



Boosting Performance Optimization with Interactive Data Movement Visualization

Philipp Schaad 

Department of Computer Science
ETH Zürich
philipp.schaad@inf.ethz.ch

Tal Ben-Nun 

Department of Computer Science
ETH Zürich
talbn@inf.ethz.ch

Torsten Hoefler 

Department of Computer Science
ETH Zürich
htor@inf.ethz.ch

Abstract—Optimizing application performance in today’s hardware architecture landscape is an important, but increasingly complex task, often requiring detailed performance analyses. In particular, data movement and reuse play a crucial role in optimization and are often hard to improve without detailed program inspection. Performance visualizations can assist in the diagnosis of performance problems, but generally rely on data gathered through lengthy program executions. In this paper, we present a performance visualization geared towards analyzing data movement and reuse to inform impactful optimization decisions, without requiring program execution. We propose an approach that combines static dataflow analysis with parameterized program simulations to analyze both global data movement and fine-grained data access and reuse behavior, and visualize insights in-situ on the program representation. Case studies analyzing and optimizing real-world applications demonstrate our tool’s effectiveness in guiding optimization decisions and making the performance tuning process more interactive.

I. INTRODUCTION

With the end of Moore’s Law [1] and Dennard scaling [2], performance optimization in modern high performance computing applications is more important than ever. However, the complex and multifaceted nature of a program’s performance in modern HPC environments necessitates conducting a careful and extensive performance analysis to make informed decisions about performance optimization steps. Due to the increasing transistor efficiency on chips, but the relatively constant cost of transferring data, a particularly important metric for performance analysis and optimization is the amount of data movement, with the goal being to exploit data locality and reuse as much as possible [3], [4].

To successfully analyze the performance of complex applications, engineers employ analysis tools such as profilers. Further instrumenting code to read data from timers or hardware counters can help gather information about fine-grained program behavior. These methods generate a wealth of data, which in itself can be difficult to understand without the help of performance visualization tools that aggregate and present it in an intuitive and understandable way. However, gathering the raw data generally requires running the entire application, which in many cases can take prohibitively long and slow down the often iterative optimization process.

In this work, we provide a performance visualization technique that gathers and shows measurements based on simulation and static dataflow analysis to assist engineers in ana-

lyzing and optimizing the data movement and reuse behavior of their application, without requiring program execution. We employ a graphical program representation that naturally exposes data flow, to build intuitive in-situ visualization overlays on the program’s dataflow graph. By mapping performance metrics directly onto the program, the cognitive load required for the attribution of observations to the original code, and the time required to perform a root-cause analysis are significantly reduced.

To reduce the need for costly, full-scale executions of an application, we propose an approach that simulates individual, parameterized program parts, to estimate an application’s data access and reuse behavior. By reducing the wait time for performance data from minutes or hours to a fraction of a second, this approach enables a more interactive performance optimization process.

To facilitate program analysis on multiple granularities, we construct our visualization using a two-level approach. A coarse view mode provides a global understanding of the program and aims to support fast and reliable data movement bottleneck detection. This is achieved by analyzing static program information such as logical data movement volumes and arithmetic operation counts, and visualizing the resulting metrics using in-situ overlays on a graphical program representation. This view mode facilitates the analysis of the overall algorithmic design and data movement or communication scheme, and enables scalability analyses with rapid feedback through the use of symbolic analysis.

A second, fine-grained view mode allows for close-up analysis of data locality and reuse behavior in individual program parts. This view mode connects statically obtained *logical* data movement to *physical* data movement, by simulating a program’s data access patterns on a simple hardware model. The view mode exposes this information alongside visualizations of temporal and spatial data locality, physical data layouts, and cache miss estimations.

We implement our visualization as a Visual Studio Code [5] extension and demonstrate its use in a realistic scenario using two case studies. By analyzing and optimizing the encoder layer for a BERT transformer deep neural network [6], and a horizontal diffusion weather stencil, we demonstrate analysis of both global data movement and close-up memory layout and reuse.

In summary, this paper makes the following contributions:

- Global data movement visualization for performance analysis.
- Approach to data movement estimation using parameterized, small-scale data access simulations.
- Visualization methods for hardware model-augmented data locality and reuse analysis.

II. BACKGROUND

The primary goal of performance engineering is to make an application perform more efficiently, typically by reducing its runtime. For any reasonably large program, achieving this goal is a complex process. An application’s performance depends on a large number of factors, both from inside the program source code, but also external factors, such as the hardware or what data the program operates on.

To undertake effective optimizations, a performance engineer first needs to understand the performance characteristics of the application with a detailed analysis. Only then can they perform educated optimization steps to improve particular aspects of the measured performance. Given the complex interplay between different system factors, an application’s performance then typically has to be reassessed before further optimizations can be performed, since even small changes and optimizations can have large effects on the previously measured performance.

This sequence of performance analyses followed by optimization steps is repeated until a program’s performance targets are met. Given the iterative nature of this procedure, shortening either the analysis or optimization step can create a large speedup in the overall performance engineering process. With this work, we focus on reducing the time spent in the analysis step of this process.

A. Performance Analysis

Model-driven performance tuning [7] suggests a top-down workflow during the analysis process. Engineers initially identify theoretical limits to the achievable performance, gauging the proximity to the obtainable optimum. They further identify input parameters, program parts (or *kernels*), and communication or data movement patterns that impact the system’s performance. This information helps them focus further close-up analyses and optimization efforts with more fine-grained metrics.

Engineers have a variety of tools at their disposal to acquire such performance metrics. They can choose to manually instrument their code to generate timing information, use hardware counters [8], or run an application with detailed, and often hardware-specific profiling tools like NVIDIA’s *nvprof* [9], Intel’s *VTune* [10], or *gprof* [11]. The large amount of performance data generated by these methods is typically challenging to understand and needs to be carefully sifted through to extract useful information [12].

Performance visualization tools can help simplify the process of interpreting the wealth of generated performance data. In particular, they present the data in a clear and aggregated

manner, often highlighting a specific aspect of a program’s performance. This enables engineers to rapidly pinpoint what particular performance problems can be attributed to.

III. PRODUCTIVE PERFORMANCE VISUALIZATION

To support a top-down performance analysis process, we equip our visualization with two separate view modes, each specialized towards a specific set of tasks in different stages of the analysis workflow. A global view is provided to assist in building an overview of the application and to perform an analysis of coarse program structures, such as communication and data movement patterns or the overall algorithmic design. Additionally, the global view allows engineers to quickly identify program kernels or input parameters with high performance impact.

A second, local view specializes on fine-grained performance analysis of specific, smaller program parts. The main task of this view is to highlight more detailed performance metrics that help inform specific tuning decisions. Both view modes are described in detail in Sections IV and V.

To best support engineers in a productive optimization and analysis process, and to facilitate a top-down information seeking behavior, we stipulate that our performance visualization should adhere to the following principles:

- **Performance:** The visualization must primarily show factors that are important for performance optimization.
- **Minimality:** To avoid clutter, anything that is not strictly necessary for the task at hand, should not be shown.
- **Attribution:** It must remain clear what system or program elements specific parts of the visualization depict and relate to.
- **Continuity:** Changes in the visualization should not happen unexpectedly, and visual differences should carry a clear meaning.

A. Highlighting Data-Movement

Based on the **performance** principle and the importance of data movement optimizations in modern HPC applications [3], [4], we design our visualization centered around data movement, which can be analyzed separately from the computation and used to promote data locality and reuse. To express this, we need to represent programs using a graph-based dataflow intermediate representation, which can be used to visualize performance metrics in-situ as overlays or augmentations directly in and on top of the program, simplifying the attribution of observations to the responsible program parts. In such graphical dataflow representations, graph nodes represent computations and data containers, while directed edges between them represent data flowing through the program.

There are a number of graph-based dataflow programming languages or representations, like *PROGRAPH* [13], *Stateful Dataflow Multigraphs (SDFGs)* [14], or *LabVIEW* [15], that can be used to express this aspect of a program. For this work we use SDFGs to extract dataflow from general programs, because the accompanying data-centric parallel programming

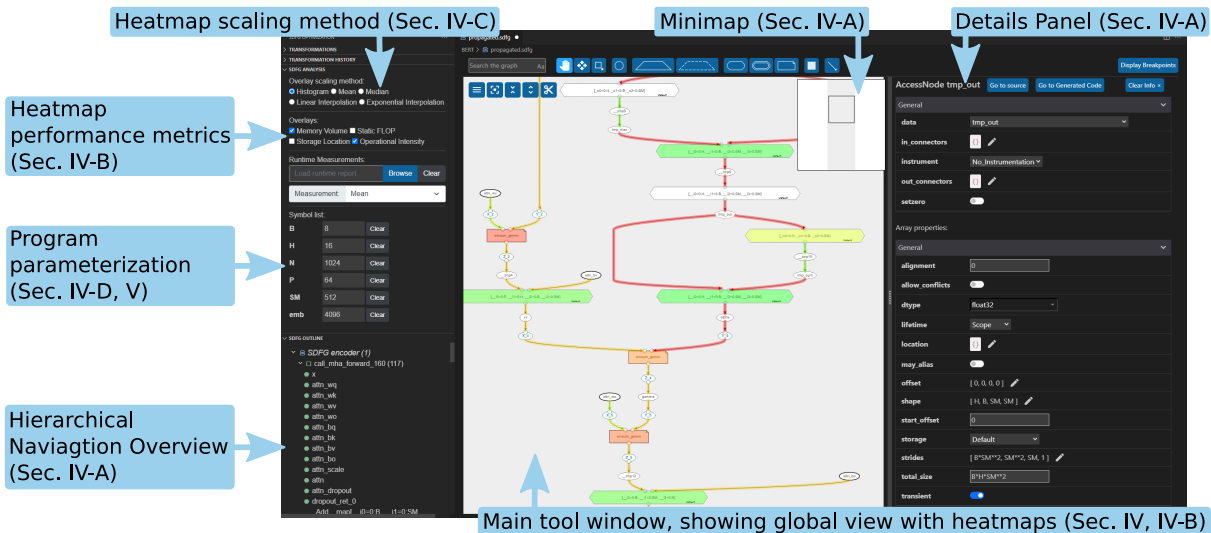


Fig. 1: Screenshot of the visualization tool’s main interface.

framework (DaCe) can compile programs from both Python and C programs [16] into this intermediate representation, and they can directly be used for subsequent optimizations in DaCe.

IV. GLOBAL VIEW: DATA MOVEMENT ANALYSIS

In our two-level approach to performance analysis, the global view is first tasked with providing a global comprehension of the program’s performance characteristics. The view is further responsible for exposing coarse data movement behavior, and highlighting the performance impact of individual program parts, to assist engineers in problem detection and diagnosis. Lastly, the global view should facilitate the identification of input parameters which have a large impact on performance, by providing a fast-feedback scaling analysis.

A. Global Comprehension

To give the engineer an overview of the global state of a program’s behavior, this initial view provides mechanisms for exploring the entire application in its graphical representation. When interacting with elements in this representation, the option is given to jump to the corresponding location in the original source code representation, helping to fulfill the **attribution** principle.

Maintaining an overview can be difficult for complex applications. Particularly, as programs get larger, the number of visual elements needed to represent the application grows together with the number of code lines in the program’s source representation. To ensure legibility of large programs, it is imperative that the principle of **minimality** is followed. Traditional code editors such as Visual Studio Code (VS Code) [5], Atom [17], and most integrated development environments address this by allowing for logical code regions to be folded into a single line. The SDFGs used to extract our dataflow graph representation are constructed in a hierarchical manner. Individual program parts, or subgraphs, form logical regions similar to their counterparts in other high-level languages (e.g.

loop contents and subroutines). We exploit this to allow entire subgraphs to be folded and hidden, instead representing them with a single graph element that summarizes their content until they are deliberately expanded again.

To further improve legibility in large applications, more detailed visual elements are gradually hidden as the user zooms further out, similarly to Google Maps [18], dynamically pulling focus towards the more coarse-grained structure of the application. Additionally, the graphical representation contains only information strictly necessary to analyze the functional behavior and dataflow of an application. Any additional information like data types, sizes, and alignment are hidden away and appear on-demand in a separate details panel or in tooltips, shown only when the user interacts with the corresponding visual object. As with traditional source code, the graphical representation can be searched to find specific elements, and it further allows for some types of elements to be filtered out and hidden from view.

The user can employ the type of pan-and-zoom navigation common in mapping software to explore the graphical program representation. To assist in navigation in accordance with the principle of **continuity**, two separate overviews help maintain situational awareness. A minimap in the top right corner of the visualization shows the current program in its entirety, with a box drawing the current viewport in relation to the graph. A second, outline overview shows a hierarchical view of the graph, enabling quick navigation to a specific graph element by selecting it from this list. This type of overview in combination with pan-and-zoom navigation helps in analyzing large program graphs, and has been shown to be a very efficient mode of exploration that is perceived as enjoyable by users [19]. Any automatic navigation through interaction with one of the overviews is further animated as a slowed down motion of the viewport to maintain continuity and avoid disorientation.

An overview of the interface can be seen in Fig. 1, showing the global analysis view on the graphical program representa-

tion, embedded in the popular code editor VS Code. A short supplementary video¹ demonstrates program navigation and exploration in the global view.

B. Problem Detection and Diagnosis with Heatmaps

To facilitate efficient problem detection and diagnosis, the global view enables reasoning about the performance of the program as a whole. Given the importance of reducing data movement when improving performance, it is crucial that optimizations focus on maximizing data reuse. This can be achieved by coalescing computations that rely on the same data, which allows for better utilization of caches, in turn reducing the amount of data that needs to be read from or written to main memory. In doing so, one increases the arithmetic intensity, i.e., the number of arithmetic operations performed per transferred data byte.

The amount of data being accessed by or moved between individual operations in the program is statically determined when SDFGs are generated in a dataflow extraction procedure [16]. We can use this information to augment the graphical program representation, by showing a color-coded heatmap as an overlay directly on top of the program’s dataflow graph. This heatmap uses a green-yellow-red color spectrum to mark data movement edges with a color corresponding to the relative amount of data being moved, where green represents a low amount of data, and red represents higher volumes.

We extract information on arithmetic or operational intensity separately by parsing the abstract syntax tree of individual computations, counting the number of arithmetic operations. This can be used to directly construct a heatmap, coloring nodes based on their arithmetic operations count, or be combined with data access information to color nodes based on their arithmetic intensity.

This form of color-coded overlays shown directly on top of the program structure has been found to be an effective tool for communicating additional information to software engineers [20]. We exploit this technique by applying it to the graphical program representation, with the goal of reducing the cognitive load required to perform attribution of individual measurements to their corresponding program parts. Fig. 1 shows the global view of a program colored using both a logical memory movement volume and arithmetic intensity heatmap.

The proposed visualization is not directly tied to static analysis. Profiling data could orthogonally be used as metrics, which would be crucial for bottleneck analysis of data-dependent programs.

C. Heatmap Coloring

In real-world applications, we observe large value ranges for performance metrics, with data movement volumes ranging from individual bytes, to multiple megabytes or gigabytes. These magnitude changes, even within applications, make determining an individual value’s heatmap color with any

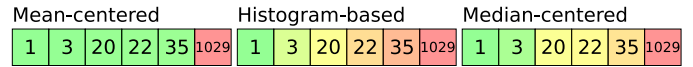


Fig. 2: Heatmap scaling methods and their respective uses.

fixed scale impractical. Color scales consequently have to be adaptive and account for varying value distributions.

Heatmaps are a popular tool in performance visualization systems [21]–[24], and there are different approaches to handle this. The Scalasca toolkit’s visualization component, Cube, has a separate plugin with options for the user to change this scaling behavior by switching the interpolation method used for determining the scale between using linear or exponential interpolation from the minimum to maximum observed values [25], [26].

We employ three further approaches not based on interpolation, to serve three separate use cases. The user can dynamically select the fitting one, updating the heatmap display. The first two methods work by determining a center value c for the color scale, and then setting the scale to run in the interval $[0, 2c]$. The center value c is determined by sorting all observed data values in increasing order, and then picking either the *median* or *mean* of all observations to form the center of the scale. Observations above the maximum of the scale are clamped to $2c$. In a third method, we group the observed values into histogram buckets and set the color scale to the interval $[0, n]$, where n is the number of distinct buckets. A value’s color is then chosen based on the position of index i of its corresponding histogram bucket on the scale.

Fig. 2 demonstrates the use cases for each of these scaling methods, highlighting their behavioral differences. Centering the scale around the mean (Fig. 2, left) is heavily influenced by outliers, making it ideal for detecting bottlenecks by giving them visually distinct colors from the remainder of the distribution. Histogram-based scaling (Fig. 2, middle) distorts the scale to give each distinct observation a different color. This is the most useful method for clearly highlighting the distribution of observed values, independently of the distances between observations. Centering the scale around the median (Fig. 2, right) fills the gap between these two methods: by being more outlier resistant than mean-centered scaling, outliers are less visually distinct, but by distorting the scale less than histogram-based scaling, this method is ideal for visually grouping values of similar magnitudes with similar colors.

The colors used to represent these scales are equally important, and some color schemes used in many visualization systems, such as rainbow maps (also known as *jet* maps), are less than desirable and can be actively confusing [27]–[29]. To combat this, a popular alternative is a green-red spectrum, which leverages intuitive color associations of red=slow and green=fast. This works well with sparse data sets, where individual values are well separated. However, for data sets where distinction between individual values is important, and where values are closer relatively to one another, there is little visual separation between individual data points.

¹<https://youtu.be/sxMmEnzkX64>

We address this by introducing the additional color yellow in the center to increase this separation while keeping the clear color ordering from fast to slow:



To further account for color blindness, this color scale can be manually changed to fit the user’s needs.

D. Parametric Scaling Analysis

In many applications, computations and program behavior are largely dependent on the input size of the data or other runtime parameters. As such, many resulting arithmetic operations or data volume counts are expressed as symbolic expressions that depend on input data dimensions or parameters passed to the program at runtime. The SDFG representation used in the global view is thus inherently *parametric*, a fact that can be exploited to observe how program performance is affected by changes to input parameters, and by extension identify which parameters have a particularly large impact on program performance.

By allowing the user to define and change values for input parameters in a configuration panel, we can adapt the heatmap visualizations on the fly by re-evaluating symbolic expressions with the new values. This facilitates an analysis of the program’s scaling behavior with respect to input sizes or runtime parameters. The user can visually follow how data movement volumes, operational intensity, or localized bottlenecks are affected by changes to the data sizes. With this, we allow the user to interactively determine what input parameters are crucial factors in the program’s performance, without requiring costly program executions.

The use of heatmaps, including parametric scaling, is further demonstrated in a short supplementary video².

V. LOCAL VIEW: LOCALITY AND REUSE ANALYSIS

To perform close-up analyses of data locality and reuse behavior, we employ a small-scale simulation approach, which approximates values for the program’s runtime behavior with respect to data accesses and layouts. Observations of this behavior are conventionally obtained via profiling and program counter collection, or with costly hardware simulations. This approach is particularly well suited for localized bottleneck analysis, avoiding executions of the entire application. By instead solely simulating the relevant program kernel, this enables more interactive, fast-iterating optimizations.

Engineers can specify a region of the program where a closer investigation is desired, and provide sample values for input parameters that determine execution behavior, such as data dimensions. This focuses the view on only the specified part of the graphical representation, and adapts the visualization to better facilitate this close-up analysis.

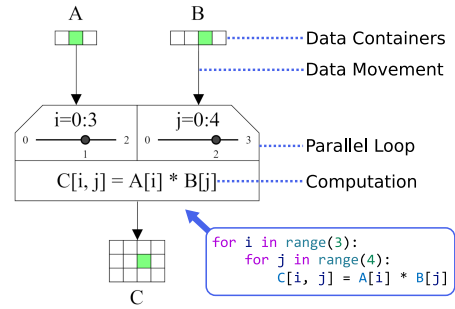


Fig. 3: Parameterized outer vector product $C = A \otimes B$, for $A \in \mathbb{R}^3$, $B \in \mathbb{R}^4$, and $C \in \mathbb{R}^{3 \times 4}$. The sliders set to the loop parameters $i = 1$ and $j = 2$ highlight memory elements accessed with those parameters (green).

A. Program Parameterization

With provided input parameters, we *parameterize* the program view, augmenting graph elements that depend on input parameters to reflect the given concrete values. Specifically, nodes representing data containers such as arrays are now expanded to reflect their parameterized size, showing each individual data element, to represent individual memory locations.

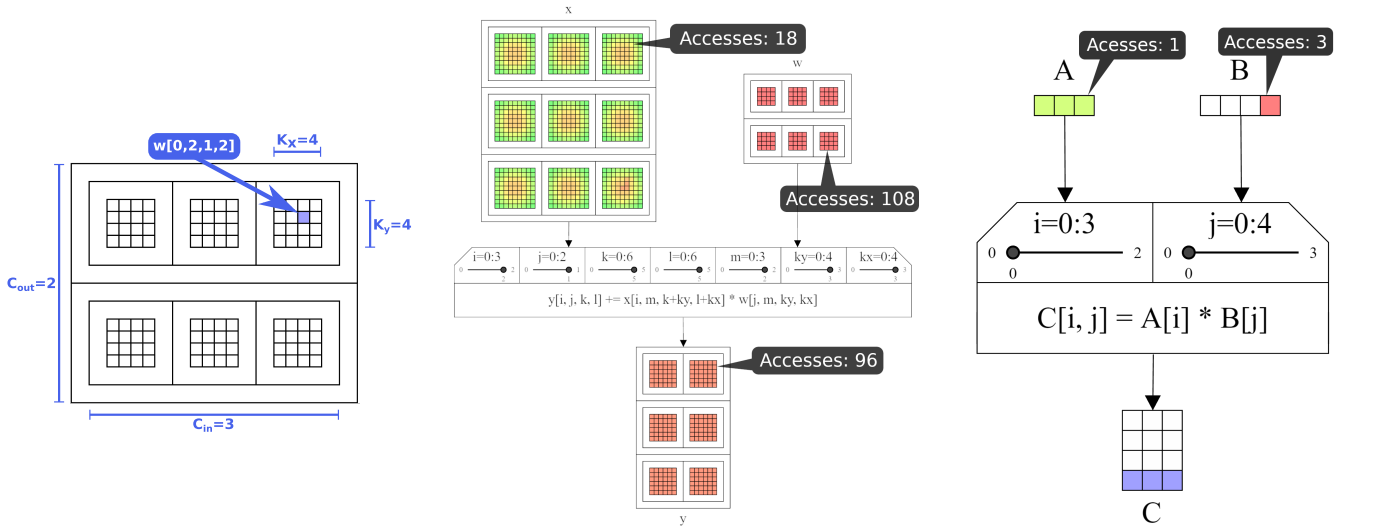
Parallel loops and concurrent regions with parametric bounds are also parameterized and have their bounds fixed. These structures are shown as boxes with trapezoidal header bars, where everything inside the box represents the loop or concurrent region’s contents, and the trapezoidal header bar shows loop parameters and their bounds. The example in Fig. 3 shows the outer product calculation of two vectors $A \in \mathbb{R}^3$ and $B \in \mathbb{R}^4$, where a parallel loop with an iteration space given by $i \in [0, 2]$ and $j \in [0, 3]$ contains the calculation $C[i, j] = A[i] * B[j]$ in its loop body. All data containers show an individual tile for each element. Additionally, each parameter to the parallel region is accompanied by a slider with which the engineer can set individual parameter values. This consequently highlights all memory elements accessed inside the parallel region for that specific parameter combination, as demonstrated in Fig. 3 for parameters $i = 1$, $j = 2$.

B. Visualizing High-Dimensional Data

While it is intuitive to visualize one or two dimensional data on a 2D surface such as the user’s screen, many programs deal with data containing more than two dimensions. Shneiderman’s taxonomy for visualization systems [30] identified a number of ways to deal with this, usually involving filtering or slicing to only view specific sub-dimensions at a time. While this makes it easier to visualize specific portions of the data and removes visual confusions, the burden of keeping a global picture of the remaining, hidden data is usually left to the user or has to be visualized separately.

We propose an alternative approach, in which we aim to visualize the entire data simultaneously using a hierarchical view that mimics how multidimensional arrays are abstracted in most high-level programming languages. The two innermost

²<https://youtu.be/HPQafJeOsCI>



(a) Four-dimensional container for 3D convolution weights $w \in \mathbb{R}^{C_{out} \times C_{in} \times K_y \times K_x}$. (b) Distribution of the number of accesses in a 3D convolution without padding, which maps 3-channel, 9×9 inputs to 2-channel, 6×6 outputs. (c) Showing related accesses to A and B for accesses to $C[3, 0]$, $C[3, 1]$, and $C[3, 2]$ in an outer vector product $C = A \otimes B$.

Fig. 4: Multi-dimensional data containers, and screenshots of access pattern visualizations in the parameterized view.

dimensions are laid out in a 2D grid, and those are nested in alternating horizontal and vertical 1D grids for the remaining higher dimensions. For one dimensional data containers, a simple 1D grid is used. An example of this can be seen in Fig. 4a, where the weight tensor $w \in \mathbb{R}^{C_{out} \times C_{in} \times K_y \times K_x}$ for a 3D convolution is shown. While this representation rapidly grows in space with more added dimensions, it works well in this parameterized setting, due to the small values expected for each individual dimension.

C. Access Pattern Simulation

With both data sizes and parallel region parameters specified, we can derive the exact access pattern for each data container in the graph. To do this, we exploit the fact that programs translated to the SDFG intermediate representation carry an annotation of exactly what data subsets are being accessed by each computation in the form of a symbolic expression. This expression can traditionally not be evaluated statically without dependent symbol values such as loop iteration variables. In the parameterized graph, where parallel regions have their bounds fixed, we can perform an iteration space simulation to evaluate these symbolic expressions and derive the exact data accesses performed by each computation in the graph. By extension, this determines the exact access pattern for each data container.

The resulting access pattern can be played back using a variable speed animation, which highlights the exact individual elements or memory locations in each data container accessed at that specific time-step of the simulation. Alternatively, the time dimension can be flattened, summing up the number of accesses for each element in each data container, and showing the resulting distribution using a colored heatmap, where elements with a higher number of accesses are colored red,

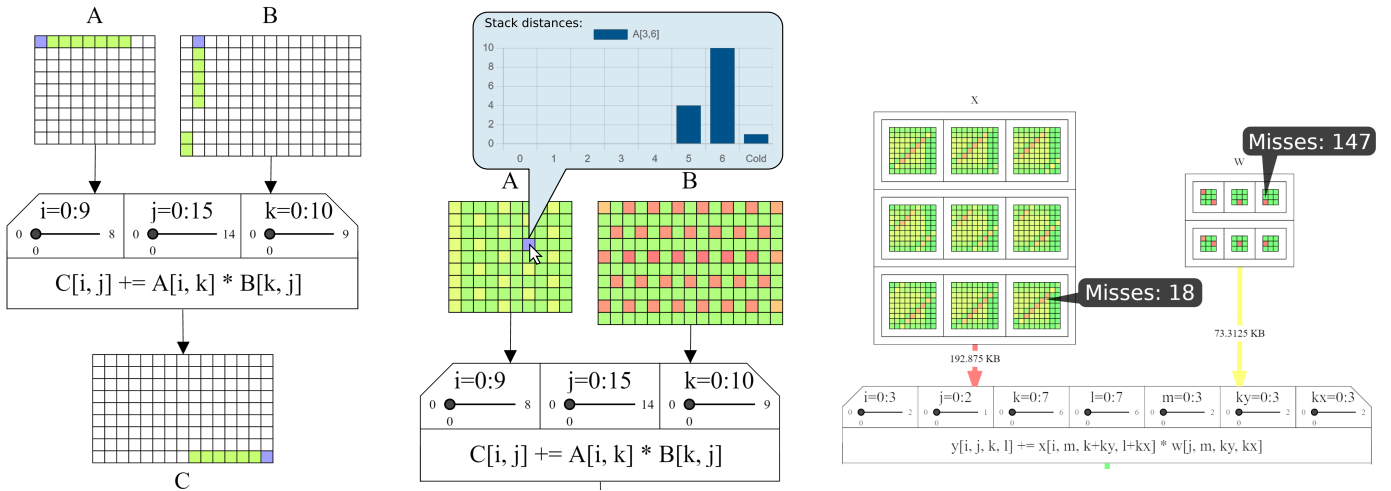
and lower numbers are represented in green. The exact number of accesses can be viewed using a tooltip when hovering the corresponding data elements. An example of the flattened time dimension using a heatmap can be seen in Fig. 4b, where the access pattern of a 3D convolution is shown with tooltips superimposed.

The same information can be used to derive and visualize data accesses related to other accesses, based on whether they occur in the same computations. For example, in the case of an outer product calculation $C[i, j] = A[i] * B[j]$, where $i \in [0, 2]$ and $j \in [0, 3]$, an access to $B[0]$ is associated to accesses of $C[i, 0]$ and $A[i]$, for all $i \in [0, 2]$. The engineer can click one or more memory locations, stacking the number of related accesses to form a corresponding access heatmap, which helps analyze for potential replication or loop tiling opportunities. An example of this can be seen in Fig. 4c.

D. Data Layout and Spatial Locality

Apart from the access pattern, the second important factor that determines how efficiently data locality is being leveraged, is the physical data layout. This is usually opaque to the engineer and needs to be derived from the alignment, offsets, and padding used by the compiler or a specific data structure. However, this information is crucial in determining how efficiently the cache is used to exploit *spatial locality*.

To expose this information to the user, we provide an overlay that visualizes which elements are adjacent to one another in memory, by highlighting which data elements are pulled into the cache alongside a specified other element. To determine this, an engineer must only provide the cache line (cache block) size in bytes for the target architecture. The remaining information, like individual element sizes, alignment, offset, and padding, can all be extracted from



(a) Visualizing data layouts by highlighting spatial locality in the form of cache lines. (b) Distribution of median reuse distance per element, with a detailed histogram breakdown (top). (c) Estimated cache misses and physical data movement overlay based on reuse distances and cache line sizes.

Fig. 5: Screenshots from our tool, visualizing physical data layouts, reuse distances, and estimated physical data movement.

the program’s intermediate representation. The user can then select memory elements, and the overlay highlights any other memory elements that fit into the same cache line, effectively exposing the physical data layout to the user and providing a guidance for exploiting spatial locality.

Fig. 5a shows the parameterized matrix multiplication of two matrices $A \in \mathbb{R}^{9 \times 10}$ and $B \in \mathbb{R}^{10 \times 15}$, where each value is 4 bytes in size and the cache line size is set to 64 bytes. By selecting $A[0, 0]$, $B[0, 1]$, and $C[8, 14]$, the visualization highlights all elements pulled into cache together with these accesses in green, revealing that A and C are stored in row-major format, while B is stored in column-major ordering.

E. Stack Distance and Temporal Locality

The obtained exact data access patterns can further be used to expose *temporal locality*. To do this, we calculate a metric called the **stack distance** for each data element, which is defined as the number of accesses to *unique* addresses made since the last reference to the requested data element [31]. We use the stack distance at a cache line granularity, meaning that for each reference to a data element, all other elements in the same cache line are also referenced, resetting their stack distance to zero. If an element has not been referenced yet, its stack distance is set to infinity.

The distribution of stack distances for each individual memory element can be visualized in-situ using a heatmap. The engineer can choose whether the heatmap should visualize the distribution of the minimum, maximum, or median stack distances for each element. To analyze this data at a finer granularity, an additional histogram with all the calculated stack distances over time is plotted in the details panel when an individual memory element or container is selected. Fig. 5b shows a heatmap for median reuse distances on the input matrices to the previously used matrix multiplication, using

a cache line size of 32 bytes. Selecting the element $A[3, 6]$ plots a histogram that reveals the exact distribution, and shows that this element was accessed once without having previously been moved to the cache, by listing one cold miss.

F. Cache Misses and Physical Data Movement

With a now complete prediction of the program’s data access behavior, including both temporal and spatial locality, we can construct a rough prediction on the number of cache hits and misses [32]. This in turn can be used to refine the abstract, logical memory movement volume shown on memory movement edges in the global, parametric view. To count the number of predicted cache misses, there are three types of misses that should be considered.

a) **Cold miss**: A cold miss happens when a memory address is accessed for the first time without having been referenced before via spatial locality (i.e., in the same cache line as a different referenced address). We count a cold miss for every access where the stack distance is infinite.

b) **Capacity miss**: A capacity miss is encountered when a memory address is accessed after it has been evicted from the cache because of the eviction strategy after the cache or associated cache set has become full. Assuming an LRU or LRU-derived eviction strategy, we count a capacity miss for every access where the stack distance is above a certain threshold value. To model different cache sizes or degrees of cache associativity, the user can modify this threshold on-the-fly through the user interface. It can further be used to adjust for the fact that the simulated data sizes are not equal to the expected data sizes in the target environment, which would make calculations based on the total cache size unrealistic.

c) **Conflict miss**: A conflict miss occurs when a memory address is accessed after it has been evicted due to a conflict, meaning a different element and its cache line was mapped

TABLE I: Case Study Benchmark Results

Application		System					
		Piz Daint (Supercomputer)		High-Performance Workstation		Consumer Hardware	
		Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup
BERT encoder	Baseline	8254.13	1.0×	13670.66	1.0×	8959.78	1.0×
	1st set of loop fusions	2273.37	3.6×	2443.13	5.6×	1427.03	6.3×
	2nd set of loop fusions	1163.36	7.1×	452.54	30.2×	336.84	26.6×
Horizontal diffusion	Baseline	667.54	1.0×	449.63	1.0×	358.39	1.0×
	Best NPBench CPU result	31.65	21.1×	18.43	24.4×	41.33	8.7×
	Hand-tuned using our tool	4.41	151.4×	3.26	138.0×	7.00	51.2×

to the same location in the cache. This type of miss can only appear in set-associative or direct mapped caches. Because the physical addresses of data containers depend on runtime conditions and their respective sizes, counting this type of cache miss in a small-scale, parameterized setting may introduce significant complexity in interpreting and generalizing the results. To combat this, we assume a *fully-associative* cache and do not count conflict misses. McKinley and Temam [33], and Beys and D’Hollander [32] show that this gives a good prediction for the total number of cache misses in low set-associative or direct mapped caches, since most cache misses can be attributed to capacity, and only a minority of misses are due to conflicts.

The resulting distribution and number of cache misses can be directly visualized using the same heatmap technique, or can be used to derive a rough prediction for the amount of physical data that needs to be moved to and from main memory. We can obtain this estimate for each data movement edge by multiplying the number of misses in both the edge’s source and destination nodes, with the number of bytes per cache line. The resulting value can be used to refine the heatmap on the data movement overlay, as shown in Fig. 5c, where the number of estimated cache misses and data movement volume are visualized on top of the input and weight tensors to a 3D convolution, using a cache line size of 64 bytes and 8 byte data values. This enables informed decisions about where local replication or changes to the data layout could be beneficial, or where to apply techniques like loop tiling, fusion, or reordering.

VI. CASE STUDIES

To demonstrate the use of our visualization in a realistic scenario, we walk through the performance analysis and optimization procedure of two real-world HPC applications. We evaluate each application on three systems:

- The Swiss National Supercomputing Center’s *Piz Daint* supercomputer, which is a cluster of 5,704 Cray X50 nodes with a 12-core Intel Xeon E5-2690 v3 CPU at 2.6 GHz and 64 GB of RAM each. We run each application on a single Cray X50 node.
- A high-performance workstation with two 16-core Intel Xeon Gold 6130 CPUs at 2.1 GHz and 1.5 TB of RAM.
- A consumer-grade system with a 10-core Intel i9-7900X CPU at 4.5 GHz and 32 GB of RAM.

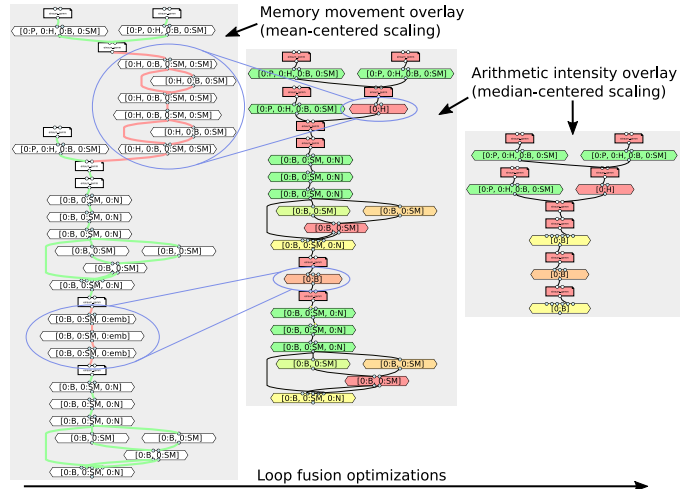


Fig. 6: Annotated snapshots of our global view, showing the BERT encoder layer at different stages of optimization.

For all experiments we use Python 3.8 with DaCe version 0.13 and GCC 8.3. Each experiment is run 100 times and we report the median runtime.

A. BERT Transformer

Machine learning is particularly data intensive, making data movement costs harder to analyze. With this case study, we demonstrate how a program’s global data movement scheme can successfully be optimized with the help of information exposed through our visualization’s global view. For this purpose we select the encoder layer from the natural language model BERT [6]. This type of Transformer [34] is a widely used neural network, where even pre-trained models take hours to tune in large-scale distributed environments. We use a NumPy [35] implementation of this application as our baseline, using Intel MKL [36] to accelerate linear algebra operations. The input parameters are selected to match the ones used in the original BERT publication [6] (BERT_{LARGE}), with a batch size $B = 8$, $H = 16$ attention heads, an embedding size $I = 1024$, an input/output sequence length $SM = 512$, an intermediate size $emb = 4096$, and a projection size $P = \frac{I}{H} = 64$.

Despite the large graph generated by this program, turning on heatmap overlays immediately helps identify some problems. The logical data movement heatmap with scaling around the mean (shown in Fig. 6, left) reveals two distinct

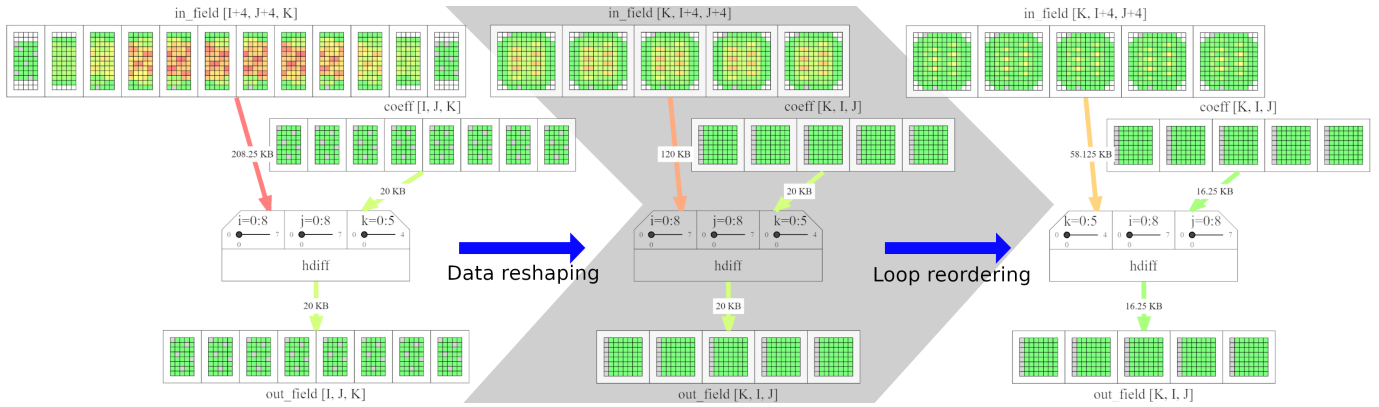


Fig. 7: Screenshots of the local view, showing the number of cache misses and physical data movement for horizontal diffusion through the optimization process.

series of edges highlighted in red. This indicates that large amounts of data are being moved between individual graph nodes. Clicking those nodes reveals that they represent parallel loops over similar loop bounds, which we can combine via *loop fusion*, removing the data movement between them. This results in a new graph where these high-volume data movement edges are not present anymore (Fig. 6, center). This optimization already provides a significant speedup of between $3.6\times$ and $6.3\times$ over the baseline implementation, as shown in Table I.

Using the arithmetic intensity overlay with median-centered scaling, as shown in Fig. 6 (center), we can see a few computation nodes with a relatively low arithmetic intensity highlighted in green. By clicking these nodes, the details panel again reveals that they represent parallel loops which can be fused together, resulting in the much smaller program graph on the right side of Fig. 6, where the number of computation nodes with low arithmetic intensity is visibly reduced.

Using only information directly exposed through our visualization, we get **speedups of $7.1\times$ to $30.2\times$** depending on the target system, as shown in Table I.

B. Horizontal Diffusion

To demonstrate the analysis capabilities of our visualization’s close-up, local view, we walk through the tuning process of horizontal diffusion (or *hdiff*), which is a stencil composition that plays an important role in weather and climate models [37]. We take an implementation of this application from the high-performance NumPy [35] benchmarking suite NPbench [38]. The program has three free parameters I , J , and K and operates on two inputs $\text{in_field} \in \mathbb{R}^{I+4 \times J+4 \times K}$ and $\text{coeff} \in \mathbb{R}^{I \times J \times K}$, and an output $\text{out_field} \in \mathbb{R}^{I \times J \times K}$. We evaluate the application using the same parameter sizes used in the NPbench paper [38], with $I = J = 256$ and $K = 160$, which represent a typical per-node scenario in a cluster running weather and climate simulations [37]. The default NumPy implementation in NPbench serves as our baseline.

To start the analysis process in the local view, we parameterize the program with smaller input parameters $I = J = 8$

and $K = 5$, which represents a $\frac{1}{32}$ scaled version of the full-sized program. These parameters can be chosen arbitrarily, but scaling all input parameters down by a common factor ensures that all analysis parameters maintain the same size relative to each other. The full resulting graphical representation of the program, which can be represented as one 3-dimensional loop operating on two input and one output data containers, can be seen in Fig. 7 (left). We further set the cache line size to 64 bytes to reflect our target architecture, and since the program operates on double-precision floating point values, the value size is 8 bytes.

By moving any one of the loop sliders, or by hovering over or clicking on any output data element, the corresponding access pattern on the input data containers is shown. This access pattern shows that each loop iteration accesses a number of elements from the in_field input, as shown at the top of Fig. 8a. If these accesses occur far apart in memory, this leads to poor spatial reuse, since a large number of separate cache lines would have to be accessed for one operation. Clicking any data element with the cache line overlay enabled exposes the data layout of the in_field container. The highlighted cache line shows that the container uses row-major ordering, which implies that dimension $J + 4$, over which the accesses are spread out, is a non-contiguous dimension, and that the accesses are spread out in memory with poor spatial locality. Reshaping in_field from $[I+4, J+4, K]$ to $[K, I+4, J+4]$ visibly improves the access pattern, with all accesses now occurring much closer to each other in memory, as shown in the bottom half of Fig. 8a. This optimization step further comes with a visible reduction in the number of cache misses, and consequently almost halves the amount of data being requested from main memory for in_field , as seen in Fig. 7.

While the accesses per loop iteration are now physically closer together, by playing back the access pattern animation or moving the sliders, the overlay reveals a new problem. The innermost loop, $k \in [0, 4]$, now iterates over a non-contiguous dimension, as shown in the top of Fig. 8b. A simple re-ordering of the loops, such that $k \in [0, 4]$ becomes the outer-most loop, addresses this problem and improves the access pattern,

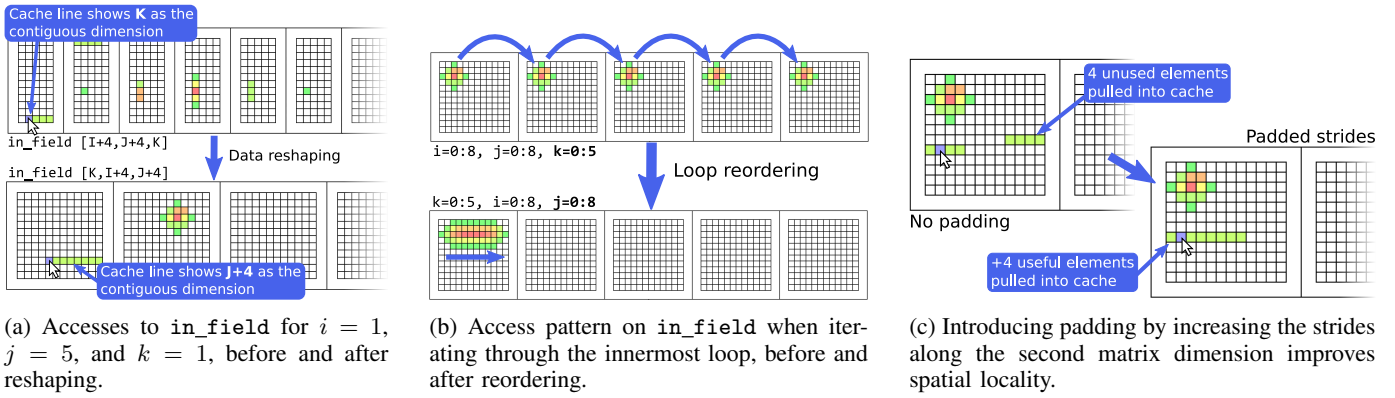


Fig. 8: Analysis of horizontal diffusion during individual steps of the tuning process.

as demonstrated in Fig. 8b (bottom). Fig. 7 shows a further reduction of both cache misses and the number of data bytes moved following this optimization step.

Finally, the cache line visualization shown in Fig. 8c (top) highlights that accesses to some of the first elements in individual rows are located on cache lines that wrap around from the previous row. Knowing the access pattern of an individual loop iteration in horizontal diffusion (bottom of Fig. 8a), an iteration accessing these elements would simultaneously require accesses to more elements located in the same row. The elements residing in the previous row are consequently unused and pollute the cache. By increasing the strides along the second data dimension to a number divisible by the cache line size, we can introduce post-padding to align each individual data row to the cache line. As shown in the bottom half of Fig. 8c, this improves spatial locality by pulling elements into the cache that will be accessed in the same loop iteration.

We benchmark the optimized application and observe between a $51.2\times$ and $151.4\times$ improvement over the original NumPy implementation, depending on the target system, as shown in Table I. This speedup through optimizations based on information obtained from our visualization, was attained *without requiring profiling* or the analysis of hardware counters, and even **outperforms the fastest CPU-based result measured in NPbench [38] by between $5.7\times$ and $7.2\times$** , depending on the target system.

We provide three short supplementary videos demonstrating the analysis process for *hdiff* when identifying the suboptimal memory layout³, loop order⁴, and improper alignment⁵.

VII. RELATED WORK

Several other works exist with the goal of assisting in the performance analysis workflow by either visualizing performance data, exploring data reuse with data-centric analysis, or obtaining performance insights through simulation. Many approaches combine a subset of these techniques to

provide visualization-augmented, data-centric analysis, but require lengthy program executions to obtain instrumentation data. Even simulation- and modeling-based approaches often rely on memory traces obtained through program execution. Existing approaches further often show performance metrics and observations separate from the program definition, requiring context switching between analysis and subsequent optimization steps.

Our solution provides a holistic approach that attempts to remove the need for program executions entirely, enabling a more interactive performance optimization process. We visualize memory access and reuse behavior on multiple scales, directly in-situ on an intermediate program representation that can be used for subsequent optimizations. An overview of the related works can be seen in Fig. 9.

a) Performance Visualization: Many tools have been created to expose a program’s performance characteristics by aggregating and visualizing profiling and tracing data. Zinsight [49] visualizes large event traces with a set of independent views that extract event statistics and patterns. Event traces can also be explored using JumpShot [50], VAMPIR [52], or Paraver [53], which offer different approaches to navigating these large traces. JumpShot, VAMPIR, and Paraver are all part of the larger TAU parallel performance analysis system [62], which further includes PerfExplorer [57], a tool to summarize parallel profiles with a set of 2D and 3D graph visualizations. In a similar manner, Projections [56] visualizes profile data in summary graphs, detailed timeline views, and other graphs, employing the help of overviews to facilitate easier navigation. Projections further allows comparing profiles gathered from different runs side-by-side. The use of overviews to help navigate large profiles can also be seen in Extravis [12], which shows call relations in a circular bundle view. Scalasca [24] and its visualization component CUBE [55] have also been created with the goal of visualizing profiles obtained from programs using many thousands of processors. PerformanceHat [58] and Beck et al. [59] visualize execution time measurements extracted from profiles directly in the code editor, using icons, color highlights, and tooltips in the source code. Some visualizations, such as Trevis [47]

³<https://youtu.be/rYNcNnhZBLE>

⁴https://youtu.be/hZv3U_tlgII

⁵https://youtu.be/hM6VT_j06C8

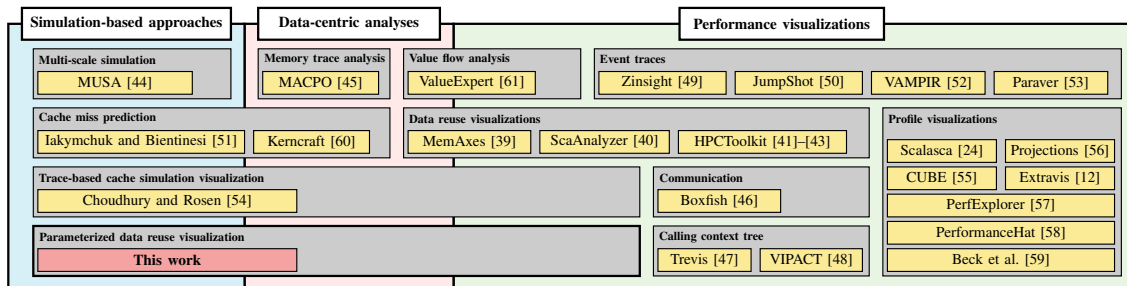


Fig. 9: Related work overview.

and VIPACT [48], focus on extracting and exploring full calling context trees from profile traces, while others, e.g., Boxfish [46], extract communication behavior and visualize that on the network topology.

b) Data-Centric Analysis: Solutions which focus specifically on data movement and reuse analysis include MACPO [45], which obtains memory traces and uses those to compute access behavior metrics for source-level data structures in C, C++, and FORTRAN, using cache models geared towards execution on multi-core chips. ValueExpert [61] constructs value flow graphs for GPU applications to help with detecting inefficient value-related patterns and guiding subsequent optimization decisions. MemAxes [39] facilitates analysis by visualizing memory performance obtained from traces, and performing a scoring of obtained performance metrics to guide the user’s attention. ScaAnalyzer [40] and other augmentations [41], [42] made to the HPCToolkit [43] architecture support data-centric profiling of parallel programs, and attribute and visualize the recorded metrics to pinpoint scaling losses and other performance bottlenecks due to memory access behaviors.

c) Simulation-Based Approaches: Other works based on performance simulation predict performance behavior to avoid runtime-based analyses, but they typically require memory traces obtained through execution for their simulations. MUSA [44] offers a multi-scale simulation approach to analyze inter-node communication and intra-node microarchitecture interactions. Iakymchuk and Bientinesi [51] have constructed a performance model that can accurately predict cache misses for fundamental linear algebra operations on Intel and AMD processors, and have managed to combine individual models to predict the performance of BLAS subroutines like matrix factorization. Kerncraft [60] follows a similar direction by constructing roof line and execution-cache-memory models for loop nests in stencil codes. Choudhury and Rosen [54] created an animated visualization that shows memory transactions based on a memory reference trace, and uses a cache simulator to visualize data elements according to their simulated cache location.

VIII. DISCUSSION

The proposed visualization is designed to be flexible and extensible. In the following, we discuss potential extensions, advantages, and limitations in the approach.

a) Cache Model: Our tool performs a general-purpose cache miss estimation based on the simulated access patterns. However, this estimation is separate from the visualization itself. Cache miss predictions for specific architectures could be derived from the simulated access patterns using different, more hardware-specific back-ends, such as Kerncraft [60], while leveraging the same visual exploration and analysis methods demonstrated in our visualization.

b) Remote Analysis: By using Visual Studio Code [5] as the basis for the implementation of our visualization tool, we can leverage the remote development feature in the code editor to perform analyses and optimizations directly on target machines, such as the compute or login nodes of supercomputers. This further increases the interactivity of the optimization workflow, by providing intuitive visual analyses even for remote scenarios.

c) Program Parameterization: Since data access patterns for regular programs do not depend on specific values of input parameters, finding good parameters to analyze a program in the local view is a matter of choosing what is easiest for the user to observe data access and reuse behavior. A good strategy is to keep these parameters small, because this keeps simulation times short when transitioning to the local view, and makes analysis easier by limiting the visual search space and cognitive load required. While the exact impact of individual optimizations may differ between the small-scale, parameterized setting and the full-scale scenario, our case study results demonstrate that the general insights obtained in the small-scale setting are helpful in uncovering key performance problems that transfer to the full-scale scenario.

This visualization approach could also be used to simulate and analyze the full-sized parameters. However, this would significantly increase simulation times, and would require aggregating multiple data elements in one visual tile to avoid making the visualization harder to interpret.


d) Limitations: A key limiting factor in our approach is in the analysis of dynamic or irregular programs. For this class of programs, exact data movement and access information cannot usually be determined statically or through small-scale, parameterized simulations. However, the global and local visualization techniques employed in our tool can similarly be used to analyze and explore traditional instrumentation data for such applications, allowing for a comparable, though less interactive optimization workflow.

IX. CONCLUSION

We have implemented a performance visualization tool in Visual Studio Code that exposes critical performance characteristics to the user, which are generally not directly visible in traditional source code and require closer analysis, such as data layout and movement, and spatial or temporal locality. By leveraging a combination of static dataflow analysis and small-scale simulations, our approach avoids costly profiling or instrumentation runs that slow down the tuning process. In combination with visualizing results in-situ, directly on a graphical program representation, this facilitates a more interactive and streamlined optimization process.

Two case studies on the optimization of real-world applications demonstrate the effectiveness of our visualization in both global data movement reduction, and fine-grained data reuse optimizations. Our tool demonstrates the ability to inform impactful tuning decisions on multiple levels, enabling optimization decisions that lead to a speedup of up to $7.2\times$ compared to the state of the art on an HPC benchmark. With data movement optimizations increasingly becoming a crucial part of performance tuning in HPC applications, we hope our approach to an interactive analysis process inspires further work on streamlined performance engineering processes.

ACKNOWLEDGMENTS

This project received funding from the European Research Council (ERC)  grant PSAP, grant agreement No. 101002047, and the European Union's Horizon Europe programme DEEP-SEA, grant agreement No. 955606. P.S. and T.B.N. are supported by the Swiss National Science Foundation (Ambizione Project No. 185778). The authors also wish to acknowledge the Swiss National Supercomputing Centre (SCS) for access and support of the computational resources.

REFERENCES

- [1] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, 1998.
- [2] R. Dennard, F. Gaensslen, W.-N. Yu, L. Rideout, E. Bassous, and A. Le Blanc, "Design of Ion-Implanted Small MOSFET 'S Dimensions with Very," *IEEE Journal of Solid State Circuits*, vol. 9, no. 5, pp. 257–268, 1974.
- [3] A. Tate, A. Kamil, A. Dubey, A. Grobinger, B. Chamberlain, B. Goglin, H. Edwards, C. Newburn, D. Padua, D. Unat, E. Jeannot, F. Hannig, G. Tobias, H. Ltaief, J. Sexton, J. Labarta, J. Shalf, K. Fuerlinger, K. O'Brien, L. Linardakis, M. Besta, M.-C. Sawley, M. Abraham, M. Bianco, M. Pericas, N. Maruyama, P. Kelly, P. Messmer, R. Ross, R. Ciedat, S. Matsuoka, T. Schulthess, T. Hoefler, and V. Leung, "Programming Abstractions for Data Locality," Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (United States), Tech. Rep., 11 2014. [Online]. Available: <https://www.osti.gov/servlets/purl/1172915/>
- [4] D. Unat, A. Dubey, T. Hoefler, J. B. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas, "Trends in Data Locality Abstractions for HPC Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.
- [5] Microsoft, "Visual Studio Code - Code editing. Refined." <https://code.visualstudio.com/>, [Online; accessed 10-March-2021].
- [6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, no. M1m, 10 2019. [Online]. Available: <http://arxiv.org/abs/1810.04805>
- [7] T. Hoefler, W. Gropp, M. Snir, and W. Kramer, "Performance modeling for systematic performance tuning," *State of the Practice Reports, SC'11*, 2011.
- [8] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173. [Online]. Available: <http://link.springer.com/10.1007/978-3-642-11261-4>
- [9] NVIDIA Corporation, "NVIDIA CUDA Toolkit Documentation - nvprof," <https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview>.
- [10] Intel, "Intel oneAPI - Intel VTune Profiler," <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html>, [Online; accessed 10-March-2022].
- [11] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "gprof: a Call Graph Execution Profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 6 1982. [Online]. Available: <https://dl.acm.org/doi/10.1145/872726.806987>
- [12] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen, "Understanding execution traces using massive sequence and circular bundle views," *IEEE International Conference on Program Comprehension*, pp. 49–58, 2007.
- [13] S. Matwin and T. Pietrzykowski, "PROGRAMPH: A preliminary report," *Computer Languages*, 1985.
- [14] T. Ben-Nun, J. De Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2019.
- [15] J. Kodosky, "LabVIEW," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, 2020.
- [16] A. Calotoiu, T. Ben-Nun, G. Kwasniewski, J. d. F. Licht, T. Schneider, P. Schaad, and T. Hoefler, "Lifting C Semantics for Dataflow Optimization," *arXiv*, vol. 1, no. 1, 12 2021. [Online]. Available: <http://arxiv.org/abs/2112.11879>
- [17] Atom, "A hackable text editor for the 21st Century," <https://atom.io/>, [Online; accessed 10-March-2021].
- [18] Google, "Google Maps," <https://www.google.com/maps/>, [Online; accessed 26-March-2021].
- [19] D. Nekrasovski, A. Bodnar, J. McGrenere, F. Guimbretière, and T. Munzner, "An evaluation of pan & zoom and rubber sheet navigation with and without an overview," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, vol. 1. New York, NY, USA: ACM, 4 2006, pp. 11–20. [Online]. Available: <https://dl.acm.org/doi/10.1145/1124772.1124775>
- [20] M. Harward, W. Irwin, and N. Churcher, "In Situ Software Visualisation," in *2010 21st Australian Software Engineering Conference*. IEEE, 2010, pp. 171–180. [Online]. Available: <http://ieeexplore.ieee.org/document/5475058/>
- [21] A. G. Landge, J. A. Levine, A. Bhatlele, K. E. Isaacs, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2467–2476, 12 2012. [Online]. Available: <http://ieeexplore.ieee.org/document/6327252/>
- [22] A. Bhatlele, T. Gamblin, K. E. Isaacs, B. T. N. Gunney, M. Schulz, P.-T. Bremer, and B. Hamann, "Novel views of performance data to analyze large-scale adaptive applications," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2012, pp. 1–11. [Online]. Available: <http://ieeexplore.ieee.org/document/6468450/>
- [23] S. Moreta and A. Telea, "Visualizing Dynamic Memory Allocations," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 6 2007, pp. 31–38. [Online]. Available: <http://ieeexplore.ieee.org/document/4290697/>
- [24] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. n/a–n/a,

2010. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/cpe.1556>
- [25] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From Performance Report Explorer to Performance Analysis Tool," *Procedia Computer Science*, vol. 51, no. 1, pp. 1343–1352, 2015. [Online]. Available: <http://dx.doi.org/10.1016/j.procs.2015.05.320>
- [26] Jülich Supercomputing Centre, "Cube GUI User Guide: Advanced Color Map Plugin," <https://apps.fz-juelich.de/scalasca/releases/cube/4.6/docs/guide/html/AdvancedColorMapPlugin.html>, [Online; accessed 14-March-2021].
- [27] K. Moreland, "Why we use bad color maps and what you can do about it," *Human Vision and Electronic Imaging 2016, HVEI 2016*, pp. 262–267, 2016.
- [28] D. Borland and R. M. Taylor, "Rainbow color map (still) considered harmful," *IEEE Computer Graphics and Applications*, vol. 27, no. 2, pp. 14–17, 2007.
- [29] Y. Liu and J. Heer, "Somewhere over the rainbow: An empirical assessment of quantitative colormaps," *Conference on Human Factors in Computing Systems - Proceedings*, vol. 2018-April, pp. 1–12, 2018.
- [30] B. Shneiderman, "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations," in *The Craft of Information Visualization*. Elsevier, 2003, pp. 364–371. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/B9781558609150500469>
- [31] E. G. Coffman and P. J. Denning, *Operating systems theory*. prentice-Hall Englewood Cliffs, NJ, 1973, vol. 973.
- [32] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, pp. 617–662, 2001. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.2405>
- [33] K. S. McKinley and O. Temam, "Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks," *ACM Transactions on Computer Systems*, vol. 17, no. 4, pp. 288–336, 1999.
- [34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, no. Nips, 6 2017. [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [35] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 9 2020. [Online]. Available: <http://www.nature.com/articles/s41586-020-2649-2>
- [36] Intel, "Intel oneAPI Math Kernel Library," <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html>.
- [37] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 4, 2021.
- [38] A. N. Ziogas, T. Ben-Nun, T. Schneider, and T. Hoefler, "NPBench: A Benchmarking Suite for High-Performance NumPy," in *Proceedings of the ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 6 2021, pp. 63–74. [Online]. Available: <https://dl.acm.org/doi/10.1145/3447818.3460360>
- [39] A. Gimenez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P. T. Bremer, and B. Hamann, "MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 7, pp. 2180–2193, 2018.
- [40] X. Liu and B. Wu, "ScaAnalyzer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 15-20-Nove. New York, NY, USA: ACM, 11 2015, pp. 1–12. [Online]. Available: <https://dl.acm.org/doi/10.1145/2807591.2807648>
- [41] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 11 2013, pp. 1–12. [Online]. Available: <https://dl.acm.org/doi/10.1145/2503210.2503297>
- [42] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," *ACM SIGPLAN Notices*, vol. 49, no. 8, pp. 259–272, 11 2014. [Online]. Available: <https://dl.acm.org/doi/10.1145/2692916.2555271>
- [43] 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*, vol. 22, no. 6, pp. 685–701, 2010.
- [44] T. Grass, C. Allande, A. Arnejach, A. Rico, E. Ayguade, J. Labarta, M. Valero, M. Casas, and M. Moreto, "MUSA: A Multi-level Simulation Approach for Next-Generation HPC Machines," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2016, pp. 526–537. [Online]. Available: <http://ieeexplore.ieee.org/document/7877123/>
- [45] A. Rane and J. Browne, "Enhancing Performance Optimization of Multicore/Multichip Nodes with Data Structure Metrics," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, pp. 1–20, 10 2014. [Online]. Available: <https://dl.acm.org/doi/10.1145/2588788>
- [46] K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann, "Abstract: Exploring Performance Data with Boxfish," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 11 2012, pp. 1380–1381. [Online]. Available: <http://ieeexplore.ieee.org/document/6495985/>
- [47] A. Adamoli and M. Hauswirth, "Trevis: A Context Tree Visualization & Analysis Framework and its Use for Classifying Performance Failure Reports," in *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*. New York, New York, USA: ACM Press, 2010, p. 73. [Online]. Available: <http://portal.acm.org/citation.cfm?doi=1879211.1879224>
- [48] H. T. Nguyen, L. Wei, A. Bhatele, T. Gamblin, D. Boehme, M. Schulz, K. L. Ma, and P. T. Bremer, "VIPACT: A visualization interface for analyzing calling context trees," *Proceedings of VPA 2016: 3rd Workshop on Visual Performance Analysis - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 25–28, 2017.
- [49] W. De Pauw and S. Heisig, "Zinsight: A visual and analytic environment for exploring large event traces," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 143–152, 2010.
- [50] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 8 1999. [Online]. Available: <http://journals.sagepub.com/doi/10.1177/109434209901300310>
- [51] R. Iakymchuk and P. Bientinesi, "Modeling performance through memory-stalls," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 86–91, 2012.
- [52] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, 1996.
- [53] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *Proceedings of WoTUG-18: transputer and occam developments*, no. February, pp. 17–31, 1995.
- [54] A. N. M. I. Choudhury and P. Rosen, "Abstract visualization of runtime memory behavior," in *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 9 2011, pp. 1–8. [Online]. Available: <http://ieeexplore.ieee.org/document/6069452/>
- [55] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. Wylie, "Scalable collation and presentation of call-path profile data with CUBE," *Advances in Parallel Computing*, vol. 15, pp. 645–652, 2008.
- [56] L. V. Kalé, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using Projections performance analysis tool," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 347–358, 2 2006. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0167739X04002262>
- [57] K. Huck and A. Malony, "PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing," in *ACM/IEEE SC 2005 Conference (SC'05)*, vol. 2005. IEEE, 2005, pp. 41–41. [Online]. Available: <http://ieeexplore.ieee.org/document/1559993/>
- [58] J. Cito, P. Leitner, M. Rinard, and H. C. Gall, "Interactive Production Performance Feedback in the IDE," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 971–981, 2019.
- [59] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," *IEEE International Conference on Program Comprehension*, pp. 63–72, 2013.
- [60] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels," in *Tools*

for *High Performance Computing 2016*, C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Cham: Springer International Publishing, 2017, pp. 1–22. [Online]. Available: http://link.springer.com/10.1007/978-3-319-56702-0_1

- [61] K. Zhou, Y. Hao, J. Mellor-Crummey, X. Meng, and X. Liu, “ValueExpert: Exploring value patterns in GPU-Accelerated applications,” *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, no. Line 2, pp. 171–185, 2022.
- [62] S. S. Shende and A. D. Malony, “The Tau Parallel Performance System,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 5 2006. [Online]. Available: <http://journals.sagepub.com/doi/10.1177/1094342006064482>