Visualization of Data Movements and Accesses

Seminar Thesis
Til Mohr

Chair for High Performance Computing, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Supervisor: Isa Thärigen, M.Sc.

The widening processor-memory performance gap and the increasing complexity of programs necessitate better data locality optimization methods for efficient computation. This paper presents a comprehensive overview of visualization techniques for data movements and accesses to aid in data locality optimization. It includes methods of data gathering like dynamic analysis, static analysis, and simulation, and discusses their usage in various visualization tools at different granularities. Three specific tools are detailed, providing unique perspectives on data movement visualization. The paper further outlines the standard performance optimization workflow and provides an outlook on future enhancements in data gathering, visualizations, and automated program optimization.

Keywords: Data Locality, Memory Access Visualization, Dynamic Analysis, Static Analysis, Simulation, Performance Optimization, High-Performance Computing, Data Movement

1 Introduction

The pursuit of performance optimization in the field of high-performance computing (HPC) continues to push boundaries, with significant emphasis being placed on mitigating the impact of the increasing processor-memory speed gap and the rising computational memory requirements. These challenges are amplified by the escalating complexity of modern programs, making it increasingly difficult for experts to form a mental model of a program's data movement, let alone domain researchers. These factors have led to a marked surge in the costs of data movement and the appearance of severe performance bottlenecks. While advancements in hardware can alleviate some of these issues, the software community must also step up to the challenge, enhancing data locality through software to optimize data movement and access.

In this context, this paper focuses on the visualization of data movements and accesses, an often overlooked yet critical aspect of understanding and optimizing the complex data behavior of modern programs. Through a detailed overview of various methods of data acquisition, including dynamic analysis, static analysis, and cache simulation, this paper aims to shed light on the intricate world of data movement. By discussing different visualizations at varying granularities, it seeks to arm performance engineers with the necessary tools to enhance a program's data locality.

This contribution stands out as it provides a consolidated overview of different data visualization methods, enabling practitioners to select and employ the most suitable ones based on their specific needs and the complexity of their programs. This overview is not limited to any single approach, but instead offers a comprehensive understanding of the methods available, highlighting the strengths and limitations of each.

The rest of the paper is structured as follows: Section 2 discusses the prevailing memory-related performance problems and their implications for modern computing systems. In Section 3, we delve into the various methods of acquiring memory-related performance data, with a focus on dynamic analysis, static analysis, and cache simulation. Section 4 provides a comprehensive overview of different visualization techniques used to interpret this data, while Section 5 outlines the standard workflow adopted by performance engineers to identify and mitigate memory-related bottlenecks. Section 6 presents an in-depth examination of exemplary memory access visualization tools, highlighting their unique strengths, weaknesses, and data gathering methods. Finally, the paper concludes with a discussion on the outlook for future work and potential improvements

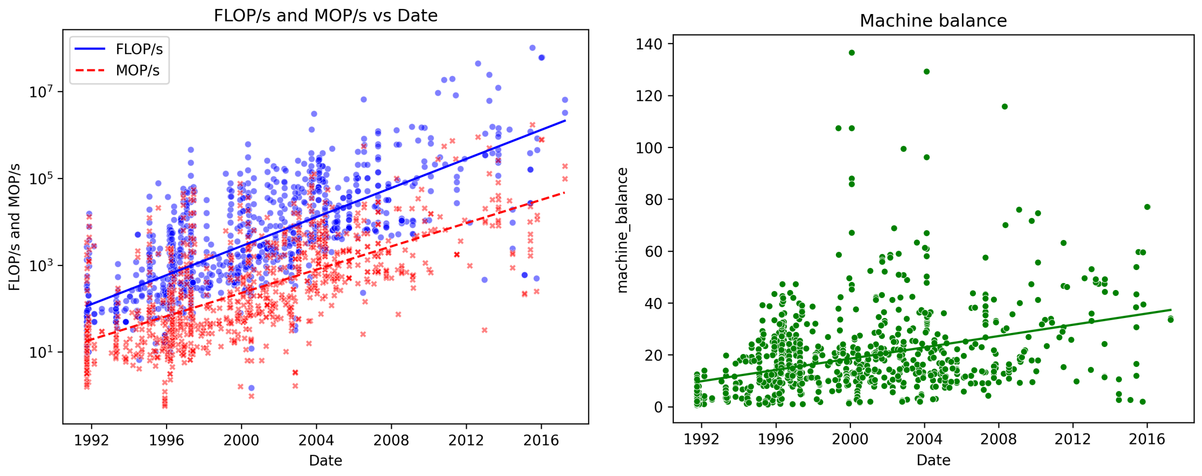


Figure 1: Illustration of the expanding processor-memory gap. The left graph charts the progression of FLOPs and MOPs on a logarithmic scale across various computing platforms, with the FLOPs trendline demonstrating a steeper ascent, indicative of the widening gap. The right figure depicts the development of the machine balance score for these platforms.[1]

in this vital and rapidly-evolving field.

2 Memory-Related Performance Problems

As modern computing systems evolve, the demand for increased computational power and memory resources has become more prevalent. This demand is driven by the increasing complexity of applications and the need to process larger amounts of data. In this section, we will explore the challenges and performance problems that result from the ever-growing requirements for memory and computational resources. We begin by discussing the processor-memory performance gap and its implications in Section 2.1, followed by a brief examination of the increasing computational and memory requirements of modern applications (Section 2.2). The processor-memory performance gap and the increasing computational and memory requirements combined result in a need to tackle high data transfer costs and bottlenecks (Section 2.3). Finally, we will define the concept of data locality in Section 2.4, which solutions to the aforementioned problems take into consideration.

2.1 Processor-Memory Performance Gap

It is well known, that the performance of CPUs doubles roughly every two years, a phenomenon resulting from Moore's law. Similarly, memory technology has also been progressing exponentially, however, at a slower pace [1–4]. Since the difference between two exponential functions is also exponential, this gap will expand rapidly. This concept is known as the processor-memory performance gap. Figure 1 illustrates this trend in improvements in computational and memory performance, measured by floating point operations and memory operations per second, respectively.

The increasing processor-memory performance gap becomes a critical problem when considering data access times. Take the equation for the average memory access time:

$$t_{avg} = p \cdot t_c + (1 - p) \cdot t_m \tag{1}$$

Here, $p \in [0, 1)$ denotes the probability of a cache hit (Section 2.3). As at least one instruction has to be fetched from memory, at least one cache miss per application is guaranteed, thus $p < 1$. t_c and t_m denote the times to access data from a cache and the main memory, respectively [5, 6]. These times measure the performance of the cache and the main memory as a combination of memory latency and bandwidth (Section 2.3). To simplify our understanding, let's assume that the CPU clock speed is constant. Under this assumption, we can interpret these memory access times as corresponding to a specific number of clock cycles.

Interestingly, over the past decade, the improvement of CPU clock times has seen a stagnation [7]. This suggests that with the progress in memory technology, the times required to access a given amount of data, as represented by t_m and t_c, are decreasing.

[1]Data was acquired from a collection of STREAM benchmark results on `https://www.cs.virginia.edu/stream/`.

However, other processing-centric innovations such as hyperthreading and multicore CPUs [7] continue to outpace these memory performance gains. As a result, the overall system performance will be increasingly determined by memory performance. At some point, CPUs would be able to execute code faster than we can feed them with instructions and data. For this reason, the processor-memory performance gap is also known as the memory wall problem [3, 5, 6].

To quantify the processor-memory performance gap, the notion of machine balance has been introduced [4, 8]:

$$balance = \frac{peak\ FLOP/s}{sustained\ MOP/s} \tag{2}$$

This metric, also depicted in Figure 1, is a measure of how well a system is balanced between computational and memory performance. A balance of 1 indicates a perfectly balanced system, whereas $balance \ll 1$ or $balance \gg 1$ indicates a system that is entirely compute or memory-bound, respectively [4, 8].

2.2 Computation and Memory Requirements

Different applications have different requirements for system resources. There exist some programs that have a larger computational demand, thus benefiting from a higher machine balance [8]. However, the memory wall problem states that it will be increasingly more difficult for such applications to exploit further advances in computational performance, as for any application the memory performance will grow to be the limiting factor [3, 5].

Furthermore, we notice that computational as well as memory requirements are increasing rapidly. A prime example of this is the field of artificial intelligence systems, which currently sees exponential growth in the number of parameters used [9]. Hence, for any application, regardless of its computational or memory demands, significant strides must be made in enhancing both the processing and memory capabilities. This ensures that the constraints imposed by the memory wall problem do not inhibit the potential performance of these applications.

2.3 Data Transfer Costs and Bottlenecks

In this section, we delve into the different elements that contribute to and potentially mitigate the processor-memory performance gap. Understanding these factors can assist in strategizing efficient and effective solutions to address this pervasive issue.

Memory Latency Memory latency pertains to the time delay between a request to access data from the main memory and the start of the execution of this operation. Increased memory latencies, measured in clock cycles, lead to the processor waiting longer for data, significantly tightening the performance bottleneck. This latency challenge can adversely impact the execution of applications, and its reduction is often a complex task [2].

Memory Bandwidth Memory bandwidth denotes the volume of data transfer to or from memory per unit of time. A bottleneck arises when the bandwidth is insufficient to handle the required data transfer volume, causing the processor to wait for data [2].

Cache Misses To alleviate the impact of memory latencies, a multi-tiered memory hierarchy has been implemented in modern computing systems. This hierarchy includes the use of caches, which are smaller, faster, and more expensive memory modules placed between the CPU and main memory. When the processor needs to access data, it first checks if the data is already in the cache, a situation known as a cache hit. However, if the data is not in the cache, the processor has to retrieve it from the slower main memory, a process known as a cache miss [1, 2, 6]. When a cache miss occurs, an entire block of memory known as a cache line is loaded into the cache. The cache line includes the requested data and some adjacent memory locations. However, this process of retrieving data from the main memory takes considerably more time due to the higher latency of the main memory. Therefore, cache management, handled by software, is vital to maintain optimal performance. Improper management can lead to an increase in cache misses, thereby significantly degrading the system's performance.

2.4 Data Locality

Data locality is a key concept in enhancing memory performance and therefore reducing the implications of the processor-memory performance gap. It refers to the tendency of a processor to access the same set of memory locations, or closely stored memory locations, repetitively over a short period. This concept capitalizes on the multi-level memory hierarchy of modern computers: By improving data locality, one reduces the number of cache misses (Section 2.3), i.e., decreases p in Equation 1, improving overall system performance [10, 11].

There are two main types of data locality: temporal and spatial locality. Temporal locality involves reusing the same data within a relatively small duration. This means that if a memory location is accessed, it is probable that the same location will be accessed again soon. Spatial locality, on the other hand, refers to the use of data items stored in proximity. In other words, if a memory location is accessed, memory locations nearby will likely be accessed shortly [12].

Data layout plays a significant role in the realization of data locality, as it can substantially influence the memory access patterns and hence, the underlying performance of a program. A thoughtful arrangement of data in memory can encourage both temporal and spatial locality, thus reducing cache misses and enhancing the overall system performance [12].

To illustrate, consider a two-dimensional array laid out in memory, where elements in the same row are stored in consecutive memory locations. If an application iterates through this array row by row, it benefits from spatial locality, as loading one element of the matrix also loads the few next elements in the row into the cache due to loading of entire cache lines, thus reducing the number of cache misses. On the contrary, if the application were to traverse the array column by column, it would not benefit from spatial locality due to the dispersed memory locations of elements in the same column, leading to a higher rate of cache misses and reduced performance.

While this example demonstrates a simple scenario, the reality is often more complex, especially for larger and more intricate applications. Understanding the data access patterns of an application is key to deciding the best data layout, and this often requires an intimate knowledge of the program's structure. Moreover, optimizing data locality can be quite challenging due to the diversity of hardware architectures. The same program can exhibit different data locality characteristics on different hardware due to variations in the memory hierarchy, such as differences in cache sizes and levels, memory bandwidth, and latency.

This paper will provide an overview of an approach to optimize data locality with the help of visualizations.

3 Data Gathering Approaches

In the pursuit of optimizing a program's data locality, implementing visual aids to represent data movements and data layouts can be particularly helpful. This approach enables quick and effective identification of data-related issues, their comprehension, and ultimately, their resolution. This method empowers not only program optimization experts but also domain researchers to effortlessly optimize their programs.

To enable such effective visualization, however, it is essential to first collect information regarding data locality. Several studies have explored this area, leading to the identification of three primary strategies: dynamic analysis, static analysis, and simulation. These strategies, which we will delve into in Sections 3.1, 3.2, and 3.3, each bring their unique benefits and drawbacks. Furthermore, it is important to note that some techniques used for gathering data locality information may not be confined to just one of these three fundamental categories, and could instead exhibit characteristics of multiple approaches.

Once data locality information is gathered, it needs to be presented in a user-friendly manner. There exists a wide variety of visualization techniques that can fulfill this requirement, some of which we will detail in Section 4.

Finally, in Section 5, we will provide a brief overview of the standard procedure a performance engineer employs to pinpoint memory-related bottlenecks and subsequently enhance the program's data locality.

3.1 Dynamic Analysis

Dynamic analysis involves examining a program's data locality by running the program and simultaneously collecting relevant memory-oriented data and statistics. These techniques are widely utilized not only for memory performance analysis, but also to gain a comprehensive understanding of a program's overall performance. Hardware counters, special-purpose registers built into CPUs, are commonly used to measure diverse aspects of a program's execution including the number of cache misses, the number of instructions executed, and the number of floating-point operations performed.

Nevertheless, for effective performance analysis, it is essential to pinpoint the exact location in the source code where bottlenecks occur, such as specific lines of code or function calls. In the absence of this contextual information, discerning the root cause of a performance issue can be challenging. Thus, simply monitoring hardware counters during the program's execution is insufficient. It is equally crucial to track the program's execution flow, so that the hardware counter data can be tied back to the source code. This can, for example, be achieved through the instrumentation of the program's source code with additional instructions that record the program's execution and store performance-related data.

There exist several prominent techniques for dynamic analysis of a program's data locality. Profiling works by capturing both hardware-derived attributes and context-related information for specified regions of a program and aggregating the results into a single report per region [13–15]. Profiling techniques analyze the program's call stack and program counter to provide specific details such as the current line of code being executed, the symbol, and, for arrays, the accessed index. This information facilitates the derivation of deeper metrics, such as the number of cache misses per array [15].

Tracing, another technique, allows for a temporal understanding of a program's behavior by logging event-specific data over time. Tracing functions by documenting specific events or functions during program execution, providing a chronological account of these events and their corresponding data. This timeline view of the data contrasts the result that profiling produces, where data is aggregated per instrumented region [15–17].

Profiling and tracing can be implemented through source code instrumentation, recording all pertinent memory accesses. In this way, all relevant memory accesses are recorded. An alternative to instrumentation is statistical sampling, capturing the program's state at regular intervals instead of event-triggered measurements. The advantage of statistical sampling over instrumentation is that it avoids frequent interruption of the program's execution, reducing the runtime overhead. However, high-quality measurements require sufficiently high sampling rates to capture all relevant details [15]. Hence, a trade-off between the granularity and the quality of the measurements is necessary.

In conclusion, dynamic analysis offers several distinct advantages in the study of a program's behavior concerning data locality. As the program is being executed, it offers precise practical insights into hardware oriented data locality optimization. Further, dynamic analysis can be employed in conjunction with actual data, making it more representative of real-world scenarios.

However, it is important to note the inherent disadvantages of dynamic analysis. The act of running an entire program can be time-consuming and costly, particularly for larger and more complex software. In addition, dissecting specific parts of a program, isolated from the rest, can be rather complex, if not impossible with dynamic analysis. In such cases, other techniques such as static analysis may be more applicable.

```
void square(double &v) {
    v = v * v;
}

// ...

square(A[i]);
```

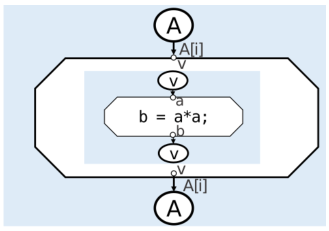


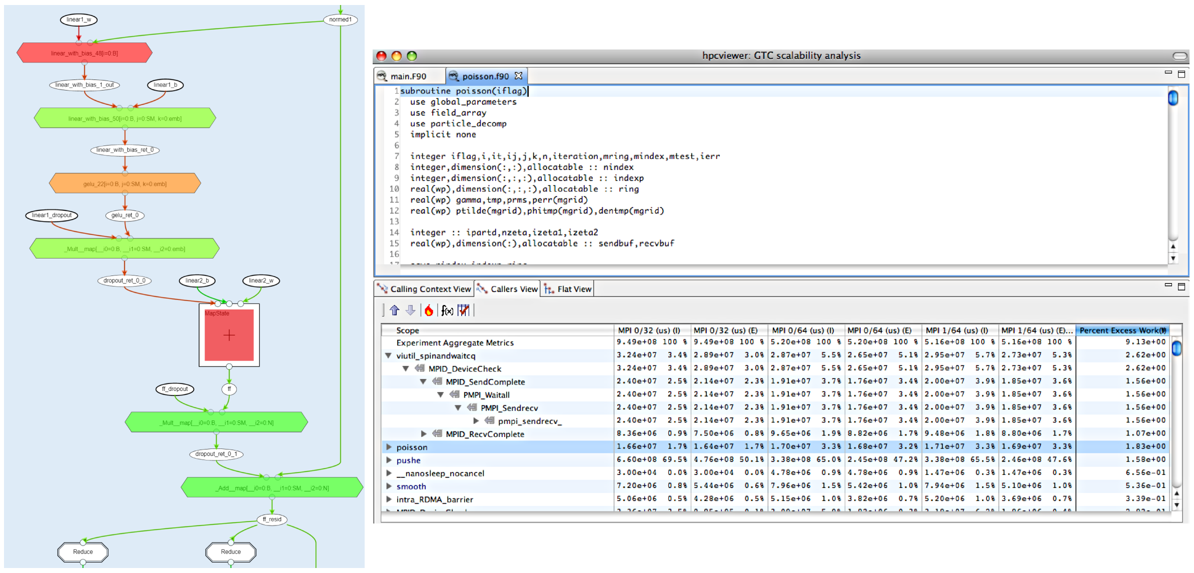
Figure 2: C++ language source code and its corresponding SDFG representation [20].

3.2 Static Analysis

Unlike dynamic analysis, static analysis takes a different tack in examining a program's data locality. Instead of operating the program in real-time to gather data, static analysis scrutinizes the program's source code itself. By transforming the source code into an intermediate representation (IR) that centers on data, and subsequently analyzing this IR, static analysis is able to uncover memory-related issues [18–21].

There are myriad of IRs in use, like MLIR [22], which are predominantly control-flow oriented, facilitating optimizations pivoting around control elements like loop restructuring [23]. However, in the context of data locality, data-centric IRs such as SDFG [24], PROGRAPH [25], and LabVIEW [26] provide a more direct approach. By prioritizing memory, its movements, and its computation-induced alterations, these IRs allow for both automated [24] and, when paired with visual aids, manual enhancements of data locality [19, 21, 24].

Taking the example of SDFGs, the entire data flow of a program can be represented as a directed graph. Nodes within this graph symbolize N-dimensional arrays of data, computations (tasklets), or map scopes that denote general parallelism (such as loops). The edges, or memlets, in an SDFG represent explicit data movements [24]. An example of the SDFG IR is provided in Figure 2. Here, the `square` function found in the source code corresponds to the outer tasklet in the SDFG, symbolized by the outer octagon. The reference `v` is the sole input and output of this function. This function contains a single computation and assignment `v = v*v;`, which is translated within the SDFG IR to the tasklet `b = a*a;`, where `a` is the input to this computation and `b` the output. To signify that the value stored in the reference `v` must be loaded prior and written to following the computation, memlets are used from `v` to and from the inner tasklet. Ultimately, the method `square` is invoked, which aligns with loading the parameter `A[i]` into the reference `v` and writing the result back to `A[i]`. In other



```
subroutine poisson(iflag)
  use global_parameters
  use field_array
  use particle_decomp
  implicit none

  integer iflag,i,it,ij,j,k,n,iteration,mring,mindex,mtest,ierr
  integer,dimension(:,:),allocatable :: nindex
  integer,dimension(:,:,:),allocatable :: indexp
  real(wp),dimension(:,:,:),allocatable :: ring
  real(wp) gamma,tmp,prms,perr(mgrid)
  real(wp) ptilde(mgrid),phitmp(mgrid),dentmp(mgrid)

  integer :: ipartd,nzeta,izeta1,izeta2
  real(wp),dimension(:),allocatable :: sendbuf,recvbuf
```

Scope	MPI 0/32 (us) (I)	MPI 0/32 (us) (E)	MPI 0/64 (us) (I)	MPI 0/64 (us) (E)	MPI 1/64 (us) (I)	MPI 1/64 (us) (E)...	Percent Excess Work
Experiment Aggregate Metrics	9.49e+08 100 %	9.49e+08 100 %	5.20e+08 100 %	5.20e+08 100 %	5.16e+08 100 %	5.16e+08 100 %	9.13e+00
viutil_spinandwaitcq	3.24e+07 3.4%	2.89e+07 3.0%	2.87e+07 5.5%	2.65e+07 5.1%	2.95e+07 5.7%	2.73e+07 5.3%	2.62e+00
MPID_DeviceCheck	3.24e+07 3.4%	2.89e+07 3.0%	2.87e+07 5.5%	2.65e+07 5.1%	2.95e+07 5.7%	2.73e+07 5.3%	2.62e+00
MPID_SendComplete	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
PMPI_Waitall	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
PMPI_Sendrecv	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
pmpi_sendrecv_	2.40e+07 2.5%	2.14e+07 2.3%	1.91e+07 3.7%	1.76e+07 3.4%	2.00e+07 3.9%	1.85e+07 3.6%	1.56e+00
MPID_RecvComplete	8.36e+06 0.9%	7.50e+06 0.8%	9.65e+06 1.9%	8.82e+06 1.7%	9.48e+06 1.8%	8.80e+06 1.7%	1.07e+00
poisson	1.66e+07 1.7%	1.64e+07 1.7%	1.70e+07 3.3%	1.68e+07 3.2%	1.71e+07 3.3%	1.69e+07 3.3%	1.83e+00
pushe	6.60e+08 69.5%	4.76e+08 50.1%	3.38e+08 65.0%	2.45e+08 47.2%	3.38e+08 65.5%	2.46e+08 47.6%	1.58e+00
__nanosleep_nocancel	3.00e+04 0.0%	3.00e+04 0.0%	4.78e+06 0.9%	4.78e+06 0.9%	1.47e+06 0.3%	1.47e+06 0.3%	6.56e-01
smooth	7.20e+06 0.8%	5.44e+06 0.6%	7.96e+06 1.5%	5.42e+06 1.0%	7.94e+06 1.5%	5.10e+06 1.0%	5.36e-01
intra_RDMA_barrier	5.06e+06 0.5%	4.28e+06 0.5%	5.15e+06 1.0%	3.82e+06 0.7%	5.20e+06 1.0%	3.69e+06 0.7%	3.39e-01

Figure 3: Left: Coloring in-memory volume onto memlets and arithmetic intensity onto tasklets in an SDFG [19]. Right: HPCToolkit's viewer, which displays the program's source code and its corresponding memory access information [15].

words, the two memlets of A from and to the function's tasklet correspond to the loading and writing actions.

After the SDFG IR of a program is constructed, it is possible to compute memory-related properties crucial for data locality. For instance, each memlet carries information regarding the volume of data transported between nodes [24], and tasklets and nested SDFGs can be annotated with metadata related to the number of executions and arithmetic operations undertaken [19]. Consequently, SDFGs offer a comprehensive view of the program and facilitate the identification of data movement bottlenecks on a large scale.

Despite static analysis's robust capability for macroscopic program analysis - a trait shared with dynamic analysis - it does not provide the same level of accuracy in the details. Given that performance bottlenecks are often induced by memory accesses that are tied to physical access patterns and hence are hardware-specific, static analysis alone may not accurately predict, for example, the number of cache misses for a particular function. However, the advantage of static analysis lies in the fact that it does not necessitate program execution, thereby enabling quicker and more cost-effective optimization of logical data movements compared to dynamic analysis.

3.3 Cache Simulation

Positioned between dynamic and static analysis lies the realm of simulation-based approaches, of which cache simulation is particularly noteworthy. Cache simulation is a method used to simulate a program's data accesses on a virtual memory hierarchy model. This process allows for an in-depth examination of both spatial and temporal data locality, as introduced on in Section 2.4.

The process of setting up a cache simulator can be divided into two ways:

In the first approach, the program is pseudo-executed without any of its actual computations. This process starts by constructing a virtual memory hierarchy that includes caches, in an optimal scenario fully reconstructing an identical virtual copy of the actual hardware in use. Here the underlying protocols of the hardware are also recreated, such as the cache miss procedure as discussed in Section 2.3. As the program proceeds through its lifecycle, corresponding space is allocated in the simulated memory for each instance of allocation, and reciprocally, space is deallocated as per the program's instructions. The simulator emulates each memory access operation, both read and write, according to how a CPU would handle the task. This entails an initial probe in the L1 cache, followed by a potential cache miss protocol if the required data is absent, as discussed in Section 2.3. This methodology facilitates a comprehensive and accurate representation of the system's memory hierarchy and its interaction with the computing process [18, 27].

The second approach uses dynamic analysis (Section 3.1) to generate memory traces. These traces are then rerun through the simulator similarly to the previously described approach. Replaying memory traces enables performance engineers to better understand memory access and management behavior within the program [28].

The deployment of cache simulators extends beyond mere prediction of cache misses. When operated in a step-by-step manner, these tools permit the exploration of data access patterns of a procedure at a granular level. Such detailed inspection can uncover potential enhancements in spatial locality, either through modifications in data layout or access strategies, ultimately contributing to improved performance [18, 27, 28].

Moreover, cache simulation enables close-up performance analysis of a program, such as focusing on a single function within the source code or limiting memory traces to a specific functional context.

Despite its advantages, accurate cache simulation demands accurate representation of the target architecture, including aspects like cache hierarchy, cache replacement policy, and cache coherence protocol. Any inaccuracies in these parameters can lead to misleading results, potentially causing optimization attempts to inadvertently degrade the program's data locality. As such, securing the necessary information to build a virtual memory hierarchy, whether automated or utilizing the performance engineer's extensive knowledge of the hardware, is essential for successful performance optimization through cache simulation.

4 Visualization Techniques

Visualization is an essential aspect of data locality analysis, providing the vital link between the analysis results and user comprehension. Effective visualization techniques should balance intuitiveness and informational value. Generally, visualizations for memory-related data can be classified into three categories, each catering to a specific level of detail:

4.1 High-Level View

This category of visualization provides the most abstract or "bird's eye" view of data locality, aiming to deliver a global understanding of the program's performance. It emphasizes the logical data movement behavior, spotlighting the performance impact of individual parts of the program [14, 15, 18, 19], as illustrated in Figure 3. The left portion of the figure displays a colored-in SDFG IR of the program, used to demonstrate the arithmetic loads of specific program parts (tasklets), as well as the volume of data circulated throughout the program (memlets). Here, tasklets and memlets marked in red signify above-average intensity, indicating these areas are particularly noteworthy for further exploration concerning performance bottlenecks [19]. The figure's right side presents a snapshot view of the `HPCToolkit`'s viewer, wherein performance-related issues can be traced to their source using a hierarchical view of the program's execution [15]. In general, an integral feature of such high-level visualizations is hierarchical clustering, which allows users to zoom into specific areas of the program. Despite providing a broad performance landscape, these visualizations do not shed light on the root causes of identified bottlenecks, warranting a more detailed examination.

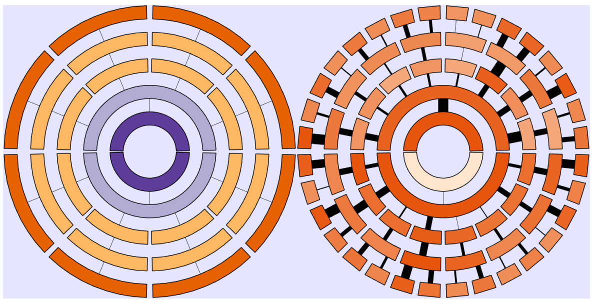


Figure 4: Left: Illustration of the radial design used to represent the hardware topology. Main memory resources are here illustrated in deep purple and processing units in dark orange, with different layers of caches in between. Right: A concrete complex architecture is portrayed based on the radial design on the left, featuring performance data tagged onto the hardware resources through a color code, and transactions among them indicated by line thickness [14].

4.2 Intermediate-Level View

At the intermediate level, visualizations offer more detailed insights than high-level overviews, targeting specific segments of the program like functions or loops. One common technique involves displaying the hardware topology to visualize physical data movements across different levels of the memory hierarchy, as well as operational intensity for each memory module, as shown in Figure 4. This more granular perspective assists in understanding performance bottlenecks and can help identify the most promising optimization opportunities [14, 28]. Yet, it does not provide information about underlying problematic data layout or access patterns, necessitating a deeper, fine-grained examination.

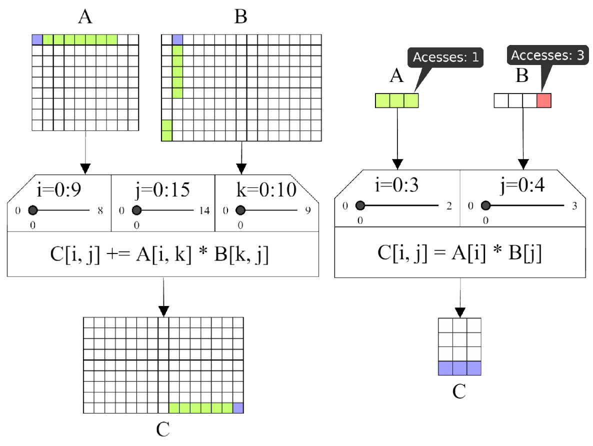


Figure 5: Left: Illustration of data layouts emphasizing spatial locality via cache lines. Right: Visualization of correlated accesses to elements A and B with respect to accesses to $C[3, x], x \in \{0, 1, 2\}$ [18].

4.3 Detailed View

The detailed or fine-grained visualizations delve into specific aspects like data layout and access patterns within specific program segments, for instance, a loop nest. Figure 5 displays two examples of such visualizations. The left image illustrates the data layout of a matrix, highlighting the spatial locality of elements within a cache line. The right image presents the access patterns of a loop nest, showing the correlation between accesses to different arrays. This level of visualization aids in identifying potential optimization routes, such as reshaping data to improve spatial locality or reordering the loop to enhance data access patterns [18]. However, these visualizations provide insights into only one program segment at a time and lack a broader picture of the program's overall data locality. Therefore, it is advisable to use a mix of visualization techniques to obtain a comprehensive understanding of a program's data locality.

5 Optimization Workflow

In the realm of performance optimization, the workflow that an engineer undertakes unfolds in a progressive manner, moving from a macroscopic to a microscopic examination of a program's performance characteristics. This sequential inspection process serves to identify, understand, and eventually resolve performance bottlenecks, particularly those related to data locality.

The optimization journey commences with a high-level, panoramic view of the program's performance landscape (Section 4.1). This abstracted perspective provides a global understanding of how the program operates, emphasizing performance aspects on a module, function, or code-line level. However, while these coarse-level views may signal where performance issues lie, they often fall short in explaining the "why" behind these issues - the root causes that contribute to elevated memory intensity or the sub-optimal utilization of resources.

For these deeper insights, the engineer transitions to more granular, intermediate-level views (Section 4.2). These visualizations elucidate the interactions between particular program components and the memory hierarchy, shedding light on data movements across different cache or memory levels and their impact on performance.

In cases where performance irregularities remain elusive, the engineer resorts to the most detailed, low-level views (Section 4.3). These visualizations put the data access patterns under a microscope, offering the necessary detail to pinpoint, understand, and eventually rectify the root causes of performance issues.

This stepwise deepening in focus, from high to intermediate to low-level views, constitutes the typical progression within the performance optimization workflow. However, it is crucial to note that not every tool caters to each level of granularity.

In the following section (Section 6), we will explore several prominent tools dedicated to visualizing memory movements and accesses. We will discuss their capabilities in data gathering, visualization techniques, and their ability to provide insights at different levels of detail. We will contrast these tools, emphasizing their respective strengths and weaknesses, and the extent to which they support the comprehensive workflow outlined above.

6 Exemplary Memory Access Visualization Tools

This section examines several prominent works dedicated to visualizing memory movements and accesses. Each tool will be discussed in terms of its data gathering methods, visualization techniques, and demonstrated results. We will then contrast these tools, highlighting their strengths and weaknesses.

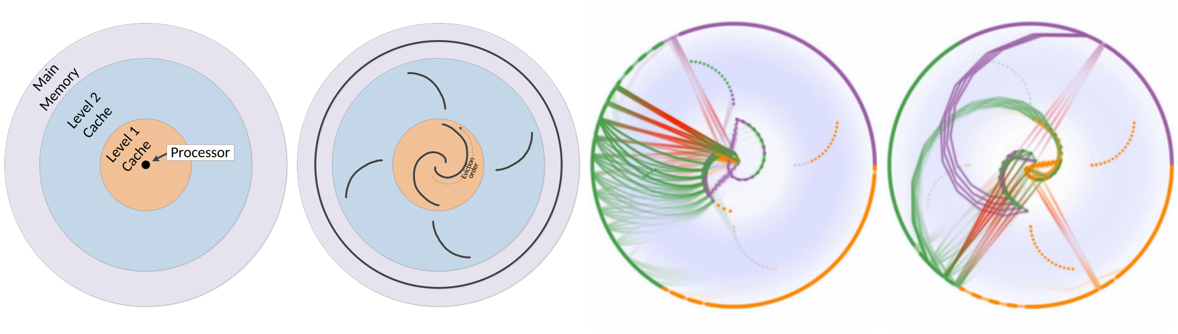


Figure 6: Left: Radial design used in [28]. Glyphs arrange themselves into groupings indicating storage on the same cache, with data closer to the boundaries between the levels more likely to be evicted. Right: A comparison of a standard 16 × 16 matrix multiplication and an optimized version using 4 × 4 blocking. The colors green, purple, and orange represent the memory locations belonging to the left- and right-hand matrices of the multiplication, and its output matrix. Red colored traces signify cache misses. The standard version shows poor data reuse for two of the three matrices [28].

6.1 MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors

The tool `MemAxes`, developed by Giménez et al. [14], utilizes dynamic analysis (Section 3.1) to generate an event log of memory accesses. Each logged event incorporates contextual information, facilitating a link back to the source code and recording the memory hierarchy depth at which the memory access occurred. This feature allows for the identification of problematic code lines, similar to `HPCToolkit` [15], as illustrated in Figure 3. Recording the resolution depth of memory accesses enables the determination of resource utilization across each memory module and the quantification of physical data movements between them. This information is then visualized using a radial design of the hardware topology, as seen in Figure 4. `MemAxes` also supports the display of additional attributes such as access times, latencies, and memory addresses through histograms.

In practical applications, `MemAxes` has been employed successfully to detect and mitigate performance bottlenecks, even without prior knowledge of the application's source code [14]. Performance engineers can, for instance, identify large load imbalances or significant spikes in access times, and use these insights to hypothesize the cause of a performance bottleneck. This hypothesis can then be explored further through the backlink to the source code. This approach demonstrates that low-level visual aids are not always necessary for optimizing data locality in an unfamiliar program.

6.2 Abstract Visualization of Runtime Memory Behavior

Choudhury et al. [28] offer a unique perspective on runtime memory behavior through their visualization tool, conceptually different from `MemAxes`. Their approach involves dynamic analysis to record an event log of memory accesses during runtime (Section 3.1), which then feeds a cache simulator (Section 3.3). The output is a series of radial visualizations, exemplified in Figure 6, which are generated throughout the program's execution, forming an animation of evolving data movements within the memory hierarchy.

The visualization in Figure 6 uses a concentric layout to demonstrate memory usage patterns. Glyphs, symbolizing memory locations, move across layers representing main memory and different cache levels. Movements towards the center imply recent references, while those towards the periphery indicate aging or eviction. Performance issues, such as inefficient memory usage or frequent evictions, are suggested by rapid, large-distance glyph movements. Conversely, slow in-layer movement indicates high cache hits, signaling efficient memory utilization [28].

Choudhury et al. argue that this dynamic approach is more intuitive than static visualizations, such as those provided by `MemAxes`, as it presents an overview of large-scale memory access and caching behavior. However, its granularity is insufficient for targeted bottleneck resolution, given that the visualizations lack linkage to specific contextual information such as precise addresses or lines of code.

6.3 Boosting Performance Optimization with Interactive Data Movement Visualization

The tool developed by Schaad et al. [18, 19] enables two-tier program analysis: At the global level, static analysis (Section 3.2) is used to compile the program source code into an SDFG graph, providing an overview as shown in Figure 3. This graph, with its color-customizable nodes and edges, aids in identifying problematic program sections, especially when utilizing the automatic node and edge collapsing feature for easy zooming. For in-depth analysis of data locality and reuse behavior, the tool uses cache simulation (Section 3.3) to offer detailed views of specific program segments, as depicted in Figure 5.

The authors successfully employed this tool to significantly optimize two applications. After pinpointing problem areas in the global view, the engineers utilized the local view for a thorough investigation and subsequent optimization of these areas.

6.4 Comparison

Among the three tools, the animation provided by Choudhury et al. [28] offers the most intuitive understanding of large-scale memory access and caching behavior. However, comprehensive program optimization requires contextual information about inefficient memory access locations, which is supplied by `MemAxes` [14] and the tool by Schaad et al. [19]. Of all three, Schaad et al.'s tool provides the most detailed low-level visualizations. The tool's ability to depict the influence of data layout on the cache hit ratio proves invaluable for optimizing data locality. However, the tool's reliance on cache simulation necessitates consideration of parameterization, as discussed in Section 3.3.

7 Conclusions

This paper provided a comprehensive exploration of the different methodologies employed to understand and visualize data movements and accesses in computer programs, an imperative for optimizing performance. The data acquisition processes are based on three principal methodologies, which can often be combined to achieve a more comprehensive analysis. Dynamic Analysis, involving the execution of a program to gather data, provides accurate results, albeit time-consuming. Static Analysis, on the other hand, is a swift method that just analyzes the source code, bypassing the need for program execution, but can sometimes produce insufficient insights. Lastly, Cache Simulation offers an in-depth understanding of a program's interaction with the memory hierarchy, allowing for a meticulous examination of its performance.

Once the data locality information is collected, it becomes essential to present it in an intuitive yet informative way. Here, visualizations play a critical role. These visualizations, varying in their granularity, offer insights into different levels of the program, from high-level overviews to fine-grained dissections of memory usage. The use of multiple visualizations in tandem empowers performance engineers to gain a comprehensive understanding of data movements and accesses, leading to effective bottleneck resolution.

In this regard, we delved into three prominent works that provide unique tools for visualizing data movements and accesses. Each work was presented and compared, highlighting their distinctive contributions to the wider landscape of methods employed in this domain.

The broad-ranging implications of these methods were underscored, highlighting their relevance to not just high-performance computing (HPC) but to any application that would benefit from performance optimization. This paper thus provides a solid foundation for understanding and visualizing data movements and accesses, a crucial aspect of programming and performance engineering.

As we look to the future, the field holds exciting prospects. The continuous refinement of data gathering methods and visualizations is one aspect, but the ultimate goal extends to automatic program optimization. Initiatives have already been launched to develop algorithms for automatic optimization [20], significantly easing the programmer's workload. Such advancements could be especially beneficial for domain researchers, allowing them to focus more on their domain-specific work.

The emergence of machine learning, particularly deep learning, also presents new avenues for program optimization. Preliminary work in this area indicates that future compilers might utilize machine learning for automatic, compile-time program optimization [29].

In conclusion, the tools and methods presented in this paper provide a solid foundation for understanding and visualizing data movements and accesses, a crucial aspect of optimizing program performance. With ongoing advancements in automatic program optimization and the advent of machine learning techniques, the future looks bright for further improvements in this vital aspect of programming and performance engineering.

References

[1] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. “A survey of different approaches for overcoming the processor-memory bottleneck”. In: *International Journal of Computer Science and Information Technology* 9.2 (2017), pp. 151–163.

[2] Philip Machanick. “Approaches to addressing the memory wall”. In: *School of IT and Electrical Engineering, University of Queensland* (2002).

[3] Sally A McKee. “Reflections on the memory wall”. In: *Proceedings of the 1st conference on Computing frontiers*. 2004, p. 162.

[4] John D McCalpin. “A survey of memory bandwidth and machine balance in current high performance computers”. In: *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)* (1997).

[5] Wm A Wulf and Sally A McKee. “Hitting the memory wall: Implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[6] Nihar R Mahapatra and Balakrishna V Venkatrao. “The processor-memory bottleneck: problems and solutions.” In: *XRDS* 5.3es (1999), 2–es.

[7] Herb Sutter et al. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobbs journal* 30.3 (2005), pp. 202–210.

[8] John David McCalpin. “Memory bandwidth and system balance in hpc systems”. In: *UT Faculty/Researcher Works* (2016).

[9] Our World in Data. *Artificial Intelligence Parameter Count*. `https://ourworldindata.org/grapher/artificial-intelligence-parameter-count`. [Online; accessed May 12th, 2023].

[10] Didem Unat et al. “Trends in data locality abstractions for HPC systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020.

[11] Adrian Tate et al. *Programming abstractions for data locality*. Tech. rep. Office of Scientific and Technical Information (OSTI), 2014.

[12] Karim Esseghir. *Improving data locality for caches*. Rice University, 1993.

[13] Marty Itzkowitz et al. “Memory profiling using hardware counters”. In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2003, p. 17.

[14] Alfredo Giménez et al. “Memaxes: Visualization and analytics for characterizing complex memory performance behaviors”. In: *IEEE transactions on visualization and computer graphics* 24.7 (2017), pp. 2180–2193.

[15] Laksono Adhianto et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[16] Sameer Shende. “Profiling and tracing in linux”. In: *Proceedings of the Extreme Linux Workshop*. Vol. 2. 1999.

[17] Kathryn S McKinley and Olivier Temam. “Quantifying loop nest locality using SPEC’95 and the perfect benchmarks”. In: *ACM Transactions on Computer Systems (TOCS)* 17.4 (1999), pp. 288–336.

[18] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. “Boosting performance optimization with interactive data movement visualization”. In: *arXiv preprint arXiv:2207.07433* (2022).

[19] Philipp Schaad. “Boosting Performance Engineering with Visual Interactive Optimization and Analysis”. MA thesis. ETH Zurich, 2021.

[20] Alexandru Calotoiu et al. “Lifting C semantics for dataflow optimization”. In: *Proceedings of the 36th ACM International Conference on Supercomputing*. 2022, pp. 1–13.

[21] Tal Ben-Nun et al. “Bridging Control-Centric and Data-Centric Optimization”. In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 2023, pp. 173–185.

[22] Chris Lattner et al. “MLIR: A compiler infrastructure for the end of Moore’s law”. In: *arXiv preprint arXiv:2002.11054* (2020).

[23] William S Moses et al. “Polygeist: Raising C to polyhedral MLIR”. In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2021, pp. 45–59.

[24] Tal Ben-Nun et al. “Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–14.

[25] Stan Matwin and Tomasz Pietrzykowski. “Prograph: a preliminary report”. In: *Computer Languages* 10.2 (1985), pp. 91–126.

[26] Jeffrey Kodosky. “LabVIEW”. In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), pp. 1–54.

[27] Julian Hammer et al. “Kerncraft: A tool for analytic performance modeling of loop kernels”. In: *Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany*. Springer. 2017, pp. 1–22.

[28] ANM Imroz Choudhury and Paul Rosen. “Abstract visualization of runtime memory behavior”. In: *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE. 2011, pp. 1–8.

[29] Chris Cummins et al. “Programl: A graph-based program representation for data flow analysis and compiler optimizations”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 2244–2253.