc.
eranian@google.com

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00058

***Abstract*—A challenge of memory trace analysis is combining detailed analysis and low overhead measurement. Currently, hardware/software-based analysis of load-level sequences easily incurs time slowdowns of 100×. We present MemGaze, a tool for low-overhead, high-resolution memory trace analysis. MemGaze uses Intel's Processor Tracing (PT) instruction `ptwrite` to collect sampled and compressed memory address traces for load-level, sequence-aware analysis of data reuse. We describe multi-resolution analysis for locations vs. operations, accesses vs. spatio-temporal reuse, and reuse (distance, rate, volume) vs. access patterns. Both trace size and resolution are controllable.**

We use MemGaze to elucidate the memory effects of different data structures and algorithms. For sampled traces that are ≈1% of a full one, analysis metrics have 1-25% MAPE for histograms of varying dynamic sequence lengths. With current suboptimal kernel support (PT runs continuously), MemGaze's time overhead is typically 10–95%; 7× at worst. However, when PT runs only during samples, overhead is 10–35% on memory intensive regions and correlates with executed `ptwrite`s.

***Index Terms*—memory access tracing, processor tracing, spatio-temporal reuse, footprint, memory access patterns, MemGaze**

I. Introduction

The performance of the state-of-the-art applications on parallel machines is far below the limit set by Amdahl's law. Whether the machine is based on many-core, GPUs, FPGAs, or a heterogeneous combination, usually the most significant bottleneck is accessing data from the memory system [1]–[3]. Thus, memory analysis and optimization are critical. The major challenge of memory analysis tools is delivering detailed insight without orders-of-magnitude of additional time, space, and execution resources.

Detailed application insight requires analysis of both movement and data reuse [4]–[11]. Data movement includes access frequencies and costs. Data reuse includes temporal and spatial locality, footprint, and access patterns. Such insight is typically gathered with memory reuse and modeling tools [8], [9], [12] or memory simulators [10], [11], but requires orders-of-magnitude of additional resources (time and execution) for tracing and space for data (intermediate and final). Recent measurement techniques permit low-overhead application analysis, but address only one form of locality, usually reuse distance [13].

We present MemGaze, a new tool that provides low-overhead, load-level analysis of memory accesses and data reuse for applications. MemGaze differs from prior low-overhead tools in two ways. First, it analyzes *sampled access traces*, where each sample is a long address sequence (≈1K), collected with *emerging hardware support for processor tracing*. Such sequences are useful because they enable temporal, spatial, location, and access pattern analyses. Second, via sampled traces, it provides a broad set of memory analyses that include locations vs. operations, accesses vs. spatio-temporal reuse, and reuse (as distance, rate, and volume) vs. access patterns.

Prior low-overhead memory analyses do not analyze memory address sequences because prior collection methods incur very high overheads. Software-based sampling can capture sequences of memory addresses by switching between instrumented and non-instrumented execution, but easily incurs time overhead of 100× [12], [14]. Hardware for performance monitoring was not designed for detailed data reuse analysis. Many tools analyze cache misses or statistics from load-store queues. Costly data accesses [15], [16] can be identified by sampling loads and capturing data addresses with AMD's IBS (or LWP) [17], Intel's PEBS-DLA [18], IBM's Marked Events [19], and ARM's SPE [20]. Temporal reuse can be captured by sampling reuse intervals [13], [21], [22]. However, none of these methods simultaneously permit temporal, spatial, location, and access pattern analyses.

To enable low-overhead and high-resolution memory analysis, MemGaze uses Processor Tracing (PT) to collect sampled and compressed memory address traces. PT was originally designed to collect control flow to assist debugging and is supported by different vendors, e.g., Intel x64 [23] and ARM [24]. Both Intel's and ARM's PT also support gathering data addresses, though neither is widely available. MemGaze leverages Intel's `ptwrite` instruction, which can write an arbitrary word data packet, such as an address, to a pinned OS buffer without OS intervention. That is, *PT can be entirely enabled or disabled by hardware*. `ptwrite` is currently supported by Goldmont Plus cores (Gemini Lake product);

[§]Work performed while at Pacific Northwest National Laboratory.

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00058

it very recently appeared with Alder Lake ('desktop') and is scheduled for Sapphire Rapids ('server') [25], which will be used in Argonne National Laboratory's Aurora supercomputer.

Although `ptwrite` presents an interesting opportunity for collecting memory traces, there are several challenges. First, analysis of such traces must account for gaps due to sampling. Second, straightforward use of `ptwrite` generates huge traces — larger than PT for control flow [26] — that require frequent copying from OS memory to application memory to storage, which consumes memory bandwidth and introduces blocking delays unless throttled. For example, Linux `perf` [27] drops an unpredictable 30–50% of data, even when CPU frequency is throttled. Thus, satisfactory solutions should reduce time overhead and data rates.

We make the following contributions. *First*, we demonstrate low-overhead sequence-aware memory analysis with MemGaze.[1] Specifically, we show how to use `ptwrite` to collect sampled memory access traces, i.e., samples of memory accesses and their data addresses. The sampled traces enable instruction-level analysis of access sequences and their data address. Because both the sampling rate and sequence size are controllable, trace size is also controllable. Although our prototype is Intel-specific, the method is easily generalizable.

Second, we describe static analysis and binary instrumentation (§II and §III) that enables a compressed (non-lossy) sample representation via selective instrumentation of memory instruction. Although most data reduction comes from sampling, the compression adds another 1.2–2×. This analysis enables rapid decomposition of footprint by access patterns without expensive sequence analysis.

Third, we describe multi-resolution analysis (§IV) for locations vs. operations, accesses vs. spatio-temporal reuse, and reuse (distance, rate, volume) vs. access patterns. Because trace size is controllable, analysis times are reasonable, even with prototype implementation.

Fourth, we evaluate MemGaze on benchmarks and parallel applications with different access patterns (§VI and §VII). We elucidate the memory effects of different data structure implementations and algorithms. For a sampled trace that is ≈1% of a full one, analysis metrics have 1-25% MAPE (mean absolute percentage error) for histograms of varying dynamic sequence lengths. With current suboptimal kernel support (PT runs continuously), MemGaze's time overhead is typically 10–95%; 7× worst. However, when PT runs only during samples, overhead is 10–35% on memory intensive regions and correlates with executed `ptwrite`s, which are expensive.

II. Overview

MemGaze is based on collecting and analyzing sampled memory traces. Detailed memory analysis of applications is often not feasible using other methods either with respect to time, data volume, or resource usage.

Figure 1 shows an overview of collecting and analyzing memory traces. The first two steps implement sampled tracing.

[1]Publically available at https://github.com/pnnl/memgaze.

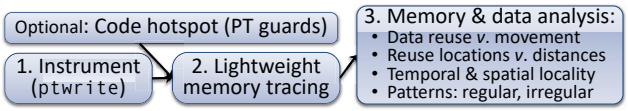


Fig. 1. Collecting and analyzing memory traces.

Binary instrumentation (Step 1) inserts Processor Tracing instructions (`ptwrite`) that record sampled sequences of memory addresses (Step 2). The instrumentation can be entirely enabled or disabled by hardware.

To help focus results, one may optionally perform standard hotspot analysis based on time or memory loads. This result defines a region of interest (set of functions) that are used to limit tracing. The region of interest limits tracing using one of two methods: selective instrumentation (Step 1) or Processor Tracing's hardware guards (Step 2). With PT's hardware guards, the region of interest can change without re-instrumentation.

Figure 3 shows a sampled trace fragment. A sample is a sequence of w recorded accesses followed by z non-recorded accesses. Since $(w + z) \gg w$, e.g., $(w + z)/w \approx 10^{3\cdots5}$, our traces are a fraction of a full memory trace. The result is address- and sequence-aware traces for very low time and space overhead.

To diagnose data reuse and memory movement problems, we provide analyses to characterize data movement, temporal and spatial reuse, locations and footprints, and access patterns. The analyses fall into four categories. Analysis of data *access frequency vs. reuse* compares memory hotspots (movement) with poor locality and accesses, potential causes of unnecessary movement. Second, reuse analysis can be performed on memory *locations* or memory *operations*. The former highlights a specific memory region or data object; the latter highlights a specific sequence of memory operations. Third, we characterize both temporal and spatial locality to highlight good use of cache blocks and caches. Finally, we characterize access patterns, both regular and irregular, to distinguish between accesses that are expected to have good and poor performance. For example, the former (regular) can hide data movement with prefetching; the latter (irregular) cannot.

III. High-resolution Memory Traces

Very large traces affect (a) application execution (frequent memory copies, saturated memory bandwidth, blocking events), (b) trace integrity (unpredictable throttling and data drops), (c) downstream analysis time and memory, and (d) trace storage. This section describes how to collect sampled, high-resolution memory traces using `ptwrite`. It also describes static analysis for selective instrumentation and compressed traces.

A sampled trace is shown in Fig. 3. To collect such traces, `ptwrite` must be inserted into the instruction execution stream, which we do using static binary instrumentation.

A. Instrumenting loads

Our instrumentor, which leverages DynInst [28], takes as input an executable's important *load modules*, i.e., an executable

and relevant libraries. It outputs a new executable and an auxiliary annotation file, whose contents are described below. The off-line approach ensures that the static code analysis we perform for trace compression and access pattern decomposition has no time or space overhead for executions. A benefit of binary instrumentation is that it can instrument libraries such as the C++ standard library, which can be a significant source of complex memory behavior.

For each load module, the instrumentor analyzes memory accesses within each procedure separately. On x64 there are several addressing modes. The two main categories are:

`ptwrite` r_s ; load $r_d \leftarrow [r_s] + o$

`ptwrite` r_{s1} ; `ptwrite` r_{s2} ; load $r_d \leftarrow [r_{s1}] + k[r_{s2}] + o$

where r_s indicates source registers, brackets indicate dereference, and k and o are literals for scale and offset, respectively. `ptwrites` are only inserted for source registers, i.e., dynamic info. The literals are extracted, keyed by instruction address, and placed in the auxiliary annotation file. `ptwrites` should precede loads, because the source address can be overwritten when $r_d = r_s$.

The effect of this strategy is that *all instrumentation can be masked by hardware*. The instrumentation is a single instruction *without side effects on the CPU's architectural state.* Consequently, it does not require a function call or register saves/restores. Further, the instrumentation can be enabled or disabled by hardware.

In some situations it would be possible to pack multiple `ptwrite` payloads into a single 64 bit payload. Examples are two 32-bit registers or two registers whose contents are known to have a common 32-bit prefix. However, packing significantly complicates instrumentation by requiring an additional register, triggering either a trampoline call or register reallocation.

B. Compression via load classes

For load-based analysis we can ignore stores. Rather than instrumenting every load, we compress traces based on a load's expected access pattern. Programs often reuse constant pools of data that are uninteresting from the perspective of dynamic footprint analysis. Our analysis views as uninteresting both scalars on the execution stack and scalar global data. Thus, if a load is *Constant*, there is no need to instrument its register as long as we know it executed.

Load access patterns are also helpful during analysis because they highlight differences in expected access latencies (cf. §II). To analyze access patterns, the instrumentor analyzes data dependencies for each procedure's object code. From data dependencies, the instrumentor classifies each load to distinguish three load classes.

- *Constant* loads access scalar data either within a stack frame or to global data. Specifically, this means scalar loads (offset of 0) that are relative to a frame pointer or to a global section. All constant loads are viewed as accessing the *same* address, using total space of 1 unit.
- *Strided* loads are relative to a loop induction variable (loop-carried dependency) with constant stride.
- *Irregular*. All other loads are classified as irregular. Typically, they are indirect loads through pointers.

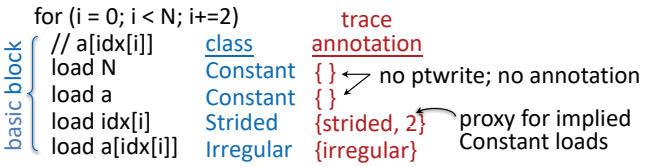


Fig. 2. Trace compression using load access classes.

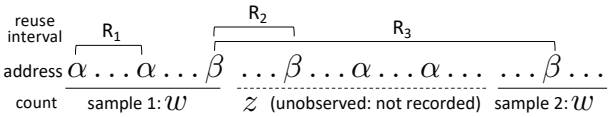


Fig. 3. Sampled memory address trace with reuse interval categories R_1–R_3.

Figure 2 shows an example of load classification and trace compression. For Strided and Irregular loads, memory addresses are always instrumented and the load class (blue) added to the auxiliary annotations (red). For constant loads, it is sufficient to know its *basic block* executed. Basic blocks divide code into straight-line sequences such that an instruction is executed if and only if any other is executed. Thus, the instrumentor selects a proxy instruction within the basic block. If the block contains a Strided or Irregular load, one is selected and annotated with the number of implied Constant loads. Otherwise, the instrumentor selects the first Constant load as the proxy and instruments it. As a result, in Fig. 2, only half the loads are instrumented. Our results (§VI-C) show that with this scheme, compression of non-optimized and optimized code is about 2× and 1.2×, respectively. The difference makes sense because of the higher rate of frame loads without optimization.

One can also imagine a compressed representation of strided loads using a tuple of ⟨begin-address, stride, end-address⟩. We choose to forgo this because it results in complex instrumentation (e.g., conditionals, register spilling).

C. Sampled memory traces

We collect traces using an extended Linux `perf` [27]. Figure 3 shows two samples from a resulting trace. A sample is a sequence of w recorded accesses followed by z non-recorded accesses. Each recorded access is associated with an instruction pointer, memory address, and timestamp. Each sample shows memory accesses (loads) in temporal context. The samples are uniform in memory *accesses* because any set of accesses is equally likely to appear. The samples' accesses represent common reuse and access patterns.

With Processor Tracing, the sample window w corresponds to the contents of a fixed-size circular buffer. The average sampling period $w + z$ specifies that a sampling trigger is generated to read the buffer.[2] Usually, $(w + z) \gg w$, e.g., ratios of $10^{3\ldots5} : 1$. Table I tracks important symbols.

Let σ be a set of samples with a total of $|\sigma|$ samples. Averaging across σ, each sample has w loads. Let $A(\sigma)$ be

[2]To ensure a uniform sample of memory addresses, the sample trigger should be a hardware counter for memory accesses, e.g., loads. Sampling in time will decrease accuracy if the load rate changes over time.

TABLE I
IMPORTANT SYMBOLS.

σ, $\lvert\sigma\rvert$	Sample of access sequences; number of samples
$\hat{}$	Population estimate from sample (vs. observation)
$w+z$	Sample period (memory accesses)
$\mathcal{A}$	All (uncompressed) accesses
A	Observed (possibly compressed) accesses
ρ	Sample ratio. All executed : all sampled accesses
κ	Compression ratio. All : selected accesses
W	Effective trace window or interval size
C	Captures, addresses with reuse
S	Survivals, addresses without reuse
D	Spatio-temporal block reuse distance
F	Footprint
F_{str}, F_{irr}	Footprint with {strided, irregular} access pattern
$F_{\mathsf{str}\%}$, $F_{\mathsf{irr}\%}$	Fraction of {strided, irregular} footprint
$A_{\mathsf{const}\%}$	Fraction of accesses to constant-sized data
ΔF	Footprint growth rate; footprint per access
$\Delta F_{\mathsf{str}\%}$, $\Delta F_{\mathsf{irr}\%}$	Fraction of strided (irregular) footprint growth

the number of observed memory accesses in σ and $\hat{\mathcal{A}}(\sigma)$ the (estimated) value for all (uncompressed) accesses. Then, if σ is a single sample, $w = A(\sigma)$ and $w + z = \hat{\mathcal{A}}(\sigma)$.

Decompressing samples: With compression, observed accesses A can differ from accesses $\mathcal{A}$ directly implied by the observation. To reason about compression, we introduce the sample ratio ρ of *all executed* to *all sampled* memory accesses, or $\rho = \frac{\hat{\mathcal{A}}(\sigma)}{\mathcal{A}(\sigma)}$. We also introduce the compression ratio $\kappa(\sigma)$ of all to selected (compressed) accesses in σ. (Cf. Table I.)

When monitoring all memory accesses (*non-selective* instrumentation), ρ is simply the ratio of all to observed accesses or $\frac{w+z}{w}$. With selective instrumentation, we account for the constant loads directly implied by the sample:

$$\rho = \frac{\hat{\mathcal{A}}(\sigma)}{\mathcal{A}(\sigma)} \overset{\textit{non-selective}}{=} \frac{|\sigma|(w+z)}{A(\sigma)} \overset{\textit{selective}}{=} \frac{|\sigma|(\kappa(w)+z)}{\kappa(\sigma)A(\sigma)} \tag{1}$$

We calculate $\kappa(\sigma)$ using the relation $\kappa(\sigma)A(\sigma) = A(\sigma) + A_{\mathsf{const}}(\sigma)$, which yields

$$\kappa(\sigma) = 1 + \frac{A_{\mathsf{const}}(\sigma)}{A(\sigma)} \tag{2}$$

It is easy to calculate $A_{\mathsf{const}}(\sigma)$ from the combination of the trace and auxiliary annotations generated during binary instrumentation. $\kappa(w)$ is analogous.

D. Enabling source code attribution

A challenge when using binary instrumentation is attributing memory analysis results to source code. The reason is that the new (instrumented) instruction stream is no longer aligned with the load module's source-line mapping. To recover the source code mapping, we extended the functionality in DynInst with an interface that reads the newly recorded mapping between the new object code and source code.

IV. ANALYZING SAMPLED TRACES

This section describes analysis for traces of sampled access sequences. Compared to low-overhead methods that sample address regions or reuse instances, these traces enable temporal, spatial, location, and access pattern analyses. However, there are limitations that must be understood. We describe these limitations and then summarize our analyses.

A. Sampling limitations

An obvious limitation of sampled traces is that they may miss very short or infrequent behaviors. More subtly, a uniform sample of memory accesses may *not* have a uniform sample of reuse intervals.

A *reuse interval* is the number of loads (or instructions) between a pair of references to the same address (cf. [29]). (In contrast, *reuse distance*, or stack distance [30], is the number of unique addresses in the interval.) It turns out that some reuse intervals may not be captured, so that the samples do not represent a uniform sample of reuse intervals.

To see this, we classify the ability to observe reuse intervals into three categories (Fig. 3). We say a reuse interval is *captured* if both accesses appear within sampled data.

- (R_1) Within a sample (w), *some* reuse intervals $2 \ldots w-1$ can be captured. Some cannot because one element of the interval could occur at the end of a sample buffer.
- (R_2) Within a sample period $w+z$, it is *not* possible to capture reuse intervals $w, \ldots, z$. Similarly, it is impossible to capture intervals $z+1, \ldots, w+z-1$.
- (R_3) Between samples we have a generalization of the prior categories. It is impossible to capture intervals such that $d \bmod w = 0, \ldots, z$. Further, although it may be possible to capture some intervals greater than $z+1$, it is impossible to distinguish a single complete interval from multiple incomplete instances to the same address.

B. Reducing error with sample aggregation

Although some reuse intervals may not be observed, with sufficient samples, effective reuse analysis is very likely. Most of our analyses aggregate all samples across a certain dimension, such as a function instance, reuse distance range, or address region. Sample aggregation is important not only because it highlights diagnostic trends, but also because the accumulation of more samples reduces blind spots and statistical error. Recall that trace windows have blind spots for (R_2) and potentially sparse coverage for (R_3).

When trace windows are aggregated to program functions over many samples, forming *code windows*, we expect some observability of blind spots (region z in Fig. 3) and therefore useful estimates for (R_2) and (R_3). The reason is that estimates for *captures* (C) and *survivals* (S) — addresses with and without reuse, respectively — significantly improve.

With sufficient samples, we can use standard estimators to scale statistics from samples and apply them to the population. The typical case is scaling metrics by the ratio of executed to sampled memory accesses, or ρ; cf. Eq. (3). (It may be necessary to account for trace compression, κ.) For hotspots, we expect these diagnostics to be highly informative.

If error is a concern, recall that *there is often no practical non-sampling methodology* for detailed memory analysis. Even

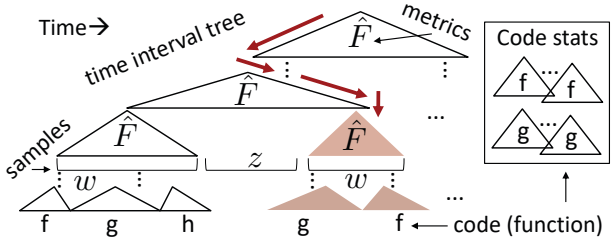


Fig. 4. Multi-resolution time analysis finds time regions with poor locality.

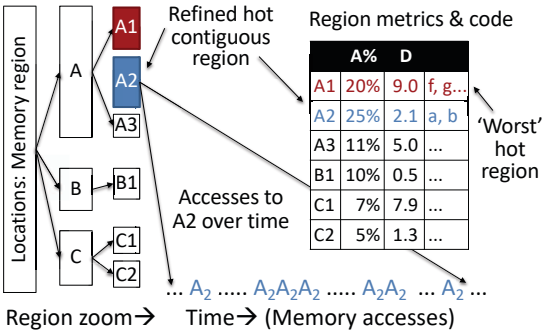


Fig. 5. Location zooming finds hot memory regions with poor locality.

with PT hardware assistance, full traces are not feasible without random drops. Rather, the choice is between coping with microbenchmarks and small data sets, devoting enormous time and machine resources to simulation, or ad hoc methods.

C. Multi-resolution time & location analysis

To quickly find interesting time intervals and memory regions, we use a multi-resolution analysis that recursively adjusts the granularity of execution time and access location using tree structures.

1) Execution time: To find time intervals of operations with poor reuse, we analyze *accesses* and *static code* (e.g., functions) over execution time. Figure 4 shows an *execution interval tree* representing sets of samples at varying window sizes (time intervals). The execution interval tree is built bottom-up, beginning with samples. Tree nodes above samples correspond to increasing inter-sample intervals. Nodes below samples correspond to intra-sample intervals. The leaf function nodes group access sequences from the same function. Metrics are associated with each node. Inter-sample metrics are estimates, whereas intra-sample metrics are exact. The red sequence of arrows descending from the root "zooms" to a hot interval (many accesses) with poor reuse (e.g., large footprint growth).

2) Location regions: To find *memory regions* with poor spatio-temporal locality, we use location-based zooming. Figure 5 shows this analysis. The zoom tree proceeds top-down from a single memory region to its hot sub regions (left-to-right in the figure). The tree's leaves show the final regions, e.g., A2. For each final region, we show hotness (% total accesses), spatial-temporal reuse distance D for accesses to that region, and the code (function, line) for those accesses. The figure's table shows this data. Accesses for region A2 are shown over time (blue), which correspond to functions a and b, account for 25% of the total. However, the 'worst' hot region is A1, which accounts for 20% of total accesses but has a far worse reuse distance (D).

The recursive zoom procedure proceeds as follows. Given a region, it is divided into fixed-sized pages and access blocks. An access block b_a, represents the unit access size for spatio-temporal reuse distance D; we default to the cache line size. The page size b_p is used to identify subregions and recursively reduces with tree level. A hot subregion is a maximal set of contiguous pages, each with at least 1 access, where the set's total access is at least $t\%$ the region's accesses. The zoom stops when a subregion reaches a minimum threshold.

The *contiguous* property of a hot region is important. Although some parts of a hot region may be cold, including them tends to capture a single object or collections of related objects. In this way, reuse distance (D) represents the spatio-temporal locality of the *entire* object. In contrast, only focusing on a region's hot blocks filters all other accesses to the region, frequently making spatio-temporal locality appear very good.

The stopping threshold is also important: when too large it can capture semantically different objects and average many behaviors. When too small, the analysis is resource intensive and potentially noisy.

V. Memory and Data Reuse Analysis

We develop several methods for characterizing reuse within sampled traces. Together they provide a broad set of diagnostics that capture locations vs. operations, accesses vs. spatio-temporal reuse, and reuse (as distance, rate, and volume) vs. access patterns.

A. Data movement and access frequency

Characterizing data movement between different system components is a complex process. In this paper, we view memory as a single system. To find hot memory regions, we calculate access frequencies for each region, focusing on accesses (A) classified as non-Constant. These accesses also represent data that must be moved by the memory system.

B. Spatio-temporal reuse distance and interval

Memory performance is related to the temporal and spatial reuse of cache lines. We therefore capture spatio-temporal locality using reuse distance and reuse interval with respect to a configurable access block size.

Reuse distance D (or stack distance) [4], [5] is defined as follows. With execution time analysis (§IV-C1), $D(w)$ it is the unique memory blocks between two operations or a window of accesses w. With location region analysis (§IV-C2), $D(b)$ for a block b is the unique memory blocks between two subsequent accesses to b. The *reuse interval* for that same block measures number of accesses; it is easier to calculate but only an estimate of unique blocks.

To adapt D to sampled traces, we either focus solely on intra-sample windows or calculate the average unique blocks

accessed between samples based on footprint growth. For cache-friendly data structures, we focus on intra-sample reuse where blocks are cache lines. For working-set analysis, we use inter-sample reuse and blocks of OS page size.

C. Data volume: Footprint

In time analysis (§IV-C1), it is important to understand an access sequence's data footprint and new data per access. Footprint is the amount of *unique* data accessed by a series of operations. The observed footprint $F(w)$ for a single sample of size w is the unique addresses in w. The estimated footprint $\hat{F}(w+z)$ for the sampled and unsampled addresses assumes $F(w)$ is representative of $w+z$, i.e., that the ratios of unique addresses between the sample and total population are similar. With sufficient samples within a uniform sample of loads, this assumption of proportionality holds. Therefore, $\hat{F}$ should scale the sample footprint F by the ratio ρ of expected and observed footprint. Thus, over a set of many samples σ, the estimated footprint $\hat{F}(\sigma)$ for window size $W(\sigma)$

$$\hat{F}(\sigma) = \begin{cases} F(\sigma) = C(\sigma) + S(\sigma) & \text{intra-window} \\ \rho F(\sigma) = \rho\,(C(\sigma) + S(\sigma)) & \text{inter-window} \end{cases} \quad (3)$$

Statistically, intra-sample intervals (R_1) can be interpreted in two ways. Besides the intra-window portion of Eq. (3), they can also be viewed as a sample end point (e.g., smaller average w) and scaled using the inter-window portion.

D. Data reuse rates: Footprint growth

Footprint growth is footprint's rate of change. Let $A(\sigma)$ be number of addresses in a sample. Then, average footprint growth $\Delta\hat{F}(\sigma)$ over the sample of window size $W(\sigma)$ is

$$\Delta\hat{F}(\sigma) = \frac{\hat{F}(\sigma)}{W(\sigma)} = \frac{F(\sigma)}{\kappa(\sigma)A(\sigma)} \quad (4)$$

An alternative way to view footprint growth is as normalized footprint, i.e., average footprint per load. Note that the *final equation form does not depend on window classes*.

E. Access Patterns

Many applications tend to frequently alternate between regular execution phases with structured memory access patterns and irregular phases with unpredictable memory behaviors.

a) Footprint access diagnostics: To highlight large differences in expected access latencies, we decompose footprint into strided (prefetchable) and irregular (non-prefetchable) access components. The footprint categories are called footprint access diagnostics and represent the most common patterns that affect memory performance. Specifically, we define the following metrics: strided and irregular portion of footprint (F_{str}, F_{irr}), their growth rate (ΔF_{str}, ΔF_{irr}), fraction of footprint growth due to them ($\Delta F_{\mathsf{str}\%}$, $\Delta F_{\mathsf{irr}\%}$), and fraction of constant loads ($A_{\mathsf{const}\%}$). This analysis is *constant time per operation*, without any pattern analysis, using the load classes of §III-B.

b) General irregularity: For a more general measure of irregularity, we use spatio-temporal reuse distance (above).

VI. Evaluation

This section evaluates metric accuracy, time overhead, and space reduction. The next section (§VII) provides case studies.

Platform: To evaluate MemGaze we use an Intel Pentium Silver J5005 (Gemini Lake) CPU with four cores and 16GB memory. (We are procuring an Alder Lake machine, but it is not yet available.)

Benchmarks: We use a set of microbenchmarks and application benchmarks. The microbenchmarks simulate accesses to both dense and sparse data structures and vary access patterns, data reuse, access sparsity, and access likelihood. Microbenchmarks test analysis of (very) short-lived access sequences that become hotspots (repeated 100 times). The microbenchmark names give access patterns using "str" (strided with stride step) and "irr" (irregular). The different accesss patterns can be composed conditionally ('/') or in series ('|'). For applications we use the graph benchmarks miniVite [31] (Louvain Community Detection) and GAP [32] (Connected Components and Page Rank); and the machine learning Darknet [33]. The graph benchmarks test analysis of reuse and access patterns within load-intensive applications that include highly irregular accesses. We vary compiler optimization levels (O0 vs. O3). For graph benchmarks, we also vary data structure implementations and algorithms. For Darknet, we vary network types for inferencing on images.

All application benchmarks support OpenMP and are executed with and without parallelism. However, note that our analyis focuses on memory behavior and is *orthogonal* to *CPU* parallelism. (Future work includes exploring the relationship between CPU concurrency and memory systems.)

Sampling configuration: For microbenchmarks, we use a small period (10K loads) and a large buffer (16 KiB yielding ≈1150 addresses) to capture short-lived phenomena. The miniVite and GAP benchmarks use a larger (typical) period (10M and 5M loads, respectively) and smaller buffer (8 KiB yielding ≈500 addresses). The reason buffers do not yield the expected addresses (size / 8 bytes) is due to the suboptimal kernel support (buffer fill and flushes occur asynchronously with the sampling trigger that record the next buffer).

A. Validation of data reuse analysis

This section validates data reuse analysis of sampled traces. Figure 6 shows results for each microbenchmark and graph benchmark. The data series represent footprint access diagnostics: footprint (F), irregular footprint (F_{irr}) and strided footprint (F_{str}). We exclude reuse distance (D) as we prefer intra-sample calculations. The three series for MAPE show mean absolute percentage error over different *trace windows*, i.e., metric histograms with power-of-2 windows. The three series for percentage error represent *code windows*, i.e., aggregated samples for functions, which reduce error (§IV-B).

For microbenchmarks, we validate against *full* traces. For graph benchmarks, we validate against sampling data that is 10× more frequent. Collecting full traces for the graph benchmarks is problematic for three reasons. First, using PT is not feasible: the data copy rate between PT's pinned kernel buffer and user

memory is too high for real-time, resulting in random drops of 30–50%. (CPU frequency throttling makes little difference.) Second, when we attempted to collect full trace data with our validation tool, we found it would take days per benchmark. Finally, full traces are huge (order 1 TB), which requires substantial resources to process per benchmark.

For *trace* windows (first three series), MAPE is <25%. Errors tend to be higher for benchmarks that are more irregular or that include more short-lived behavior, which is expected. For *code* windows (second three series), error <5%. In general, we expect code windows to be more accurate than trace windows because they accumulate more samples.

The errors within trace windows can be understood as follows. First, as expected, errors are *quantitative* overestimates (§V-D) rather than *qualitative*. With additional space, we could show each pair of overlapping histogram curves. Second, from our analysis of reuse intervals, we know traces have some blind spots (§IV-A). Third, with benchmarks that include irregular and short-lived behavior, we expect that characterizing trace windows from samples will include error within the reuse analysis (captures C and survivals S).

It should be possible to automatically detect most under-sampling by analyzing sample density and forming confidence intervals. One could flag regions with insufficient samples.

B. Time overhead

Recall that MemGaze's toolchain consists of (1) binary instrumentation, (2) memory tracing, and (3) analysis (Fig. 1). We evaluate each step. Fig. 7 focuses on memory tracing while Table II shows sample times for the first and third.

1) Memory Tracing: Our primary interest is memory tracing because low overhead implies (a) fewer timing disturbances (which enables analysis of task interleaving and memory parallelism) and (b) smaller trace sizes, which improves analysis time. Fig. 7 shows MemGaze's run time overhead. We show results for two versions of MemGaze. MemGaze is the full implementation that relies on suboptimal kernel support where PT runs continuously. MemGaze-opt is proof-of-concept (in user-space) that enables Processor Tracing only during samples. The figure also breaks down overhead by application phase. The miniVite and GAP benchmarks include a graph generation phase that has distinctly different memory accesses than the second phases ('modularity' and 'rank').

TABLE II
TIME OVERHEAD: BINARY INSTRUMENTATION & ANALYSIS.

Benchmark	Binary Size	Instrument	Analysis/1	Analysis/2
μbenchmarks	19 kB	1.1s	76.1s	79.2s
miniVite-O3-v1	1900 kB	135.5s	357.0s	284.5s
GAP pr-O3	95 kB	144.2s	37.4s	22.9s
GAP cc-O3	100 kB	122.1s	32.3s	21.5s
Darknet-AlexNet	2700 kB	83.2s	87.4s	21.1s
Darknet-ResNet	2700 kB	83.2s	898.2s	289.5s

The first three series of both figures show per-phase and total overhead for MemGaze. Even with sub-optimal driver support, MemGaze's overhead is typically 10–95%. Overhead is higher with more compiler optimization (O3) because (a) the rate of instrumented loads is higher and (b) the total run time is smaller (making it harder to amortize overhead). We hypothesize DarkNet's overhead of 5×–7× is due to `ptwrite` interfering with its much higher store rate. The fourth (red) series illustrate (a) using the ratio of `ptwrite`s to non-`ptwrite` instructions. The ratio highly correlates with total overhead.

The MemGaze-opt series (fifth, purple) reduces overhead to the near-minimum for our scheme by enabling PT only when sampling miniVite/modularity, a load intensive hotspot. MemGaze-opt reduces overhead from 80% to less than 40%, which is very close to the execution rate of `ptwrite` instructions. This makes sense because `ptwrite`s are expensive to decode and trigger data copies [26]. It may be possible to further reduce overhead with 32-bit packets and additional compression that reduces `ptwrite`s for Strided loads.

In contrast, current methods for tracing sequences of memory addresses easily incur time overhead of 100× [12], [14].

2) Binary Instrumentation and Analysis: Table II shows sample run times for MemGaze's instrumentation ('Instrument') and analysis ('Analysis/1' + 'Analysis/2') steps. These times are 'extreme worst cases' in that neither step has been optimized or parallelized. Even so, we can easily analyze memory intensive applications that are unfeasible to analyze using full traces.

As MemGaze's binary instrumentor takes a binary to analyze and instrument, its run time depends on the size and complexity of the binary. The different run times between microbenchmarks and miniVite show the effect of miniVite's much larger binary. The different run times between miniVite and PR show the effect of PR's increased routine complexity.

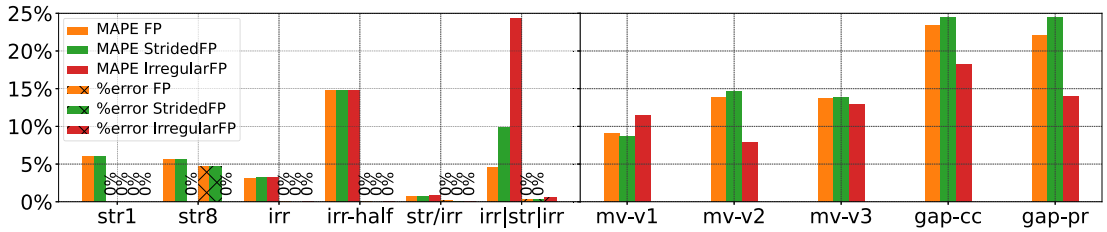
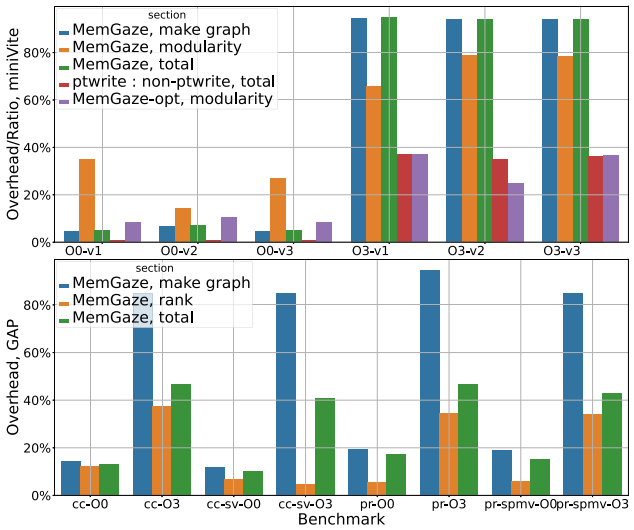


Fig. 6. Validating sampled footprint access diagnostics for microbenchmarks using mean absolute percentage error (MAPE) and comparing histograms (average window size vs. footprint metric) between sampled and 'full' traces, where 'full' is defined in §VI-A.



DarkNet	MemGaze	MemGaze-opt
AlexNet/ResNet	5× / 7×	10% / 2%

Fig. 7. Time overhead for memory tracing. Top shows miniVite; bottom, GAP.

Analysis time is divided into two sub-steps: trace building (Analysis/1) and trace analysis (Analysis/2). The building step is from converting Linux `perf`'s trace into one for our trace analysis. The run times of both depend on trace size (which is shown in Table III).

C. Space reduction

This section evaluates the space savings of sampled memory traces. Table III compares MemGaze's traces against the sizes of full memory access traces ('Full'). The table shows three versions of a 'Full' trace.

- 'Rec' shows a *compressed* full trace *that suffers data loss* because of unpredictable throttling and data drop, due to inability to copy between PT's pinned kernel buffer and user memory (§III).
- 'All' shows a *compressed* full trace *without data drops* by correcting based on `perf`'s 'DROP' records.
- 'All$^+$' show the full size *without trace compression*, i.e., by including the suppressed Constant loads.

The 'MemGaze' column shows size of a sampled, compressed trace. The 'Ratio' column shows the ratio *as a percentage* between MemGaze and the corresponding 'Full' size.

A glance at the 'All' or 'All$^+$' data shows that a straightforward use of `ptwrite` would generates huge traces — if it were possible to do without significant drops. Note that, except for the microbenchmarks, we did not use the 'Rec' traces for validation (§VI-A) because drops are most likely to occur in the most load-intensive regions, which is exactly where we want to validate. (We captured a full trace by inserting OS sleeps after each load to substantially reduce the load rate.)

Comparing the 'All' and 'All+' shows that our compression method gives an average of 1.2× and 2× space savings for compiler optimization levels O3 and O0, respectively.

TABLE III
SPACE SAVINGS OF MEMGAZE'S MEMORY TRACES.

Benchmark	Full (GB)			MemGaze	Ratio (%)		
	Rec	All	All$^+$	(MB)	Rec	All	All$^+$
all μbench-O0 (1×)	1.9	1.9	3.5	63	3.3	3.3	1.8
all μbench-O3 (1×)	1.9	1.9	1.91	20	1.1	1.1	1
all μbench-O3	112	112	113	865	0.8	0.8	0.7
miniVite-O0-v1	77	163	316.5	1620	2.1	0.9	0.5
miniVite-O0-v2	71	198	387.9	1697	2.4	0.9	0.4
miniVite-O0-v3	79	150	292.7	1660	2.1	1.1	0.6
miniVite-O3-v1	19	41	41.1	310	1.6	0.8	0.7
miniVite-O3-v2	22	43	54.9	310	1.4	0.7	0.6
miniVite-O3-v3	13	23	29.4	341	2.6	1.5	1.1
GAP-cc-O0	2.3	3.4	6.6	355	15.4	10.4	5.3
GAP-cc-O3	4.9	7.9	9.5	31	0.6	0.4	0.3
GAP-cc-sv-O0	4.4	6.4	12.5	377	8.6	5.9	3
GAP-cc-sv-O3	6.7	10.8	13	35	0.5	0.3	0.3
GAP-pr-O0	5.1	7.5	14.6	377	7.4	5.0	2.5
GAP-pr-O3	5.4	7.9	9.5	35	0.7	0.4	0.4
GAP-pr-spmv-O0	6.3	8.9	17.4	385	6.1	4.3	2.2
GAP-pr-spmv-O3	6.5	10.1	12.1	36	0.6	0.4	0.3
Darknet-AlexNet	4.6	11.2	16.9	71	1.6	0.6	0.4
Darknet-ResNet	29	59	66	748	2.6	1.3	1.2

Clearly, the major benefit of MemGaze's trace size comes from sampling. MemGaze's trace size is generally around 1% of the full trace for applications with O3 optimization. The size is controllable by changing the sample buffer size and the sampling period.

In general, 'All' traces are proportional to the number of non-Constant loads in the execution, where loads with two source registers take twice as much space (§III-A). In contrast, MemGaze's traces are proportional to the product of the number of samples, $|\sigma|$, and the sample buffer size, where $|\sigma|$ is the total executed loads divided by sample period $w + z$. Finally, trace sizes are independent of the analysis methodology adopted.

VII. CASE STUDIES

This section provides case studies for miniVite [31] (Louvain Community Detection), Darknet [33], and GAP [32] (Connected Components and Page Rank).

Effective analysis requires an understanding of access frequency (e.g., accesses are costly), spatio-temporal reuse (e.g., for cache performance) and access patterns (e.g., strided accesses leverage prefetching). Our analyses therefore combine time-centric and location-centric results.

A. miniVite

miniVite is a benchmark [31] for Louvain Community Detection. We use three variants that show effects of different hash table implementations (*map* object) on a hotspot that inspects (`buildMap`) neighboring communities for each vertex. v1 uses a C++ `unordered_map`. As an open hash table — an array of keys, each containing a linked list for items — it creates irregular accesses. v2 and v3 use TSL `hopscotch` [34], [35], a closed hash table that replaces many irregular accesses with strided ones. v2 uses the default table size. v3 right-sizes each table instance — there are many — to avoid dynamic resizing.

TABLE IV
MINIVITE/-O3: DATA LOCALITY OF HOT FUNCTION ACCESSES.

Function	Variant	F	ΔF	$F_{\text{str}\%}$	$\mathcal{A}$
`buildMap`	v1	2.3G	0.156	66.4	291K
(make *map*)	v2	2.1G	0.151	66.9	273K
	v3	2.1G	0.160	66.8	270K
map.insert	v1	>0.7G	**0.011**	**73.3**	**106K**
	v2	2.4G	**0.003**	**93.7**	**318K**
	v3	0.5G	**0.009**	**92.8**	**67.8K**
`getMax`	v1	0.4G	**0.150**	**0.5**	**44.7K**
(use *map*)	v2	1.3G	**0.040**	**98.4**	**182K**
	v3	1.5G	**0.040**	**97.8**	**194K**

Run times	
v1	8.60 s
v2	5.15 s
v3	3.88 s

TABLE V
MINIVITE/-O3: SPATIO-TEMPORAL REUSE OF HOT MEMORY (64 B BLOCK).

Object	Variant	**Reuse** (D)	# blocks	A	A **/ block**
map (hash table)	v1	**2.65**	768	55K	**71.9**
	v2	**2.79**	768	119K	**155.2**
	v3	**1.97**	768	85K	**111.3**
remote edges of local vertices	v1	**8.71**	4864	24K	**4.9**
	v2	**4.90**	4864	19K	**3.9**
	v3	**3.32**	4864	19K	**3.9**
other objs in `buildMap` (from caller)	v1	0.37	104K	19235	0.2
	v2	0.15	101K	21362	0.2
	v3	0.24	110K	22306	0.2

Time and location analyses for O3 variants are shown in Table IV and Table V, respectively. The hotspot analysis clearly highlights `buildMap` and the *map*'s logical insert function. It also highlights `getMax`, which uses *map*. Runtime for each variant is shown to the right and indicate increasing improvement between v1, v2, and v3, showing that the different hash tables configurations have an effect.

The location analysis highlights three hot regions used within `buildMap`: the *map* object; a vector of remote edges for local graph vertices; and other objects in `buildMap` coming from the caller. The region's size corresponds to allocation sizes; and accesses/block corresponds to block hotness.

We discuss results by variant. Although v1 has the fewest accesses ($\mathcal{A}$), it has poor cache behavior due to irregular accesses. This is shown in Table IV with the highest footprint growth (ΔF) and highest percentage of irregular accesses (lowest $\Delta F_{\text{str}\%}$). Table V shows that spatial-temporal reuse distance (D) is highest or high.

We created v2 to address the poor cache behavior. v2's closed hash table results in repeated (strided) traversals, indicated by an increase in $\Delta F_{\text{str}\%}$, that leverage hardware prefetchers to hide more memory latency. The access pattern also improves (lowers) footprint growth (ΔF) and tends to improve (lower) average reuse distance (D).

The trade-off for the better performance and contiguous accesses is that v2 and v3's closed design uses more memory. In contrast v1's open hash table grows only as large as needed as suggested by footprint (F).

A defect with v2 is that it significantly increases accesses: cf. $\mathcal{A}$ for *map.insert* and A for map. v3's right-sized closed hash tables avoid extra accesses from resizing (copies) and searches (over-allocation). The result reduces accesses compared to v2 but retains the benefits of strided accesses. Interestingly, because accesses decrease so much, ΔF for *map.insert* increases. v3's smaller sizes and contiguous accesses generally improve reuse distance (D).

Our analysis uncovers a common tendency: *Sparse structures have smaller footprint but more irregular access patterns, whereas dense structures have larger footprints but more regular access patterns.* Further, *several locality metrics are important for capturing a complete picture*.

TABLE VI
DARKNET: DATA LOCALITY OF HOT FUNCTION ACCESSES.

Function	Model	F	ΔF	$F_{\text{str}\%}$	$\mathcal{A}$
`gemm`	AlexNet	69M	**0.113**	100	1.2M
	ResNet152	3855M	**0.478**	100	2.6M
`im2col`	AlexNet	3.4M	0.138	100	0.05M
	ResNet152	244M	0.813	100	0.09M

TABLE VII
DARKNET: SPATIO-TEMPORAL REUSE OF HOT MEMORY (64 B BLOCK).

Object	Model	Reuse (D)	# blocks	A	A / block
`gemm`'s $\mathbf{A}$,$\mathbf{B}$,$\mathbf{C}$	AlexNet	0.76	66048	977K	14.8
`gemm`'s $\mathbf{B}$	ResNet152	0.01	38400	598K	15.6
hot region in `im2_col`	AlexNet	1.87	8192	167K	20.4
	ResNet152	2.54	3328	7K	1.9

B. Darknet AlexNet & ResNet

Darknet [33] is an open neural network framework written in C/OpenMP. We analyze image classification (inferencing) with two pre-trained models for AlexNet [36] and ResNet152 [37] for a single image.

Time and location analysis for the hottest kernels (`gemm` and `im2col`) are shown in Table VI and Table VII respectively. To capture differences between the neural network layers, Table VIII compares `gemm` across memory access intervals.

Table VI characterizes the primary hotspot, `gemm`, matrix multiplication specialized for neural networks. It dominates total footprint ($> 90\%$) and nearly all accesses are strided, as expected. The location analysis (Table VII) independently highlights `gemm` matrices as the primary hotspot. `gemm` takes matrices $\mathbf{A}$ ($M \times K$) and $\mathbf{B}$ ($K \times N$) and produces $\mathbf{C}$ ($M \times N$). Due to memory allocator decisions, all of AlexNet's `gemm` matrices are in a single region whereas ResNet152's hot region contains only $\mathbf{B}$.

Interestingly, we can compare `gemm`'s behavior between AlexNet and ResNet152. The differences in footprint (F) and footprint growth (ΔF) between AlexNet and ResNet152 correspond to differences in number and type of layers, which result in different combinations of matrix multiplication input sizes. The differences in reuse distance (Table VII) correspond to the different matrices in the hot regions. Due to the loop order in `gemm`, there is long-term reuse of $\mathbf{B}$ that is unlikely to occur *within* a sample that intra-sample reuse distance (our current interest) will not capture.

Table VIII shows how `gemm`'s locality metrics differ across access intervals over time. There are three observations. First,

TABLE VIII
DARKNET/GEMM: DATA LOCALITY OVER TIME OF HOT ACCESS INTERVALS.

Access Interval	AlexNet				ResNet152			
	F	ΔF	D	$\mathcal{A}$	F	ΔF	D	$\mathcal{A}$
0	28M	**0.475**	**0.01**	30K	639M	**0.747**	**0.47**	286K
1	55M	**0.675**	**0.02**	30K	772M	**0.799**	**0.57**	293K
2	89M	**0.983**	**0.02**	25K	640M	**0.617**	**2.71**	302K
3	64M	**0.794**	**0.14**	26K	620M	**0.599**	**2.62**	304K
4	39M	**0.489**	**1.64**	29K	591M	**0.574**	**2.69**	302K
5	55M	**0.627**	**1.66**	26K	638M	**0.618**	**2.65**	302K
6	41M	**0.493**	**1.66**	29K	648M	**0.625**	**2.63**	304K
7	38M	**0.644**	**1.49**	17K	549M	**0.514**	**2.66**	312K

TABLE IX
GAP: SPATIO-TEMPORAL REUSE OF HOT MEMORY (64 B BLOCK).

Object	Algorithm	**Reuse** (D)	**Max** D	A	A/block	Time
o-score	pr	**1.13**	**152**	**64K**	0.76	57.2 s
o-score	pr-spmv	**2.41**	**132**	**82K**	1.14	80.1 s
cc	cc	**5.21**	**154**	**581K**	8.87	2.7 s
cc	cc-sv	**0.83**	**36**	**476K**	8.65	45.5 s

AlexNet's ΔF varies more than ResNet152's. This corresponds to AlexNet's varying convolutional, fully connected, and pooling layers compared to ResNet's more consistent convolutional structure. Second, for both networks, spatio-temporal reuse distance (D) *over all objects* increases over time. This corresponds to a decrease in dimension N (`gemm`'s innermost loop) as the networks synthesize data from higher-level feature filters; and for AlexNet N decreases *very* rapidly. Finally, observe that, in contrast to AlexNet, ResNet152's footprint growth (ΔF) tends to decrease over time. This correlates with the changes in dimensions N (decrease) and K (smaller increase), corresponding to `gemm`'s inner two loops.

Our analysis captures memory behavior of the well-known matrix multiplication kernel by time, location, and across time-location. These differing perspectives are critical for capturing a complete picture.

We evaluated optimizations for `gemm` kernel. Without per-layer specialization, the current algorithm uses correct loop ordering and appropriate inner-loop reuse via unrolling. We do not expect tiling to be effective because the matrices are relatively small and it would significantly complicate outer-loop parallelism.

C. GAP's Connected Components & Page Rank

We study the memory effects of different algorithms for each of GAP's PageRank (PR) and ConnectedComponents (CC) specifications. For PR we use pr (Gauss-Seidel) and pr-spmv (Jacobi-style). For CC we use cc (Afforest) and cc-sv (Shiloach-Vishkin). In both cases, the former represents an optimization of the latter. We run each with the same graph size of 2^{22}, i.e., 4 M vertices and 64 M edges. We ignore graph building and focus on the respective algorithm. Location and time analyses are shown in Table IX and Figs. 8 and 9. The time analysis is a histogram plot showing data locality (average) with respect to hot access interval size.

PR's results show the effects of pr's optimized algorithm. With both PR variants, Table IX shows that the hot memory

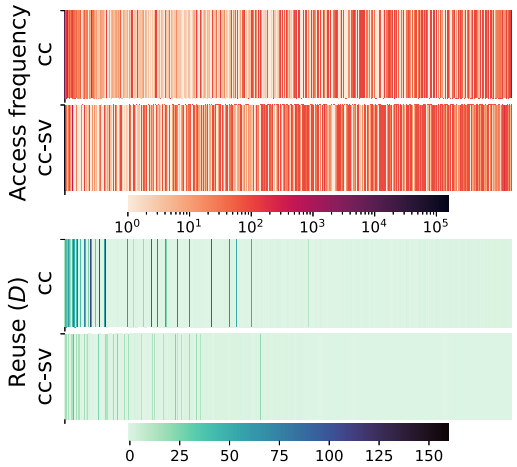


Fig. 8. GAP: Distributions of spatio-temporal metrics for hot memory.

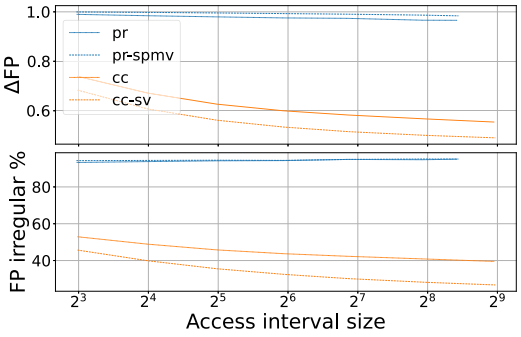


Fig. 9. GAP: Data locality of hot access intervals (intra-sample).

object is *o-score*, representing intermediate contributions to each vertex's final score (rank). With pr-spmv, updates to *o-score* are saved until the next iteration whereas with pr, updates occur immediately. As a result, Table IX shows that pr's spatio-temporal reuse distance (D) is noticeably smaller (better); and its footprint growth (ΔF) slightly smaller. The pr variant requires fewer total iterations and, hence, accesses (A).

CC's results show the value of detailed location analysis for understanding cc's optimized algorithm. The cc variant uses subgraph sampling [38], which requires more accesses (A) but can improve performance for NUMA parallelism vs. cc-sv. Interestingly, a subset of locality metrics could suggest that cc-sv has better locality: cc has higher average reuse distance (D), footprint growth (ΔF), and irregular footprint ratio ($F_{\text{irr}\%}$). However, Fig. 8's detailed heatmaps help explain the difference. The heatmaps show the distributions of access frequencies and reuse distances (D), where darker is higher. For accesses, cc has fewer and smaller dark bands, indicating more access locality than cc-sv. For reuse distance, the heatmaps show that the table's summary metrics are affected by outliers: the average behavior is actually relatively similar. *Thus, to understand CC's differences, it is important to understand memory behavior from many angles and at different resolutions.*

VIII. Related Work

We introduce low-overhead, high-resolution memory analysis with load-level, sequence-aware analysis of data reuse.

High-resolution memory reuse. Current methods for collecting load-level sequences of memory accesses use software-based instruction instrumentation, either with compilers [39] or binary instrumentation [12], [40]. Xiang et al. [12] sample address sequences by enabling and disabling dynamic instruction tracing [41], [42], which results in time overheads of 100×. For more detailed access analysis memory modeling tools [8] and simulators [9]–[11] can be used.

Low-overhead reuse interval analysis. One class of well-known low-overhead methods for analyzing memory accesses target *reuse intervals*, or the time between two accesses to the same address. Reuse intervals can be converted into reuse *distance* [43] (cf. the distinction in §IV-A). Eklov et al. [21] sample reuse intervals, measured in loads, using virtual memory traps. For lower overhead, debug registers can be used to sample reuse intervals [13], [22]. Since there are only 4 such registers, this technique obtains a uniform sample using reservoir sampling. Zhao et al. [44] disable expensive memory tracking of reuse distances when a program's working set size is stable.

In contrast, we sample address sequences rather than reuse intervals, which enables analysis of patterns and sequences in addition to reuse. Further, the incorporation of static analysis to select instrumentation points immediately suggests a variety of extensions to specialize for different use.

Various methods for reuse distance analysis have been designed for storage systems [45], [46]. The SHARDS method uses spatial sampling [45], or monitoring of sampled portions of the address space, which is similar to StatStack [21].

Other low-overhead methods. To capture interaction effects between instructions, ProfileMe monitoring hardware [47], [48] could monitor pairs of instructions. Our work extends this to sequences and data reuse.

Another low-overhead method for analyzing data reuse in memory is to use CPU performance monitoring units to collect sparse address sets or monitor cache behavior. For example, AMD's IBS (or LWP) [17] and Intel's PEBS [18] can collect data addresses. However, the results are so sparse that detecting reuse, much less access patterns, is difficult [49], [50].

Another method estimates data reuse by inferring bounds on footprint through profiles of accesses to each memory hierarchy level [14]. This method can also make some inferences about access patterns. However, the footprint estimates are coarse and qualitative. In contrast, MemGaze enables far more resolution for both data reuse and access patterns.

Thread-level data reuse can be captured by monitoring OS-level virtual memory events and thread interactions. NumaPerf [51], couples this OS-level data with source-code compiler analysis in order to diagnose NUMA-related performance problems.

Performance counters for accesses (loads) and cache locality (cache misses) are frequently used for system adaptation [52] or compiler feedback [53].

Analyzing data reuse. Snir et al. [54] establish limits on data reuse analysis. Yuan et al. [29] provide an excellent overview of temporal data locality of scalar accesses within a trace. Our analysis extends the latter in two ways. First, footprint access patterns provide information on spatial locality and expected behavior of an access stream or dynamic access sequence. Second, we consider static access sequences.

We describe multi-resolution analysis for footprint (volume), footprint growth (rate), and reuse distance; and decompose each by access pattern (strided vs. irregular). Our analysis differs from Xiang et al.'s footprint analysis of sampled traces [12], [29]. The latter's analysis is based on reuse intervals and therefore samples access sequences of varying lengths. In contrast, we sample constant-size sequences of accesses and therefore avoid long reuse intervals (sequences). As mentioned, we also identify common access patterns for insight into spatial locality and expected behavior.

Prior work has combined analysis of spatial locality and temporal locality [5], [6]. However, this work relies on static analysis or coarse (virtual memory) page-level statistics.

IX. Conclusion

The time and space costs (≈100×) of prior methods for sequence-aware analysis of data reuse make them unattractive or unfeasible for many working sets or execution scenarios. We show that Processor Tracing (PT) can be an effective technique for low-overhead, high-resolution memory analysis that includes locations vs. operations, accesses vs. spatio-temporal reuse, and reuse (as distance, rate, and volume) vs. access patterns. We demonstrate load-level analysis of applications, not kernels, because both trace size and resolution are controllable. Using sampled traces that are ≈1% of full ones, we elucidate the memory effects of different data structures and algorithms, including explaining performance differences between prefetching and irregular accesses with good spatio-temporal locality. We show that with a straightforward driver optimization, time overhead for collecting access traces is 10–35% on memory intensive regions and highly correlates with executed `ptwrite`s.

We plan to leverage our work for hardware/software co-design. Using models of different memory systems, we can obtain insight into memory system performance and concurrency with respect to data location, data movement, and workload accesses. We also believe our work motivates further attention to hardware/software co-design of Performance Monitoring Units (PMUs). `ptwrite` functionality generalizes the notion of 'state gathering' when using Performance Monitoring Units and could be used to analyze arbitrary execution state. Further, `ptwrite` generalizes the notion of sequence, permitting both, points (length 1) or true sequence analysis, something that has not thus far been generally possible without binary instrumentation. Finally, `ptwrite` can substantially reduce the number of OS interrupts needed to gather state from PMU

registers if coupled with judicious (e.g., sampled, masked, or predicated) execution.

ACKNOWLEDGMENTS

We thank Xiaozhu Meng (Amazon) for helpful discussions on DynInst. This research is supported by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Orchestration for Distributed & Data-Intensive Scientific Exploration" and "Advanced Memory to Support Artificial Intelligence for Science" and the Data-Model Convergence (DMC) initiative at Pacific Northwest National Laboratory. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[3] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[4] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, vol. 14. Citeseer, 2001, pp. 350–360.

[5] J. Weinberg *et al.*, "Quantifying locality in the memory access patterns of hpc applications," in *Proc. of the 2005 ACM/IEEE Conf. on Supercomputing*, 2005, pp. 50–50.

[6] A. Anghel *et al.*, "Spatio-temporal locality characterization," in *1st Workshop on Near-Data Processing*, 2013, pp. 1–5.

[7] M. Badr *et al.*, "Mocktails: capturing the memory behaviour of proprietary mobile architectures," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 460–472.

[8] T. E. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[9] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[10] S. Li *et al.*, "Dramsim3: a cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[11] Y. Kim *et al.*, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[12] X. Xiang *et al.*, "HOTL: A higher order theory of locality," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 343–356, Mar. 2013.

[13] Q. Wang *et al.*, "Featherlight reuse-distance measurement," in *2019 IEEE Intl. Symp. on High Performance Computer Architecture*, Feb 2019, pp. 440–453.

[14] O. O. Kilic *et al.*, "Rapid memory footprint access diagnostics," in *Proc. of the 2020 IEEE Intl. Symp. on Performance Analysis of Systems and Software*. IEEE Computer Society, Oct. 2020.

[15] X. Liu and J. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proc. of the 2011 IEEE/ACM Intl. Symp. on Code Generation and Optimization*. Los Alamitos, CA, USA: IEEE Computer Society, April 2011, pp. 171–180.

[16] J. Choi *et al.*, "Dancing in the dark: Profiling for tiered memory," in *2021 IEEE International Parallel and Distributed Processing Symposium*, 2021, pp. 13–22.

[17] AMD, "AMD64 architecture programmer's manual volume 2: System programming, 3.37," https://www.amd.com/system/files/TechDocs/24593.pdf, Mar 2021.

[18] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual, combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," 325462-073US, November 2020.

[19] IBM, "POWER9 performance monitor unit user's guide," https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf, Nov 2018.

[20] ARM, "ARM architecture reference manual," https://documentation-service.arm.com/static/, Jan 2021.

[21] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of lru caches," in *IEEE Intl. Symp. on Performance Analysis of Systems Software*, March 2010, pp. 55–65.

[22] M. A. Sasongko *et al.*, "Reusetracker: Fast yet accurate multicore reuse distance analyzer," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, dec 2021.

[23] Intel Corporation, "Intel processor tracing," https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, Aug 2013.

[24] ARM, "Embedded trace macrocell architecture specification," https://documentation-service.arm.com/static/5f90158b4966cd7c95fd5b5e, 2011.

[25] Intel Corporation, "Intel architecture instruction set extensions and future features (319433-044)," https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html, May 2021.

[26] Linux perf, "perf-intel-pt," https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html, May 2021.

[27] ——, "perf-mem," http://man7.org/linux/man-pages/man1/perf-mem.1.html, January 2021.

[28] X. Meng and B. P. Miller, "Binary code is not easy," in *Proc. of the 25th Intl. Symp. on Software Testing and Analysis*, 2016, pp. 24–35.

[29] L. Yuan *et al.*, "A relational theory of locality," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, Aug. 2019.

[30] R. L. Mattson *et al.*, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[31] S. Ghosh *et al.*, "miniVite: A graph analytics benchmarking tool for massively parallel systems," in *Proc. of the 9th Intl. Workshop on Performance Modeling, Benchmarking, and Simulation of High-Performance Computer Systems*, 2018.

[32] S. Beamer *et al.*, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[33] J. Redmon, "Darknet: Open source neural networks in C," https://pjreddie.com/darknet, Jul 2015.

[34] T. G. Tessil, "Tessil hopscotch-map," github.com/Tessil/hopscotch-map, Aug 2019.

[35] M. Herlihy *et al.*, "Hopscotch hashing," in *Distributed Computing*, G. Taubenfeld, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 350–364.

[36] A. Krizhevsky *et al.*, "ImageNet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[37] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[38] M. Sutton *et al.*, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium*, 2018, pp. 12–21.

[39] K. Beyls and E. H. D'Hollander, "Intermediately executed code is the key to find refactorings that improve temporal data locality," in *Proc. of the 3rd Conf. on Computing Frontiers*, ser. CF '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 373–382.

[40] G. Marin *et al.*, "MIAMI: A framework for application performance diagnosis," in *Proc. of the 2014 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Mar 2014, pp. 158–168.

[41] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[42] Intel Corporation, "Pin - A dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, June 2018.

[43] X. Shen *et al.*, "Locality approximation using time," in *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 55–61.

[44] W. Zhao *et al.*, "Low cost working set size tracking," in *Proc. of the 2011 USENIX Conf. on USENIX Annual Technical Conference*, ser. USENIXATC'11. USA: USENIX Association, 2011.

[45] C. A. Waldspurger *et al.*, "Efficient MRC construction with SHARDS," in *13th USENIX Conf. on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 95–110.

[46] J. Wires *et al.*, "Characterizing storage workloads with counter stacks," in *11th USENIX Symp. on Operating Systems Design and Implementation*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 335–349.

[47] J. Dean *et al.*, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proc. of the 30th Annual ACM/IEEE Intl. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 292–302.

[48] B. A. Fields *et al.*, "Interaction cost and shotgun profiling," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 3, pp. 272–304, Sep. 2004.

[49] P. Roy *et al.*, "Lightweight detection of cache conflicts," in *Proc. of the 2018 Intl. Symp. on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 200–213.

[50] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," *SIGPLAN Not.*, vol. 49, no. 8, pp. 259–272, Feb. 2014.

[51] X. Zhao *et al.*, "Numaperf: Predictive numa profiling," in *Proc. of the ACM Intl. Conf. on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 52–62.

[52] Y. Chen *et al.*, "ATMem: Adaptive data placement in graph applications on heterogeneous memories," in *Proc. of the 18th ACM/IEEE Intl. Symp on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 293–304.

[53] D.-J. Oh *et al.*, "Maphea: A lightweight memory hierarchy-aware profile-guided heap allocation framework," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 24–36.

[54] M. Snir and J. Yu, "On the theory of spatial and temporal locality," University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2005-2611, http://hdl.handle.net/2142/11077, July 2005.