MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors

Alfredo Giménez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann

Abstract—Memory performance is often a major bottleneck for high-performance computing (HPC) applications. Deepening memory hierarchies, complex memory management, and non-uniform access times have made memory performance behavior difficult to characterize, and users require novel, sophisticated tools to analyze and optimize this aspect of their codes. Existing tools target only specific factors of memory performance, such as hardware layout, allocations, or access instructions. However, today's tools do not suffice to characterize the complex relationships between these factors. Further, they require advanced expertise to be used effectively. We present MemAxes, a tool based on a novel approach for analytic-driven visualization of memory performance data. MemAxes uniquely allows users to analyze the different aspects related to memory performance by providing multiple visual contexts for a centralized dataset. We define mappings of sampled memory access data to new and existing visual metaphors, each of which enabling a user to perform different analysis tasks. We present methods to guide user interaction by scoring subsets of the data based on known performance problems. This scoring is used to provide visual cues and automatically extract clusters of interest. We designed MemAxes in collaboration with experts in HPC and demonstrate its effectiveness in case studies.

Index Terms—Performance Visualization, High-Performance Computing, Memory Visualization

1 Introduction

High performance computing (HPC) systems consist of large-scale parallel computers that solve computationally intensive problems and often feature complex and long-running applications. Such systems enable many forms of scientific inquiry, but their computational capabilities vastly outpace the abilities of applications to take advantage of them, leaving much of their potential computational power largely unused. It is becoming increasingly important to understand and analyze the behavior and performance of HPC applications in an effort to optimize them. However, this is an extremely difficult task: depending on the hardware and software properties of a particular system, the behavior of an application can be impacted by several different factors. In many cases, memory access performance plays a dominating role, due to the fact that raw computational performance has advanced (and continues to advance) at a much higher rate than that of data access performance. This trend is also referred to as the "Memory Wall" [30].

To counteract this trend, hardware designers have introduced deeper, more complex memory hierarchies. While software researchers have developed a wide variety of strategies to take advantage of them [22], [17], no general solution is currently capable of achieving performance anywhere near peak memory performance. Different application and hardware combinations have vastly different memory usage requirements and capabilities, explaining the need for analysis and optimization of individual application executions.

Hardware simulators present one option, as they can simulate the execution of each instruction on a particular piece of hardware, with the goal being to expose detailed memory behaviors at a low level. However, these simulators incur large overheads and often fail to correctly predict total system behavior by neglecting outside variables, such as operating system interference, frequency scaling due to heat, and non-deterministic effects resulting from concurrent execution. For these reasons, optimizing memory performance typically requires one to collect memory-related performance data from actual, non-simulated runs and perform analyses on the collected data.

Accurate collection of memory-related performance data is a field of research on its own. The overhead involved in collecting data makes necessary the use of hardware resources, including memory. Thus, high-overhead solutions corrupt the observed information. Not collecting sufficiently enough information makes it impossible to perform a useful analysis. To understand the execution of a single memory instruction, we need to know which line of source code generated the instruction, which data structure was accessed, which CPU was used for execution, and which memory resource was accessed. Moreover, with modern multi-level caching, the performance of one memory instruction can depend on those that happened before. In prior work, we developed a method for collecting the needed fine-grained memory access information with low overhead [10]. In this paper, we introduce visualization and analysis methods that use this information to characterize memory performance with respect to the many different factors that contribute to it.

- *Alfredo Giménez (alfredo.gimenez@gmail.com) and Bernd Hamann (hamann@cs.ucdavis.edu) are with the University of California, Davis.*
- *Ilir Jusufi (ilir.jusufi@lnu.se) is with Linnaeus University in Växjö, Sweden.*
- *Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Peer-Timo Bremer are with Lawrence Livermore National Laboratory. Emails: {tgamblin,bhatele,schulzm,ptbremer}@llnl.gov*

Manuscript received February 29, 2016

Contributions

We present the memory performance visualization and analysis tool MemAxes. MemAxes enables novel analysis of the complex relationships between source code, data structures, and hardware by mapping the input data to multiple different visual representations, each exposing different characteristics of memory behavior. This unique combination of characteristics more completely describes complex memory behavior than is possible with existing methods, and as a result, our method makes it possible to more accurately diagnose memory performance problems than existing methods. In addition, MemAxes introduces a new way to visually characterize utilization of the hardware involved in memory accesses. Finally, MemAxes is based on a new method that guides the process of data exploration by "scoring" various subsets of the performance data and presenting the results as possible interactive data exploration choices to the user.

2 Related Work

Many open-source and commercial tools exist for analyzing application performance, such as GNU gprof [11], Intel VTune [23], and Apple Instruments [2], that incorporate varied degrees of visual analysis methods. Depending on the particular analysis task, different forms of visual analysis are more or less appropriate. We briefly review a subset of the related research that involves memory and memory-related data and refer the reader to [15] for an in-depth survey of the performance visualization field at large.

Visualization of Memory Allocation. One of the motivations driving memory performance visualization techniques is the desire to understand how data is laid out in address space. Considering the design of modern memory hierarchies, the location of a particular data item in address space has a significant effect on how quickly it can be accessed. It is critically important to understand how memory allocation and utilization affects the performance of a complex computer code and how to optimize it. In the context of visualization, the problem is to depict a set of ranges (allocated memory buffers) along a single dimension (the total available address space) as they change over time (allocation, de-allocation, re-allocation). Further, one must visualize when accesses are made and by what parts of the code.

Griswold et al. devised a method that plots allocated memory buffers as blocks along an axis representing address space, encoding the buffer type via color and wrapping the axis across rows to more efficiently utilize screen space [12]. While providing a detailed visualization of memory fragmentation for small memory spaces, this method was designed to show the entirety of the memory address space and allocations of a complete program, which is simply too much information to show in its entirety, even for moderately complex applications. Furthermore, this visualization method represents the state of memory utilization for only a single point in time. Moreta et al. presented an improved approach based on plotting time as an axis orthogonal to address space [21]. However, this approach further limits the available real estate for visualizing for address space, as all allocations for a single time point are plotted along a single row. Cheadle et al. introduced a similar method for showing the address space of the heap by plotting allocations via a heat map. Their method also shows histograms of different allocations separated by size and type [7]. The resulting visualizations effectively show the distribution of different allocations for a program, but the screen space required scales linearly with respect to the number of allocations.

The methods described above are more or less effective regarding the goal of understanding memory allocation characteristics. All are severely limited concerning scalability. A modern application typically involves several thousand memory allocation operations performed per millisecond, and modern platforms can use several million memory addresses for a single process. Furthermore, address space is a virtual construct; the data and addresses also have physical counterparts, i.e., cache hardware, memory banks, etc., and their properties greatly affect memory access efficiency. The location of memory and cache resources, their efficiency with respect to the different processors that access them and their eviction policies play crucial roles in achieved performance. Therefore, virtual address space alone is insufficient for characterizing memory performance behavior.

Visualization of Hardware Topologies. The physical analog to the virtual address space is the physical addresses covering all memory hardware resources in a particular system. However, the available hardware performance data often lacks the information to connect it to the virtual address space used by the application. Hardware-centric visualizations should expose performance characteristics that effectively depict utilization across hardware resources, disjoint from the address space visible to the application developer.

Memory hardware resources include main memory, caches, and their connectivity to each other and to processors. The layout and connectivity of memory hardware resources define the *hardware topology* of the system. This topology has a hierarchical structure, with processors defining the leaves, with parent nodes representing the next-smallest available memory resource (L1 cache), until the largest memory resource (main memory) is reached at the root.

Early research done by Alpern et al. focused on connections between the processor and lower levels in the cache hierarchy [4] to illustrate the intended behavior of tiling algorithms. Their research was one of the first efforts directed at understanding the relationship between algorithms, data, and hardware used. Unfortunately, this research effort was purely conceptual and did not address the visualization of real data. Choudhury and Rosen developed a technique for depicting data movement within a simulated cache hierarchy using an abstract visualization showing individual cache lines and their transactions [8]. Their approach effectively extended Alpern et al.'s effort by showing data elements in their respective hardware containers, using visual abstraction as a tool to improve comprehensibility and utilizing visualization space more effectively. Further, their method links transactions to the instructions in the code that caused them. However, the approach is only effective for hardware with caches small enough to have individual cache line accesses visually discernable. Rosen et al. introduced an approach for visualizing memory accesses for representative subsets of CUDA hardware performance data [24]. The idea used to detect representative subsets is highly effective for reducing the visual exploration space into semantic regions, which is necessary to support the viewing of several hundred GPU cores and several thousand timesteps per execution. Unfortunately, the approach is highly hardware-specific.

The software tools *hwloc* [6] and *likwid* [28] provide capabilities to detect, output, and visualize hardware topologies, but they do not allow one to directly visualize performance data mapped to the topologies. Recently, Denoyelle et al. [9] built upon *hwloc* by color-coding performance counters mapped on the visual components representing different hardware resources. The result is a highly advanced hardware resource monitor. However, the tool does not facilitate analysis of specified time intervals; only an aggregated summary of performance for a single time point is supported.

Visualizations that incorporate hardware topology expose a myriad of hardware-dependent performance problems that would otherwise remain unseen. However, currently one either relies on simulated data to represent attributions between the hardware and software or this attribution information is not represented at all. As a result, current tools can shed light on hardware behavior, but not on their root causes within a simulation code.

Visualization of Memory-related Counters. Memory performance counters are available on most modern processors. They record the number of occurrences of specific events, e.g., cache misses or accesses to main memory over time. Each counter may be associated with a particular resource, e.g., a processor core or memory resource. MView [5] displays memory access counters on a Gantt chart, with different processors on the horizontal axis and different counters encoded via color and height. This approach effectively shows the distribution of memory-related events on a per-processor basis and scales relatively well using common interaction methods for Gantt charts. Schulz et al. [25] described a technique for aggregating memory counters by their associated locations in a physical mesh used for simulation. The result is highly scalable, but also highly application-specific.

The major drawback regarding visualizing counter data is attribution. Counters provide the number of events over time periods, but do not provide details about those particular events. Thus, events cannot be directly attributed to the instructions that caused them or the data addresses involved.

Visualization of Sampled Memory Accesses. Our research presented here concerns sampled memory instruction data, which has only become available with recent microprocessor generations. Sampling has the advantage of providing highly detailed information about individual events that occur during program execution compared to the summary information provided by counters. This enhanced information leads to a significant overhead cost compared to counters, and sampling must be carefully configured to mitigate this trade-off. Liu et al. extended *HPCToolkit* to include memory access sampling data in order to attribute memory accesses to recorded call paths and associated memory allocations [19], [3]. This attribution provides valuable information. However, this toolkit does not provide more advanced visualizations beyond scatterplots and is limited in interaction techniques. It relies heavily on a user's ability to explore the data in meaningful ways.

3 Requirements

Based on the benefits and drawbacks of the current state-of-the-art methods in memory performance visualization, we identified a set of requirements for our tool:

Scalability. The growing requirements of high-performance computing applications make scalability critical. This includes both visual scalability (how the visualization quality scales with respect to the input data size) and computational scalability (how quickly the tool runs with respect to the input data size). Many of the aforementioned tools were capable of handling input sizes typical of the era in which they were presented, but the quantity of data produced from hardware has since grown exponentially. Currently, it would not be unreasonable to expect hundreds of thousands of data points with 10-20 dimensions each, and we can only expect this number to increase. Ideally, the visual quality of our tool would be unaffected by the input dataset and the tool would perform at interactive rates regardless of data size.

Attribution. In order to perform meaningful analysis, the tool should be able to not only show aspects of memory performance behavior but their causes as well. Repeatedly, we have seen that a single visualization may be effective in showing one or two different characteristics of memory behavior, but unless we are able to correlate those characteristics with others, we can not hypothesize any causal relationships between them. For example, hardware-centric visualizations may show whether accesses were made to slow memory resources, but without attributing those accesses to their respective lines of source code, it is difficult to determine what aspects of the code and data structures may be responsible for them. Attributing data across visualizations allows us to determine what characteristics may be related and thus provides us a set of potential correlations to draw conclusions from.

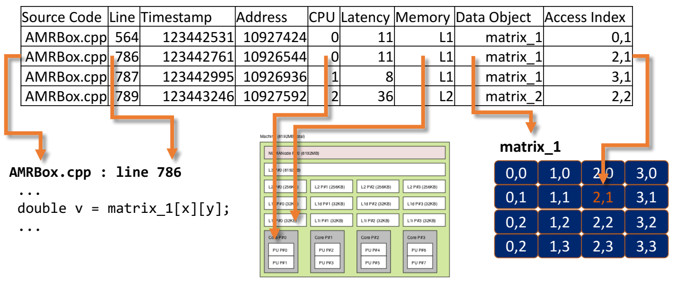
Usability. Many of the aforementioned methods provide a large visual exploration space and the necessary interaction tools to navigate it. However, doing so often requires the analyst to have a substantial amount of prior knowledge of their application and computing environment in order to perform useful interactions for analysis. This requirement imposes a significant impediment on the usability of these tools. To mitigate this, we require simple and intuitive methods for guiding the interactive analysis process without substantial prior knowledge, in addition to providing basic interaction techniques for exploratory analysis. We expand upon this point in Section 7.

4 Memory Performance

Modern memory architectures typically contain three levels of cache and either a single main random-access memory (RAM) resource or multiple non-uniform memory access (NUMA) resources. The time and energy required to access a memory resource increases exponentially with the size of the resource [13], so most often it is better to utilize small cache resources. Caches may be either shared or unshared among different processors. Typically, architectures include one or two unshared caches per processor and possibly a shared cache for a set of processors. Main memory resources are shared among all processors but are the largest and slowest to access, and NUMA resources are local to a subset of processors on a shared *socket* and incur a high penalty for accesses from processors on a remote socket.

Problems arise when application data cannot fit into small caches and when multiple processors contest for shared resources, both of which are almost always the case, especially in high-performance computing (HPC) applications. The former issue requires processors to evict data from smaller caches to replace it with incoming data, forcing subsequent accesses to acquire data from larger, slower resources. The latter also requires processors to communicate across caches in order to ensure that processors do not operate on stale copies of cached data.

A wide variety of factors affect the outcome of different memory access patterns, as discussed in Section 1. By analyzing the memory accesses in the contexts of these different factors, we are able to gain an understanding of the interplay between



Source Code	Line	Timestamp	Address	CPU	Latency	Memory	Data Object	Access Index
AMRBox.cpp	564	123442531	10927424	0	11	L1	matrix_1	0,1
AMRBox.cpp	786	123442761	10926544	0	11	L1	matrix_1	2,1
AMRBox.cpp	787	123442995	10926936	1	8	L1	matrix_1	3,1
AMRBox.cpp	789	123443246	10927592	2	36	L2	matrix_2	2,2

```
AMRBox.cpp : line 786
 ...
 double v = matrix_1[x][y];
 ...
```

Fig. 1: The table above shows some hypothetical memory access data. Each row represents a single memory access, and different columns describe different attributes associated with the access. These attributes make it possible to associate the access in different contexts, for example the source code (left), hardware topology (middle, see Fig. 4), and data layout (right).

them and in turn formulate a high-level explanation for certain memory behaviors. Finally, high-performance computing experts can use this information to form hypotheses for potential causes of problems and determine possible optimization methods.

4.1 Memory Access Samples

As described in Section 1, there exist many different types of memory performance data, each of which may provide different insights. In this work we focus on the data acquired from memory access sampling. We configure an available performance monitoring unit in hardware to record a memory access after every N accesses have completed, effectively giving us a subset of all accesses made during a program's execution [14].

Sampling can be configured for different values of N, and in addition, a latency threshold T may be specified. In the latter case, memory accesses are only recorded if their duration exceeds a threshold of T CPU clock cycles. Different N configurations allow the user to trade off sampling granularity and overhead. Because a low granularity may possibly miss important memory behaviors, it is helpful to also specify a T to bias the sampling towards slower accesses, which may represent bottlenecks and are potentially of more interest.

Every sample is essentially a tuple of attributes associated with a single memory access event. Different attributes describe the memory access in different contexts, for example in the context of the source code, the hardware, or the data layout. Figure 1 shows how the different attributes map to these various contexts. Some attributes are quantitative, representing strictly ordered quantities, and others qualitative, describing different aspects of the access. Although some values like processor ID are numerical, their numeric value bears no meaning—processor ID values may be arbitrarily ordered (and often are), therefore they must be treated as qualitative. Specifically, the hardware-provided attributes contained in an individual sample are:

Timestamp (quantitative). A timestamp taken at the time the access retires, a numerical value.

Latency (quantitative). The number of elapsed processor cycles from access request to access retirement.

Processor (qualitative). The ID of the processor that issued the access.

Instruction Pointer (qualitative). The address of the instruction that issued the access request.

Data Source (qualitative). An encoding specifying the type of resource where the access was resolved, e.g. L1 cache, remote RAM .

Data Address (qualitative). The pointer address of the data being accessed.

We extend this list of attributes using developed methods in the high-performance computing community [19], [10] to additionally obtain:

Data Symbol (qualitative). The name (as a string) of the symbol from which data was accessed, e.g. "myArray".

Access Index (quantitative). The index used to access the data element from its container, e.g. the value i in "myArray[i]".

State Variables (varies). Any additionally recorded pieces of information, as specified by the programmer.

As a result, each memory access sample can be interpreted as a multidimensional point with a minimum of 8 dimensions and a virtually limitless number of additional program state variables.

In addition to the memory access samples, we store the hardware topology on the executing system using the well- established *hwloc* [6] tool. The topology has a hierarchical format, with larger memory resources parenting smaller ones, and processors at the leaves below the smallest (L1 cache) memory resource.

5 Design Overview

Due to the multidimensional nature of the input data, it is tempting to apply existing multidimensional visualization techniques and create a single, all-encompassing view. However, such techniques typically rely on the assumption that all dimensions may be treated in the same way and compared to one another on a common metric. Because the dimensions of our input data are highly heterogeneous, this is not possible to do directly, and mapping these dimensions to a homogeneous analog strips the data of its informational value. The result is a visualization that attempts to do everything but accomplishes very little in terms of specific analysis tasks within the domain.

The most effective performance visualizations were those that tied a particular data exploration or analysis task to a representative visual metaphor. For example, Griswold et al. [12] and Moreta et al. [21] showed that when analyzing memory fragmentation, it is helpful to use a visual metaphor for a program's memory address space. However, as previously noted, such designs also limit the types of exploration or analysis that may be done. For this reason, we decided to design MemAxes with multiple *context-aware* views, each of which is suited to a particular set of characteristics relevant for memory performance analysis. In addition, by linking context-aware views, MemAxes makes it possible to identify correlative relationships between these different characteristics. Thus, this design satisfies our original requirement to make it possible to attribute performance characteristics to one another.

This design also requires effective navigation by the user, however, and the interactive search space of this data is vast. In order to make this space navigable, we developed methods for *guided interaction* and *cluster extraction*, which communicate to the user what selections are of potential interest. The result is a tool that users with limited knowledge can still glean useful insights

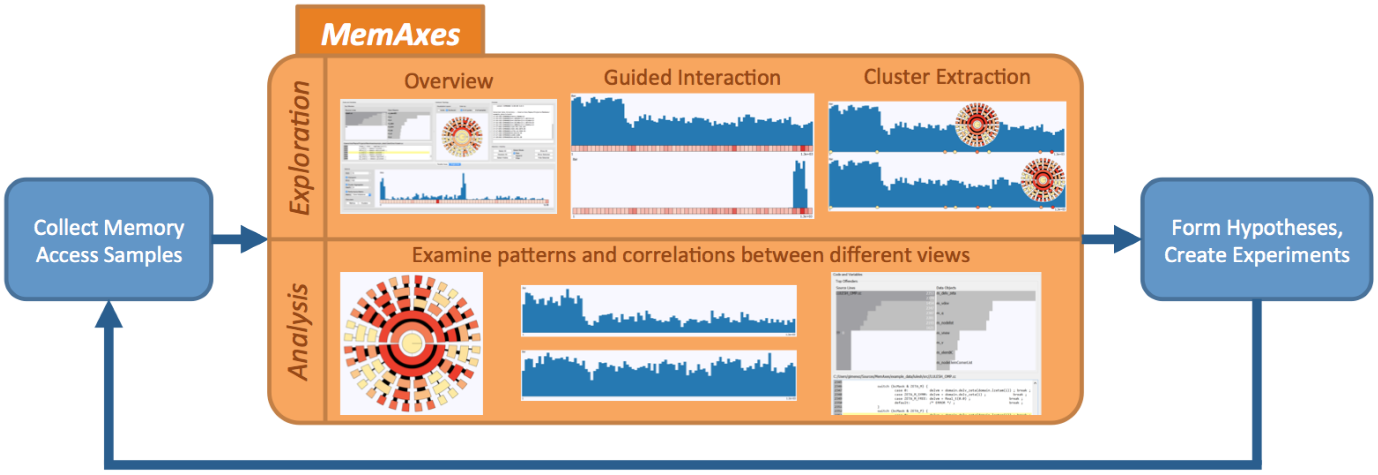

MemAxes
Exploration
Overview
Guided Interaction
Cluster Extraction
Collect Memory Access Samples
Form Hypotheses, Create Experiments
Analysis
Examine patterns and correlations between different views

Fig. 2: Exploration and analysis workflow for characterizing memory performance using MemAxes. Initially, MemAxes shows the overview of all memory access samples, then through guided interaction and cluster extraction the user may select regions of interest. On any selection of data, the analyst may draw conclusions by identifying patterns in different views and seeing how patterns between multiple attributed contexts correlate with one another.

from while not limiting the available interactions from more knowledgeable users, thus satisfying the usability requirement.

In order to fulfill the requirement that MemAxes be visually scalable, all visualization methods used rely on some form of aggregation. The aggregations used produce visual elements that are invariant to the data size, so visual clutter remains constant for all inputs. MemAxes is computationally scalable up to a limit—nearly all interactions incur basic set operations that make use of well-known optimizations.

The exploration and analysis workflow is illustrated in Figure 2.

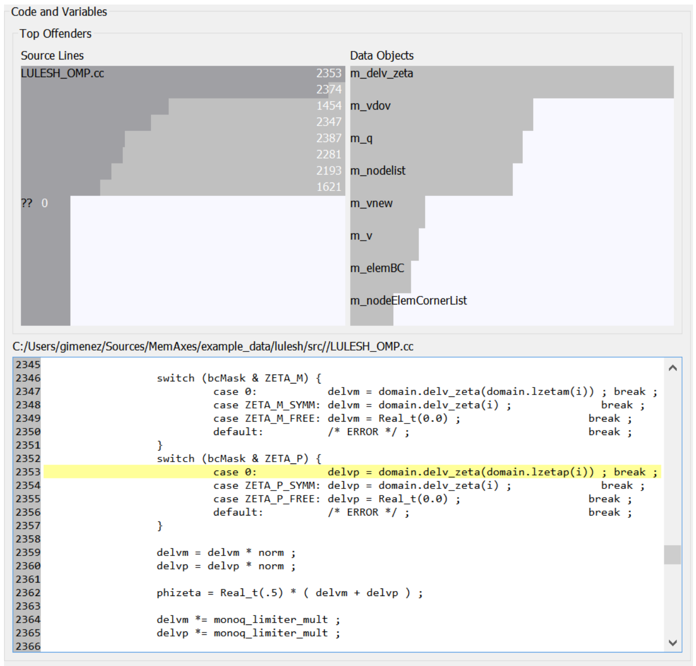
6 Views

MemAxes is comprised of five linked interactive views: two *context-aware* views (Fig. 3, Fig. 5, one multidimensional view (Fig. 6), and two single-dimensional views. The context-aware views aggregate and present a subset of the dimensions relevant for performance analysis and will be discussed in detail. To avoid limiting the user to exploring subsets of the dimensions, we included a multidimensional view showing all dimensions as well as two single-dimensional views in which the user may focus on certain dimensions in detail. The first shows a standard histogram over a single numerical dimension (such as time), and the second shows a histogram of qualitative values (such as instruction) sorted in order of appearance.

We identified two contexts most relevant to the problems in memory performance based on related work and our experience in HPC and developed context-aware views for each: (1) the program instructions and data, and (2) the hardware topology. For the multidimensional view, we developed (3) histograms along each dimension plotted in parallel, deemed parallel histograms. We omit a discussion of the single-dimensional views, as they are straightforward.

6.1 Instructions and Data

Many previous performance visualization methods have overlaid performance data onto the source code, and for good reason—optimization efforts most often require modification of the code



C:/Users/gimenez/Sources/MemAxes/example_data/lulesh/src//LULESH_OMP.cc

```

            switch (bcMask & ZETA_M) {
                case 0:          delvm = domain.delv_zeta(domain.lzetam(i)) ; break ;
                case ZETA_M_SYMM: delvm = domain.delv_zeta(i) ;               break ;
                case ZETA_M_FREE: delvm = Real_t(0.0) ;                      break ;
                default:          /* ERROR */ ;                              break ;
            }
            switch (bcMask & ZETA_P) {
                case 0:          delvp = domain.delv_zeta(domain.lzetap(i)) ; break ;
                case ZETA_P_SYMM: delvp = domain.delv_zeta(i) ;               break ;
                case ZETA_P_FREE: delvp = Real_t(0.0) ;                      break ;
                default:          /* ERROR */ ;                              break ;
            }

            delvm = delvm * norm ;
            delvp = delvp * norm ;

            phizeta = Real_t(.5) * ( delvm + delvp ) ;

            delvm *= monoq_limiter_mult ;
            delvp *= monoq_limiter_mult ;

```

Fig. 3: Visualization of memory access data in the context of the top offending lines of source code (top left) and data objects (top right). The topmost offending line of source code is shown highlighted within its containing file in the source pane (bottom).

or at least an understanding of how it affects performance. Many tools (e.g. HPCToolkit [3], Vtune [23]) show performance data embedded in *callpaths*, which are the recorded sequences of function calls in a program execution. These tools often provide the capability for the user to select a function in the callpath and see it highlighted in the source code file. Typically, the goal in this context is to identify which parts of the code contribute most to execution time for the purpose of determining which parts of the code require optimization.

With this in mind, we designed a simple visualization to enable the user to quickly identify the *top offenders* in terms of the instructions and data symbols in the source code. Top offenders are the components of the code that have the greatest contribution to the number of memory access cycles in execution. The relevant attributes of a memory access sample in this context are the lines of source code for particular instructions and the data symbols accessed. We calculate the total access cycles associated with these attributes by aggregating latency along them, and we sort the resulting aggregations in decreasing order. These results are the top offending lines of code and data symbols for memory performance in ranked order.

We visualize the top offenders using a horizontal bar chart, which effectively shows the ranking of top offenders as well as the differences between them in contribution. The user interface of MemAxes includes a pane showing the top offending lines of source code and data objects side-by-side above a text editor showing the topmost offending line of source code highlighted within its source code file, as shown in Figure 3.

The user is able to select top offenders by clicking on the appropriate bar in the chart. In doing so, all samples associated with that offender, either line of code or data object, are selected. If a selection has been made, the top offenders and the highlighted line in the text editor show only selected sample information, otherwise they show all samples. Thus, by selecting a line of code, the user may immediately see which data objects were accessed by that line, and conversely, by selecting a data object, the user may see which lines of code accessed it. Doing so may reveal whether the source of a memory performance bottleneck lies in the way in which a data object was accessed or the layout of the data object itself. The user may also determine the overall contribution of a line of code or data object with respect to the total execution time of the application by selecting either one, upon which basic statistics of the current selection are displayed in a separate pane (not shown).

6.2 Hardware Topology

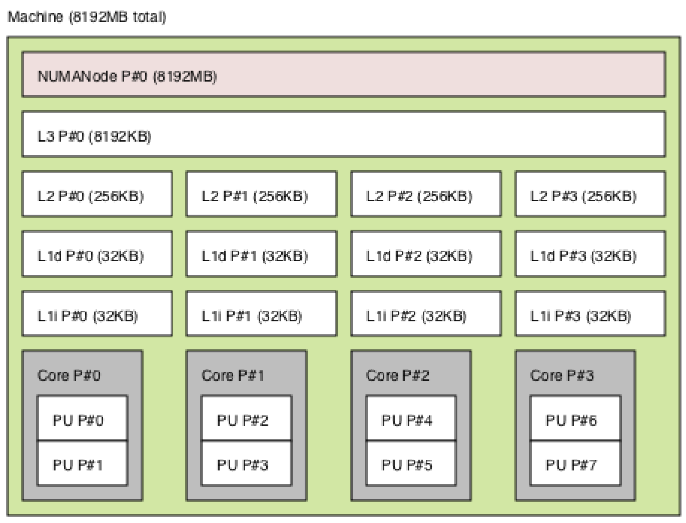


Fig. 4: The hardware topology visualization produced by *hwloc*, which uses a horizontal layout. The pink rectangle denotes a single RAM resource [6]. The caches are drawn underneath in white, processor cores dark gray, and processing units (CPUs) embedded inside the cores.

Existing tools for visualizing hardware topology are aimed at showing resource locality and basic hardware layout. For example, *hwloc* (Figure 4) and *likwid* (not shown) both depict the hierarchy of the hardware topology using a form of icicle plot. We identified two main drawbacks in the current approaches: they poorly use visual real estate and thus do not scale well to complex topologies, and no existing methods are able to visualize acquired memory access samples in the context of the hardware topology. We designed a new visualization for the hardware topology with improved scalability for complex architectures and capability to visually embed memory access samples via a function that maps them to the visual components of the hardware.

Visual Layout. The hardware topology may be interpreted as a hierarchy where a single computation node represents the root, memory resources (RAM, NUMA, caches) represent internal nodes, and processors (CPUs) represent the leaves. As such, hierarchical visualizations readily apply to visualizing hardware topology. However, the majority of visualizations show the topology using horizontal layouts, commonly called icicle plots [18], such as the one in Figure 4. This layout allocates the same amount of visual space for each level in the hierarchy; however, by design, hardware topologies have few resources in the top levels of the hierarchy (larger, slower memory resources) and many in the lower levels (smaller, faster caches and many processors). As a result, an space is wasted showing the few large memory resources, and the number of processors that may be viewed at once is limited by the horizontal viewing space. As a solution, we use a radial, space-filling visual layout for the hardware topology, essentially a sunburst chart [26], as shown in Figure 5 (a). By instead allocating *less* space to higher levels in the hierarchy and *more* space to lower levels in the hierarchy, this solution is much more well-suited for hardware topology.

We color map the nodes to show resource utilization, either by total number of cycles or samples. We also scale the color maps relative to the minimum and maximum of all resources of the same depth in the hardware hierarchy. While the same colors represent different values on different levels of the hardware hierarchy, this is preferable to using a single scale for all values; latencies in L1 cache are on the order of tens of cycles, whereas those in NUMA exceed thousands. This visual encoding therefore accomplishes the goal of displaying relative utilization patterns across equivalent resources. The user may observe exact values by hovering the cursor over a particular resource, upon which a tooltip with detailed information appears.

Additionally, we add lines between nodes to depict resource connectivity and scale the thicknesses of these lines to show data traffic between resources. Again, the thickness scale is relative to the minimum and maximum among all values of a certain depth in the hierarchy, making relative differences apparent between values of the same magnitude. While this gives a good general idea of comparisons between links, the visual range is limited to the available thickness of the lines, so we also allow the user to hover over them and view exact values via tooltips.

As a result, we are able to display relative utilization of processors, memory resources, and data traffic between resources onto our visual representation of hardware topology, as shown in Figure 5 (b).

Mapping Samples to Hardware Topology. In order to visualize accesses in this layout, we defined a function mapping each sampled memory access to a set of associated resources and links.

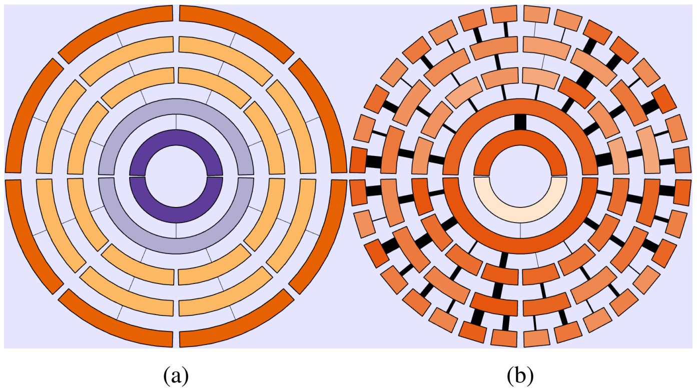


Fig. 5: (a) The radial layout of the hardware topology in MemAxes for a simple architecture. Main memory resources are dark purple, L3 caches are light purple, L1 and L2 caches are light orange, and processing units are dark orange. (b) A more complex architecture with performance data annotated onto the hardware resources, via a color map, and transactions between them, via line width.

We then aggregate all accesses over the resources and links and embed the resulting aggregations into the visualization.

Each access has an associated depth in the memory hierarchy and an an associated processing unit (CPU). These represent the level in cache or memory where a piece of memory was accessed from and the CPU that requested the access, respectively. For each access, we perform a tree traversal starting from the CPU that issued the access and ending at the depth where the access was resolved. The traversed path represents the actual motion of the data that was accessed for a particular memory access sample; the CPU searches for data first in the nearest cache, and if it fails, it searches the next nearest cache, and so on until the request is satisfied. The data is then copied to all smaller caches between the level in which it was found and the CPU, thus incurring traffic on the links between the resources on this path.

If we can define an aggregation function for an attribute, such as sum/average/deviation (if quantitative) or count/cardinality (if qualitative), we can then use this function to aggregate the attribute along the traversed path. For most types of performance analysis, it is useful to aggregate the number of accesses that traversed each link and the sum of latencies for all accesses to each resource, so this is presented by default. The resulting visualization depicts the distribution of data transactions across links as well as the distribution of access cycles across resources, as seen in Figure 5 (b). Thus, if a resource has high outgoing traffic but low cycles, we can conclude that it is being used often and efficiently, and if the opposite is true there may be a performance problem.

Because aggregation schemes provide a summary of the input data, it is critical for the user to define or discover interesting subsets for analysis, rather than attempt to analyze an aggregation of the entire dataset. A particular phenomenon of performance behavior may only be visible under a highly specific selection of the dataset. Interactive selection and filtering enable the user to navigate these subsets, but determining an effective path of interaction to find interesting subsets is not trivial. We address this problem in the following Section 7.

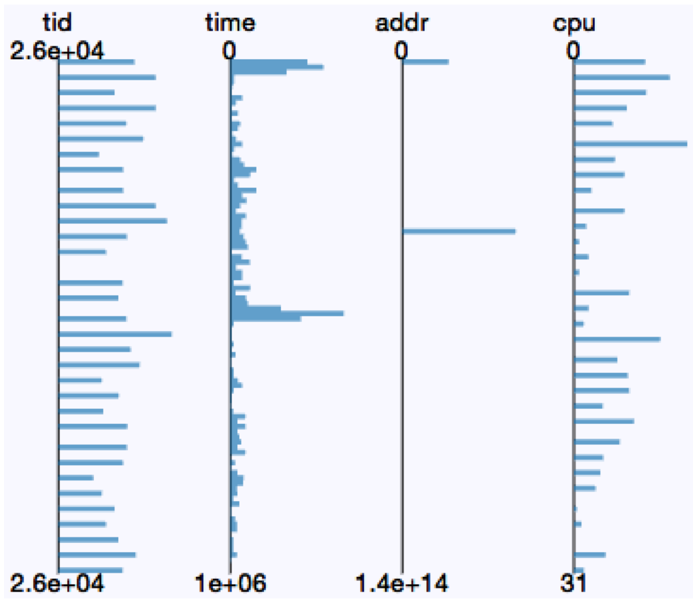


Fig. 6: Parallel histograms of memory access samples. Each attribute is plotted along a vertical histogram and all attribute histograms are placed side-by-side in parallel.

6.3 Parallel Histograms

Though context is highly valuable from an analysis standpoint, it also limits the number of visualizable attributes. In order to include the missing details, we included a context-unaware visualization we call *parallel histograms*. Parallel histograms are histograms of each attribute plotted in parallel (similar to parallel coordinates), as shown in Figure 6. The use of parallel axes makes it possible to plot any number of attributes.

Histograms are straightforward to produce for continuous attributes, such as time. For qualitative attributes, such as data objects, we assign unique identifiers to unique values, ordered by appearance. By using histograms, the data is aggregated into histogram bins, allowing for this view to scale to large numbers of samples.

Parallel histograms allow the user to easily view distributions of samples within every dimension of the access data. As selections are made in context-specific views, the user can identify whether any of those selections are correlated with an individual dimension not shown in the other views. Most importantly, through parallel histograms the user may make ranged selections along each axis by clicking and dragging along it and may combine ranged selections as well. Thus, parallel histograms complement context-specific views by providing comprehensive selection and visualization capabilities.

This combination of linked visualizations under a central dataset allows users to explore the relationships between the various actors in memory performance. For example, by selecting a top offending line of source code, we are able to determine not only which data objects contributed most to access time, but which individual elements inside the data object did. Individual elements may be slow to access if the data structures are not optimized for locality, in which case the hardware is unable to effectively use caches. It may also be the case that the decomposition of data to hardware threads limits the hardware from utilizing shared caches. This decomposition is dependent upon the topology as well as the data layout and access patterns, and as such it is imperative that the user analyze how all three contexts relate to one another to understand the memory performance behavior.

7 Guided Interaction

MemAxes provides a comprehensive set of interaction and exploration tools through linking and selection; however, unless the user knows what specific characteristics to look for, the exploration parameter space is often too extensive to effectively search through. Questions in performance analysis often require the user to identify anomalous subsets of data within the entire dataset, such as periods of poor load balance. Such subsets may be as small as 100 samples within sets of several thousand. It is obviously unfeasible for the user to select every window of 100 samples within the dataset.

Various efforts have made use of data mining and analytics to reduce this search space, for example Voinea and Telea use clustering to highlight similar data elements [29]. We propose using problem-driven analytics to focus the interactive search space towards possible memory-related performance problems. We automatically score subsets of data based on their similarity to known performance characteristics and present these scores visually to the user. The user can make selections based on these scores and analyze the resulting visualizations.

We first describe the metrics as functions of sample subsets, and then we describe how we choose those inputs.

7.1 Proposed Scoring Metrics

From collaboration with domain experts and from previous surveys in the field of performance visualization [15], we determined a set of behaviors that performance analysts typically search for. We defined expressions to calculate the relationship between a subset of samples and each behavior. We target two performance behaviors: **average access latency** and **load imbalance**.

Our proposed metrics take as input the accumulated memory access cycles on the hardware topology for a subset of samples, as described in Section 6, and produce a single value as output. In order for differently sized subsets to be compared to one another, the metric must have the property that it is invariant to the number of samples in the subset.

In the following equations 1 and 2, we define depth in the hardware topology as d, hardware resources at a specified depth as a vector $\hat{r}_d$, and specific resources as $r_{d,i}$. The value $c(r_{d,i})$ represents the number of memory access cycles associated with a particular resource, and $s(r_{d,i})$ represents the number of memory accesses.

Average Access Latency. In practice, the number of cycles taken to access a cache or main memory resources is highly variable due to hardware effects. For example, to maintain cache coherence, the hardware has to keep track of which copies of cached data are stale and update them when a core attempts to read them. This action is done opaquely in hardware but significantly contributes to the wait time for accessing data. To provide an indicator of the average access latency achieved in a particular execution, we calculate the average number of cycles per access at depth d, as shown in equation 1:

$$L_d = \frac{1}{|\hat{r}_d|} \sum_{i=1}^{|\hat{r}_d|} \frac{c(r_{d,i})}{s(r_{d,i})} \tag{1}$$

Load Imbalance Across Resources. In general, performance degrades when memory access is less uniform across the available resources. Often, a few resources are highly over- or under-utilized relative to the average utilization. This behavior is indicative of bottlenecks that may prevent an application from achieving peak performance. In terms of the data, we can identify subsets that exhibit this behavior by searching for regions with significant outliers in hardware utilization.

We calculate the imbalance metric for a specified depth in the hardware topology I_d as the maximal difference, in terms of memory accesses, between each resource and the average number of samples for all resources in the depth $\mu_d(s)$. In order for this metric to be size-invariant, we normalize the distance by the standard deviation of the number of samples across resources in the depth $\sigma_d(s)$, as shown in equation 2:

$$I_d = \frac{\max_{i=1}^{|\hat{r}_d|} |s(r_{d,i}) - \mu_d(s)|}{\sigma_d(s)} \tag{2}$$

7.2 Metric Visualization

Given these two scoring metrics, we need to provide input subsets of potential interest. Because we aim to determine how scores vary with respect to each attribute, we provide as input a subset of samples with a bounded range of values along a single attribute. In other words, we select samples within a window of values along the attribute and use that as input. We divide the attribute into N windows, calculate the metric, and visualize their outputs via color-mapped blocks along the attribute's axis, as shown in Figure 7. The user may specify the metric to use as well as the number N of windows to adjust the scope and granularity of the visualization. The resulting visual cue effectively highlights variation of the metric along an attribute, from which the user may decide to select outliers for further analysis.

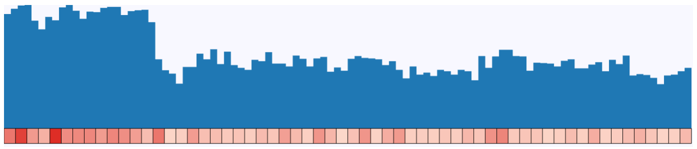


Fig. 7: Histogram and guided interaction along a single attribute. The colored rectangles on the bottom encode derived metrics for bins of samples along the attribute. Darker colors indicate areas of higher imbalance across NUMA memory resources.

7.3 Metric-based Clustering

It is often the case that a range of neighboring bins have similar metric scores. As a result, viewing each of these bins may show redundant results, and, alternately, an individual bin may not contain enough samples to show meaningful correlations. A more effective binning scheme would group together all samples that have similar properties and only present subsets with unique behaviors.

For our purposes, we require one that is scalable, both computationally and visually, and one that makes it possible to analyze how metrics vary with respect to each attribute. We therefore use the constraint that a cluster of samples must be a range of consecutive samples along an attribute. By using this approach, each subset only needs to be compared to its neighbors along a single attribute, and the resulting clusters represent different positions and ranges along the attribute. Furthermore, we want to provide a clustering solution that is dynamic relative to different datasets and adjustable in granularity. For these reasons, we use agglomerative clustering, using as input a set of sliding windows along the attribute.

We generate the initial subsets by moving a sliding window along a specified attribute. We create a window for the smallest w samples along the attribute, store the subset of w samples as a leaf, and slide the window by $\Delta < w$ samples forward until we reach the end of the attribute. For N samples, this method creates approximately N/Δ leaves, each containing w samples. The process is illustrated in Figure 8.

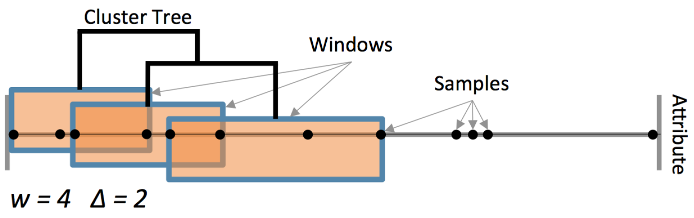


Fig. 8: The process for clustering memory access samples. We create a window with w samples starting with the minimum value along the attribute and move it by an amount $\Delta < w$ samples until we reach the end, creating a subset for each set of w samples within each window position. We agglomeratively cluster the subsets using our proposed metrics in Section 7 as distance metrics.

This approach has a number of advantages. Individual samples are insufficient to characterize performance behavior, so clustering based on individual samples is unpredictable and leads to undesirable results. This method ensures that all leaves have w samples, which is a controllable parameter. By creating windows along an attribute, we create subsets that define how the performance behavior changes with respect to that attribute, which was a primary goal for our automated analysis method. Further, because the windows overlap, we capture subsets that start and end at any point along the attribute, given that they are within the resolution of Δ, and we can capture variable length behavior patterns.

This method is agnostic to the attribute and therefore can be used to cluster along any attribute. However, the results for non-continuous attributes depends highly upon how they are ordered, thus we enable clustering only for continuous attributes currently.

We control the accuracy, shape and computational complexity for generating the tree by modifying w and Δ. Higher values of w capture longer, more stable behaviors at the loss of identifying outliers, and higher values of Δ reduce the number of leaves and time to create the tree at the loss of resolution for possible begin and end points for performance behaviors.

After creating a set of initial subsets along an attribute, we agglomeratively cluster these subsets. We define the similarity (i.e., distance) between subsets using the derived performance metrics described in Section 7. Specifically, we use the absolute difference between their metric values. When we have M metrics and D attributes, we have $M \times D$ possible cluster trees. We note that the windows often contain different numbers of samples. Because the derived performance metrics are invariant to the number of samples contained in the subset, this is not a problem.

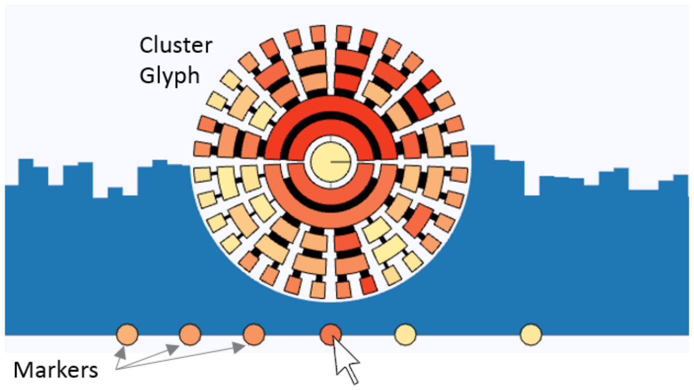


Fig. 9: Cluster viewing and selection. Clusters of a specified depth and metric are shown along the attribute's axis as markers (circles) colored by their derived metrics. When the user hovers the mouse pointer over a marker, a glyph expands showing the hardware topology visualization for the subset of samples contained in the cluster. Upon clicking the marker, the subset is selected for further analysis.

Our agglomerative clustering algorithm iteratively groups together the closest pairs of subsets, with the constraint that subsets may only be grouped together if they are neighbors along the attribute. All resulting clusters represent a range along the attribute, and separation of clusters along an attribute indicates a shift in performance behavior. This clustering method accurately extracts ranges of samples that exhibit a particular performance behavior along an attribute, and by exploring these behaviors, the user can effectively understand how a type of performance behavior varies with respect to a particular variable.

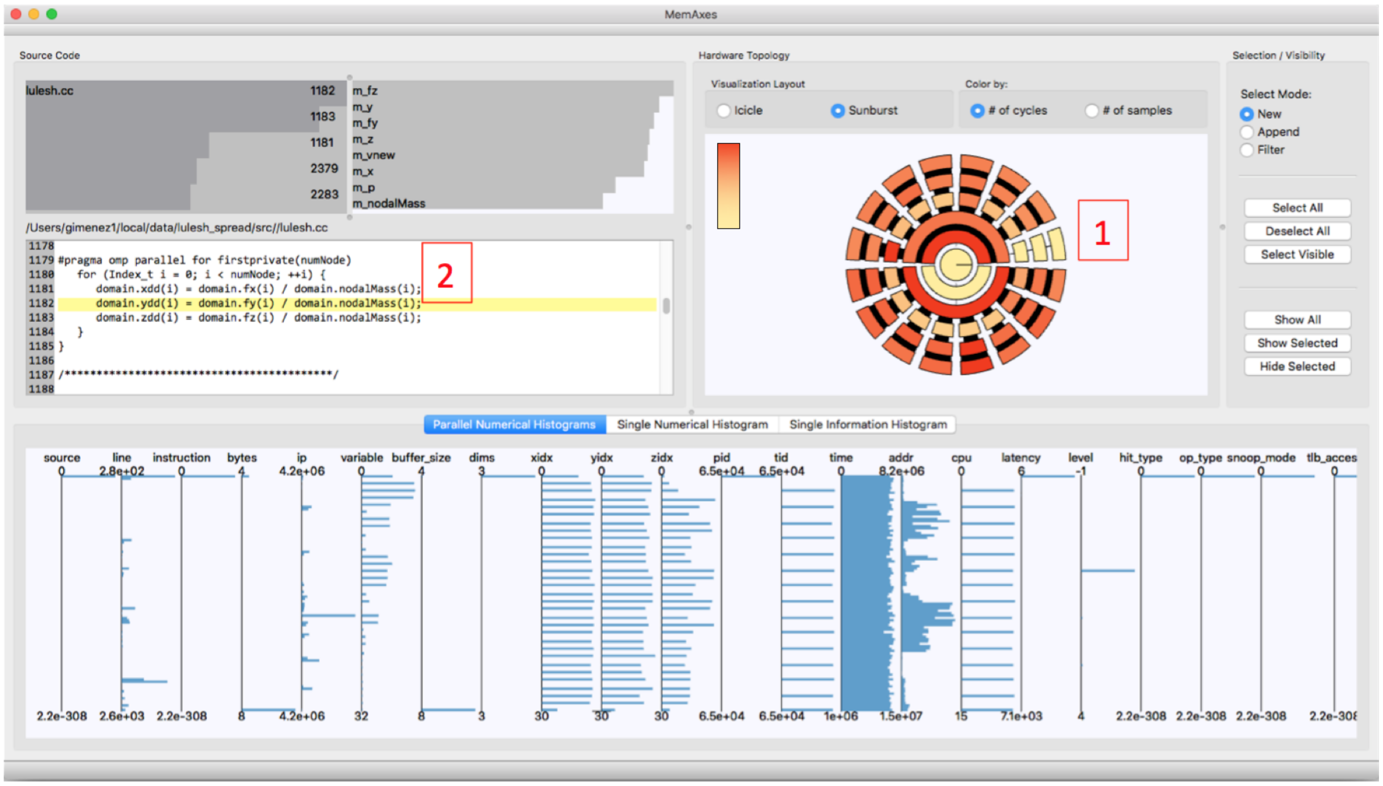
7.4 Viewing and Selecting Clusters

Once the cluster trees are generated, the user may view individual clusters at a glance and select their associated samples for further analysis if desired. The user navigates clusters along the attribute by selecting different metrics and tree depths. After making a selection, the user is presented with markers indicating the location of clusters along the attribute's axis line. The markers are small circles along the axis colored by their metric values. When the user hovers the mouse pointer over a marker, it expands to show a glyph of the hardware topology visualization for its respective sample subsets. Thus, the user can quickly gain an overview of the resulting selection and decide whether to make the selection. By clicking on the marker, the user selects the subset contained in the cluster and all views are updated to show the current selection. The cluster viewing and selection interface is shown in Figure 9.

8 Scalability

Our system requires both visual and computational scalability. Visually, we must ensure that the view is not cluttered, i.e., that we do not display too much information at once. We address this by aggregating data, but in doing so we sacrifice details for clarity. Because the user may select and filter subsets of the samples, s/he may also refine the shown aggregates to recover these details. In this context, guided interaction makes it possible to automatically recover details lost in the aggregates.

Computationally, the algorithms used in MemAxes must scale to large data sets. We minimize the complexity of filtering and



```
#pragma omp parallel for firstprivate(numNode)
   for (Index_t i = 0; i < numNode; ++i) {
      domain.xdd(i) = domain.fx(i) / domain.nodalMass(i);
      domain.ydd(i) = domain.fy(i) / domain.nodalMass(i);
      domain.zdd(i) = domain.fz(i) / domain.nodalMass(i);
   }
}

/*****************************************/
```

Fig. 10: MemAxes upon opening the memory performance data collected from LULESH, before performing any interaction. In the topology view (marked 1) we can see one processor was extremely under-utilized relative to the rest. We also see that several accesses were made to the top NUMA node (the dark orange arc directly above the center) but few if any were made to the bottom one. In the code/data view (marked 2) we see a parallel loop in the code accessing elements `fy` and `nodalMass` from the domain. In the top offenders view directly above, we see more time was spent waiting for `fy` than `nodalMass`.

selection interactions by implementing them as set operations with linear complexity (unions and intersections). The most computationally expensive operation in MemAxes is clustering. We use agglomerative clustering, which is traditionally unsuited to large data sets, as the fastest known algorithms have quadratic, $O(n^2 \log n)$, complexity [20]. However, the quadratic part of this algorithm is distance matrix calculation, and our algorithm only requires distances between neighboring windows along each attribute. We only ever compare a window to its two neighbors along a single dimension, reducing the complexity to $O(kw \log w)$, where w is the number of windows and k is the window size. We can control this complexity by adjusting the size and number of windows.

9 Case Studies

In the following, we present some case studies that demonstrate the workflow when using MemAxes, including initial exploration, data-driven hypothesis forming, and experimental validation. We, the authors, are part of a tightly integrated research organization including visualization researchers, HPC performance analysts, and other domain experts that use performance analysis tools. MemAxes has been actively used within this organization, both in collaboration with us, the authors, and independently. The case studies presented are the products of these various efforts.

All experiments were performed on the Cab cluster [1] at Lawrence Livermore National Laboratory, which features two eight-core Intel Xeon E5-2670 sockets on each node, for a total of 16 cores with L1 and L2 caches, two shared L3 caches, and two NUMA memory banks. We collected memory performance data from two commonly used HPC proxy applications. A proxy application isolates computationally complex code sections from a larger application for the purpose of analyzing their performance in detail. The proxy applications we analyzed are LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [16] and XSBench [27].

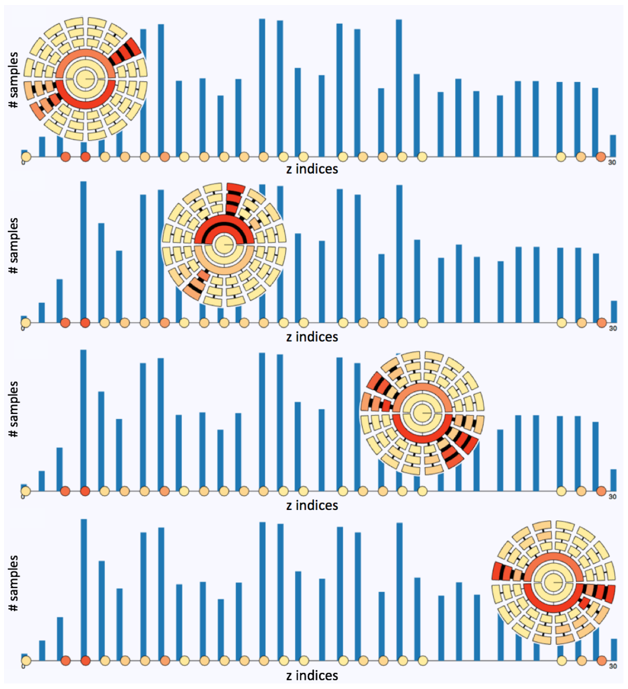
9.1 LULESH

LULESH emulates the behavior of an iterative shock-hydrodynamics simulation involving a mesh that deforms over time. The mesh deformation requires the simulation to retrieve the changing locations of cells and corners in the mesh in order to access their values. Each value retrieval requires multiple memory accesses, and it has been hypothesized that this step is a significant contributor to total computation time.

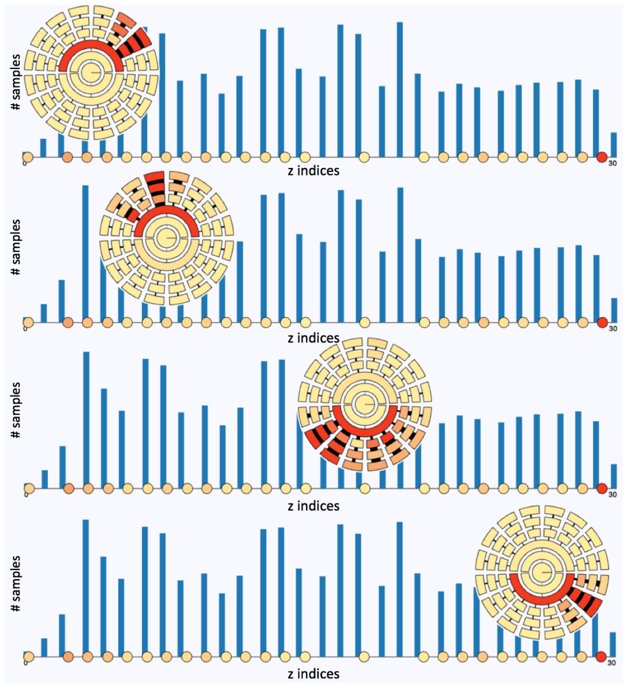
We configured the memory access sampler to record a set of additional attributes from the application – the x-, y-, and z-coordinates in the mesh for each access. These additional attributes make it possible to further discern the relationships between the layout of the data and other characteristics, such as locality in different memory resources.

We sampled the complete execution of LULESH under a set of input parameters provided by its developers and aimed to keep overhead costs at or under 10%. The resulting output contains 235,447 samples with 18 attributes each.

A screenshot of the initial state of MemAxes after loading the collected data, and before any user interaction, is shown in Fig. 10.



(a) Cluster glyphs generated by clustering along the `zidx` axis using processor load imbalance metric. Consecutive z-indices were accessed by distant processors on either side of the hardware topology, causing contention across both processor sockets.



(b) Cluster glyphs generated by clustering along the `zidx` axis using processor load imbalance metric, after mapping consecutive z-indices to neighboring processor cores. We can see that consecutive z-indices were accessed by neighboring processors, meaning cross-socket data contention was effectively reduced. We observe a 60% decrease in memory access cycles and 10% decrease in total execution time.

The views depict all collected sample points, and we can gain some initial insights into overall execution. In the topology view, marked with a red number 1, we see that one of the processors appears to be extremely under-utilized with respect to the other processors. In addition, the first NUMA node (the top innermost arc) is associated with many accesses and transactions, based on its color and link thickness, while the other appears to have been used sparingly, if at all. We presented these initial observations to the developers of LULESH and they confirmed that (1) inter-node communication was handled by the message-passing library, with this particular library implementation reserving a single core for communication operations, and (2) intra-node parallelism in this version of LULESH was achieved by distributing work to all processors on a node, with allocation done only on the first NUMA node, leaving few accesses to be made to the second. This valuable developer feedback validated our data collection and aggregation methods.

The code and data views (marked with a red number 2) revealed the most memory-intensive section of code in LULESH. Concerning the source code, we saw a loop preceded by a parallel directive, indicating a parallel region. Within this region, one access had been made to each of the variables `fx`, `fy`, `fz`, and three accesses had been made to `nodalMass`. However, in the sorted histogram of variable contributions (emphasizing top-offending variables) for this particular region of code we saw that the accesses to each of the other variables had actually been more expensive overall, although more accesses had been made to `nodalMass`. In order to further study this anomaly, we selected each of these data objects and examined the selections in the different views. This analysis revealed that `nodalMass` was rarely accessed by the L3 cache (only 16 times), and those accesses cost an average of 405 cycles. The other data objects, `fx`, `fy`, and `fz`, were only slightly more frequently accessed by L3, with average costs of between 1100 and 2700 cycles. However, these same data objects incurred few accesses by the L2 cache. These observations led us to hypothesize that the data objects were sufficiently small enough to fit into the L1 cache, and that accesses to the shared L3 cache occurred due to contention between multiple processors.

In order to further investigate the hypothesis that the `fx`, `fy`, and `fz` data objects were involved in shared L3 cache contention, we considered the load imbalance metric across processor resources. By clustering based on processor imbalance across different attributes of the data and examining the resulting cluster glyphs, we were able to identify an undesirable data decomposition pattern, shown in Fig. 11a. Along the `zidx` attribute, consecutive elements had apparently been accessed by processor cores residing at opposite ends of the hardware topology, i.e., by distant cores across both available sockets. This behavior is highly inefficient—neighboring data elements in z-direction were consequently contended across the entire topology and required constant data migration to maintain cache coherence.

We designed an experiment to resolve this observed inefficiency and collected memory access samples again. By changing the mapping of indices such that consecutive indices would be processed by neighboring processor cores, we hypothesized that remote accesses would be reduced and performance should be improved. We performed the experiment and, again, performed clustering based on processor imbalance across z-indices to verify that we had successfully reduced cross-socket data contention, see

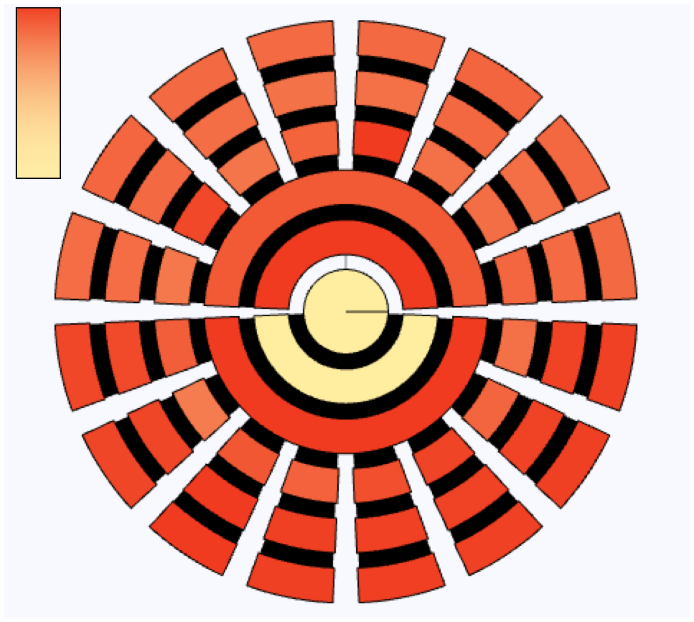


Fig. 12: Hardware topology view of sampled memory accesses of the XSBench application execution. Resource utilization was mostly uniformly distributed, with the exception of NUMA resources, where all accesses occurred in the first (top) NUMA resource.

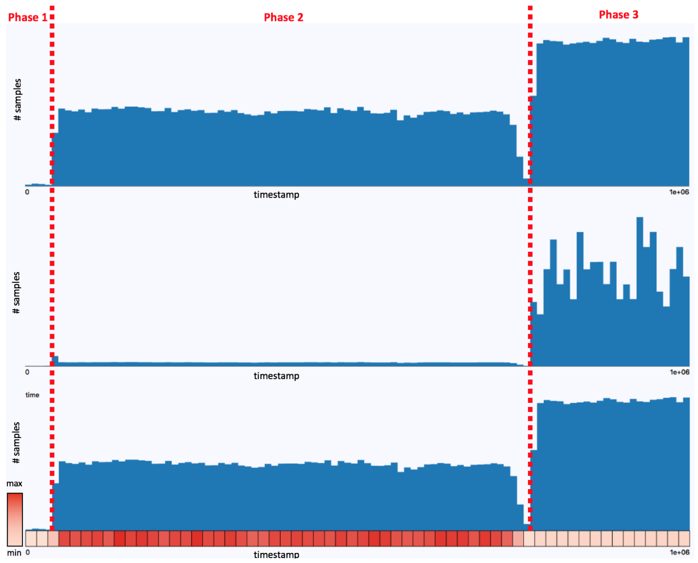


Fig. 13: Top: single histogram of `time` attribute, where three time sections can be seen, indicating different execution phases (marked). Middle: same attribute after selecting accesses to NUMA resources; nearly all NUMA accesses occurred during the third phase. Bottom: same attribute with L2 imbalance metric shown along the axis. It can be seen and concluded that the second phase caused the most L2 cache imbalance, relatively.

Fig. 11b. Most importantly, we observed a decrease in total latency by 60% and a decrease in total execution time by 10%. This result represents a significant performance improvement in terms of both time and energy consumption. (HPC applications typically consume on the order of several hundred joules per second per compute node, and applications can run for several hours using several thousand compute nodes.)

9.2 XSBench

XSBench is a proxy application for OpenMC, a Monte-Carlo simulation for calculating particle interactions. Due to its underlying non-deterministic algorithm, its memory accesses follow a less predictable pattern and thus pose a difficult performance analysis problem.

Again, we sampled the complete execution of XSBench for a set of provided input parameters, configuring sampling to maintain an overhead of 10% or lower. The resulting output contains 175,430 samples with 13 attributes each.

The initial topology view for the XSBench data shows fairly uniform utilization across all resources, except across NUMA domains, see Fig. 12 Since the first NUMA domain incurred 282 sampled accesses, while the second incurred none, we hypothesized that this application had allocated data only in one of the two available NUMA domains. This hypothesis was confirmed by the developers.

We investigated these NUMA accesses further by selecting them and observing the other attributes in the parallel histograms view. A clear correlation emerged with the `time` attribute, with three sections of time showing different distributions of sampled accesses and nearly all NUMA accesses occurring during the third one. We also examined different metrics of imbalance across this axis and observed that the second section of time incurred high amounts of L2 imbalance. These observations led us to hypothesize that the XSBench application underwent three separate phases over time, the first of which having relatively few memory accesses, the second accessing data objects too large to fit into the L1 cache, and the third causing several NUMA accesses, see Fig. 13.

We verified these hypotheses once again with the developers of XSBench. The developers confirmed that the application involved three main phases, the first being relatively inexpensive in terms of memory access, the second involving a sorting step of a large data set, and the third undergoing the non-deterministic memory accesses across the same data object by multiple processor cores. The last phase caused slow accesses by large resources as the hardware failed to predict and prefetch data elements for computation.

10 Conclusions and Future Work

MemAxes is a novel tool for the analysis of memory performance behaviors in high-performance computing environments. By creating multiple aggregation-based visualizations of a central dataset, MemAxes provides the capability to understand the complex relationships between the hardware, the data, and the code. It introduces a new visualization for data transactions in a multi-level cache hierarchy that is capable of exposing aspects of load balance, data decomposition, and parallel access patterns that were previously difficult to discover. MemAxes also explores the concept of guided interaction via visual cues and automatic cluster extraction based on scoring subsets of the data. We were able to successfully utilize MemAxes to analyze the complex memory behaviors of two memory-intensive proxy applications, and we were able to significantly improve the performance of one of them using insights from this analysis, saving time and energy.

While we proposed two metrics for scoring sample sets, we believe that much can be done to create more advanced metrics.

Furthermore, although the method of scoring is highly application-dependent, the idea to use scores to create visual cues and describe similarity between subsets could be extended to other applications. In particular, scoring would enable many visualization and data processing techniques for high-dimensional or non-quantitative data that has otherwise ill-defined distance metrics or similarities.

We found that the clustering algorithm we used was both justified for this application and practically effective. However, for different data or scoring functions, many generalizations and specializations are possible. The combination space of possible initial clusters for agglomerative clustering, beyond using sliding windows along an attribute, is unfeasibly large, and effectively exploring this space remains a challenge.

MemAxes successfully addresses the problem of analyzing on-node memory performance, but does not trivially extend to analyzing multiple nodes. We believe many of the ideas we present may provide the foundations for multi-node performance analysis, in particular computing metrics and clustering to find specific nodes with outlying performance patterns.

Analysis of memory performance behavior continues to be a challenging topic, as memory hierarchies continue to increase in complexity. New processors have complex on-node interconnection networks, and future hardware poses an interesting challenge in terms of scale and extensibility. Further, processors continue to evolve, and our techniques will need to be extended to provide insight into richer data from future performance monitoring units.

Acknowledgments

We thank John Tramm and Tanzima Islam for their expert feedback and help in analysis of XSBench. We also thank Barry Rountree for his help in diagnosing memory behaviors.

This research was supported by Lawrence Livermore National Laboratory (LLNL) and the University of California (UC) at Davis through its UC Laboratory Fees Research Grant Program.

References

[1] Cab: Intel Xeon system in Livermore Computing. http://computation.llnl.gov/computers/cab.

[2] Instruments user guide. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/. Accessed: 2017-05-31.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[4] B. Alpern, L. Carter, and T. Selker. Visualizing computer memory architectures. In *VIS 1990*, pages 107–113. IEEE Comput. Soc. Press, 1990.

[5] Bosch, R.P. and Stanford University Computer Science Dept. *Using visualization to understand the behavior of computer systems*. Stanford University, 2001.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. Hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[7] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, and J. Nystrom-Persson. Visualising dynamic memory allocators. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 115–125, New York, NY, USA, 2006. ACM.

[8] A. N. M. I. Choudhury and P. Rosen. Abstract visualization of runtime memory behavior. *VISSOFT 2011*, pages 1–8, 2011.

[9] N. Denoyelle, B. Goglin, and E. Jeannot. A Topology-Aware Performance Monitoring Tool for Shared Resource Management in Multicore Systems. In Springer, editor, *Proceedings of Euro-Par 2015: Parallel Processing Workshops*, Lecture Notes in Computer Science, Vienna, Austria, Aug. 2015.

[10] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann. Dissecting on-node memory access performance: a semantic approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 166–176. IEEE, 2014.

[11] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, Apr. 2004.

[12] R. Griswold and G. Townsend. *The Visualization of Dynamic Memory Management in the Icon Programming Language*. TR. Department of Computer Science. University of Arizona. University of Arizona, Department of Computer Science, 1989.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. A Quantitative Approach. Elsevier, 2012.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[15] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In *Proceedings of the Joint Eurographics-IEEE VGTC Symposium of Visualization 2014 (EuroVis 2014), State-of-the-Art Reports*, 2014.

[16] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[17] M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*, pages 213–232. Springer, 2003.

[18] J. B. Kruskal and J. M. Landwehr. Icicle plot: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[19] X. Liu and J. Mellor-Crummey. A data-centric profiler for parallel programs. In *SC 2013*, pages 1–12, New York, New York, USA, Nov. 2013. ACM Request Permissions.

[20] O. Maimon and L. Rokach. *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[21] S. Moreta and A. Telea. Visualizing Dynamic Memory Allocations. *VISSOFT 2007*, pages 31–38, 2007.

[22] H. Prokop. Cache-oblivious algorithms, 1999.

[23] J. Reinders. *VTune performance analyzer essentials*, volume 14. Intel Press, 2005.

[24] P. Rosen. A visual approach to investigating shared and global memory behavior of cuda kernels. 32(3pt2):161–170, 2013.

[25] M. Schulz, J. A. Levine, P. T. Bremer, T. Gamblin, and V. Pascucci. Interpreting Performance Data across Intuitive Domains. In *ICPP 2011*, pages 206–215. IEEE, 2011.

[26] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. *InfoVis 2000*, pages 57–65, 2000.

[27] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[28] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *ICPPW 2010*, pages 207–216, Sept 2010.

[29] L. Voinea and A. Telea. Visual data mining and analysis of software repositories. *Computers & Graphics*, 31(3):410–428, 2007.

[30] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

Alfredo Giménez is a Ph.D. candidate at the University of California, Davis focusing on scalable visualization and analysis techniques for performance data in HPC environments. He received a B.S. in computer science from UC Davis in 2010, worked at Intel Corporation in 2010-2012 developing tools for debugging and analyzing general-purpose GPU code, and is currently an active collaborator at Lawrence Livermore National Laboratory since 2012.

Todd Gamblin is a Computer Scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research focuses on scalable tools and algorithms for measuring, analyzing, and visualizing the performance of massively parallel applications. He leads several projects in these areas, and has been at LLNL since 2008. Todd received the Ph.D. and M.S. degrees in Computer Science from the University of North Carolina at Chapel Hill in 2009 and 2005. He received his B.A. in Computer Science and Japanese from Williams College in 2002. He has also worked as a software developer in Tokyo and held graduate research internships at the University of Tokyo and IBM Research. Todd recently received an Early Career Research Award from the U.S. Department of Energy.

Ilir Jusufi is a Senior Lecturer in the department of Media Technology at Linnaeus University, Växjö, Sweden. He was a Postdoctoral Scholar at the University of California, Davis. He received a B.S. in Computer Science at the South East European University in Macedonia and a M.S. in Computer Science at the Växjö University in Sweden. He earned his Ph.D. degree at the Linnaeus University in Sweden focusing on the visualization and interaction techniques of multivariate networks.

Abhinav Bhatele is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His interests lie in performance optimizations through analysis, visualization and tuning and developing algorithms for high-end parallel systems. His thesis was on topology aware task mapping and distributed load balancing for parallel applications.

Abhinav received a B. Tech. degree in Computer Science and Engineering from I.I.T. Kanpur, India in May 2005 and M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 2007 and 2010 respectively. Abhinav was an ACM/IEEE-CS George Michael Memorial HPC Fellow in 2009. He has received several awards for his dissertation work including the David J. Kuck Outstanding MS Thesis Award in 2009, a Distinguished Paper Award at Euro-Par 2009 and the David J. Kuck Outstanding Ph.D. Thesis Award in 2011. Recently, a paper that he co-authored with LLNL and external collaborators was selected for a best paper award at IPDPS in 2013.

Martin Schulz is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). He earned his Doctorate in Computer Science in 2001 from the Technische Universität München (Munich, Germany) and also holds a Master of Science in Computer Science from the University of Illinois at Urbana Champaign. He has published over 175 peer-reviewed papers and currently serves as the chair of the MPI forum, the standardization body for the Message Passing Interface. He is the PI for the Office of Science X-Stack project ”Performance Insights for Programmers and Exascale Runtimes” (PIPER) as well as for the ASC/CCE project on Open|SpeedShop, and is involved in the DOE/Office of Science exascale projects CESAR, ExMatEx, and ARGO . Martin's research interests include parallel and distributed architectures and applications; performance monitoring, modeling and analysis; memory system optimization; parallel programming paradigms; tool support for parallel programming; power-aware parallel computing; and fault tolerance at the application and system level. Martin was a recipient of the IEEE/ACM Gordon Bell Award in 2006 and an R&D 100 award in 2011.

Peer-Timo Bremer is a member of technical staff and project leader at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL) and Associated Director for Research at the Center for Extreme Data Management, Analysis, and Visualization at the University of Utah. His research interests include large scale data analysis, performance analysis and visualization and he recently co-organized a Dagstuhl Perspectives workshop on integrating performance analysis and visualization. Prior to his tenure at CASC, he was a postdoctoral research associate at the University of Illinois, Urbana-Champaign. Peer-Timo earned a Ph.D. in Computer science at the University of California, Davis in 2004 and a Diploma in Mathematics and Computer Science from the Leibniz University in Hannover, Germany in 2000. He is a member of the IEEE Computer Society and ACM.

Bernd Hamann is a professor of computer science at the University of California, Davis. He studied mathematics and computer science at the Technical University of Braunschweig, Germany, and received a Ph.D. in computer science from Arizona State University in 1991. His main teaching and research interests are data analysis and visualization, image processing, and geometric design and modeling.