C.A.T.S.: Memory and Control Flow Tracing for Whole-Program Performance Analysis

Philipp Schaad
philipp.schaad@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Tal Ben-Nun
talbn@llnl.gov
Lawrence Livermore National Laboratory (LLNL)
Livermore, USA

Torsten Hoefler
htor@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Performance engineering often involves localized, bottleneck-based optimization, supported by a plethora of tools. When no apparent bottlenecks exist, engineers resort to coarser whole-program optimization, consisting of data layout, sparsity, allocation strategy, and algorithmic modifications, to name a few. In this work, we aim to codify whole-program optimization by providing three global views based on a single tracing format. The format, called C.A.T.S., captures information necessary for static and runtime analysis of large applications. Instead of call stacks and function annotations, C.A.T.S. uses *control flow stacks* and *memory events* to identify common performance anti-patterns and potential optimizations. We develop interactive timeline, dataflow, and access visualizations, and implement compiler analysis passes to extract C.A.T.S. traces statically and in seconds on consumer hardware. The visualizations and analyses are demonstrated on case studies including sparse computations, hydrodynamics and climate modeling, yielding 3× memory footprint reduction, improvements in communication-computation overlap, code fusion, and data layouts.

CCS Concepts

• **Human-centered computing** → **Visualization**; • **General and reference** → *Measurement*; *Performance*.

Keywords

Tracing, Performance Visualization, Symbolic Analysis

ACM Reference Format:
Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2025. C.A.T.S.: Memory and Control Flow Tracing for Whole-Program Performance Analysis. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3712285.3759797

1 Introduction

Advances in visualization [42] have helped to improve performance optimization workflows for HPC applications. By aggregating performance data collected through instrumentation or profiling, visualizations present an abstract summary view of an application's performance characteristics. This helps performance engineers quickly identify and locate critical bottlenecks, often using more detailed, fine-grained measurements and visualizations for close-up inspection. After performing optimizations to alleviate a bottleneck, visualizations help reassess the global picture and identify the next bottleneck. Performance visualizations are thus a vital tool for such an iterative *bottleneck optimization* workflow.

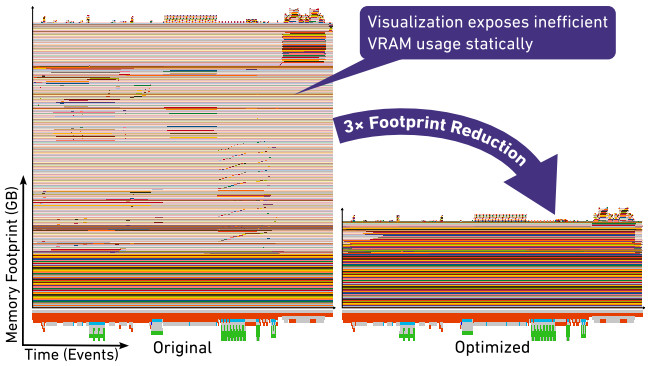


Figure 1: Memory timeline view helps reduce the number of GPUs needed for a climate model at 1km resolution by 3.5×.

In cases where a distinct bottleneck is missing but performance expectations are unmet, the optimization workflow is less well-structured and supported. Such cases typically necessitate more coarse-grained *whole-program optimizations*, such as changes to data layouts and sparsity structures, allocation strategies, or algorithmic structure. Unlike bottleneck optimizations, such broader changes typically also require a broader context to inform whether they are likely to be beneficial and can be performed safely.

To provide such context for measurements, performance visualizations frequently utilize function call stacks or call graphs [2, 33, 62, 78]. However, for many whole-program optimizations, such as data layout optimizations, this context is not fine-grained enough. The optimality of a data layout depends on all surrounding control flow, including loops and their schedules. Additionally, a layout change may improve locality for certain accesses, but may negatively affect other accesses, meaning that only highlighting problematic accesses — as is frequently the goal in bottleneck optimization — does not provide sufficient context.

In this paper, we address this challenge by introducing a trace format that tracks memory events such as allocations and accesses in the context of a detailed *control flow stack*. By capturing control flow stacks, captured traces can be used to gain a structural overview of an application and detect performance anti-patterns common in HPC applications, such as allocations inside of loops. Control flow stacks capture the repeating nature of loop-based applications typical for scientific computing, allowing repeating events to be dropped, reducing trace sizes to a minimum.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759797

Powered by this trace format, we demonstrate three performance visualizations geared towards common whole-program optimization tasks: A timeline view of memory allocations and accesses, shown in Fig. 1, helps reduce memory footprint and costly repeated allocations. A statistical memory accesses type overview helps improve data layouts and storage formats for improved data locality. A control flow and data dependency view assists in reordering, replication, or fusion and fission of subprograms, extracting task parallelism, and improving communication strategies. With a series of compiler analysis passes we demonstrate how traces can be collected statically and in seconds on consumer hardware.

A prototype implementation applied to case studies including numerical weather prediction and shock hydrodynamics demonstrates how our visualizations can guide effective whole-program optimization decisions on real-world applications written in Fortran, Python, and C++. Using our visualizations, we obtain improvements in communication strategies, yielding speedup in a distributed setting, reduced memory footprint for better scaling, and up to 2.6× **speedups** over auto-optimized codes.

In summary, our contributions are as follows.

- Identifying necessary information for common whole-program performance optimizations, specified in a trace format
- Novel visualization techniques for exploring global optimizations and finding potential performance improvements
- A set of IR analysis passes to build control flow stacks and gather memory events statically from Fortran and Python

2 Whole-Program Optimization

Whole-program performance optimization takes on many forms, requiring various different insights into specific program characteristics. The most common whole-program optimizations typically performed for HPC applications [46] can be broadly divided into four categories: improving data movement and communication, increasing coarse-grained parallelism, reducing memory footprint, and mitigating performance anti-patterns. We discuss each of these categories and the specific optimization tasks they entail, together with the program information required for them below and give an overview of them in Fig. 2.

Reduce Data Movement & Communication. Data movement and communication have become the primary causes of performance issues in HPC applications in recent years [82]. A crucial step in optimizing whole-program performance is thus often to improve the medium- and long-range data reuse and minimize the costs of communication [29, 56]. Various techniques help achieve this, but most frequently this results in code reordering, rescheduling (e.g., loop/kernel fusion and fission) [44], or introducing recomputation [79]. This is challenging without knowing about **data dependencies** between individual program parts. Additionally, identifying long-range data movement issues requires an overview over the data reuse in an application. A common metric used to measure this is the *reuse distance* [20, 67, 76], which reports the number of distinct data elements accessed between two accesses to the same data element, similar to the stack distance for LRU caches [59]. To get this information, insight into the **data accesses** performed by an application is required.

Optimization		Visualization	Required Information			
			Allocations	Accesses	Control Flow	Dependencies
Reduce Memory Footprint	Memory Pooling	Memory Timeline (§4.1)	✓	✓		
	Reuse Buffers		✓	✓		✓
Anti-Pattern Mitigation	Memory Allocation Rescheduling		✓	✓	✓	
	Change Data Layout & Storage Formats	Access Statistics (§4.2)	✓	✓†	✓†	
Reduce Data Movement	Code Reordering	Control Flow Data Dependency (§4.3)			✓	✓
	Code Replication, Fusion, & Fission				✓	✓
Increase Parallelism	Extract Task Parallelism				✓	✓

†Requires symbolic information

Figure 2: An overview of common whole-program performance optimizations, the information they require, and how the visualizations presented in this paper support them.

Increase Coarse-Grained Parallelism. If exploiting data parallelism is not enough to fully utilize the available hardware, a common optimization is to increase the amount of exploited task parallelism [18, 80]. As with exploiting data parallelism, this requires insight into the **data dependencies** between parts of an application that may be defined as separate tasks.

Reduce Memory Footprint. Especially in data-intensive HPC applications, the memory footprint of an application can be problematic [84, 90]. An excessively large memory footprint can prevent or negate the benefits of offloading to accelerators [77], or lead to communication overhead as the problem needs to be distributed across more nodes. Excessive memory consumption is not always a direct performance threat for smaller problems or individual kernels, but can lead to prohibitively large resource requirements when scaling HPC applications to larger problem sizes [7]. Often, the available resources limit the scaling that can be achieved, as we will demonstrate in Section 6.2. Whole-program optimization thus frequently includes reducing this footprint through optimizations such as memory pooling, reusing no-longer-used buffers [7], or introducing recomputation [88]. Performing these optimizations requires knowledge about when and where memory is being **allocated and de-allocated**, as well as when memory is being **accessed** or 'used'.

Mitigate Performance Anti-Patterns. Finally, applications can contain performance *anti-patterns* that inhibit performance at the hardware level or prevent crucial compiler optimizations. One of the most common anti-patterns for HPC applications is the use of indirect or non-contiguous loop-based data accesses. Scientific computing algorithms often have complex loop structures where the required iteration order may not align well with physical data layouts [14, 47, 63]. Similarly, the prevalence of sparse computations often leads to the use of indirect data accesses, which, especially when offloaded to accelerators, can cause uncoalesced accesses and low utilization. Addressing this often necessitates changes to sparsity structures [27, 68] or fundamental algorithmic changes [17, 30, 66]. Detecting such anti-patterns requires knowledge about **data accesses** in the context of the **control flow** around them.

Additional common anti-patterns include allocations or the use of data-dependent branching inside loop nests [32, 83], the latter causing performance-degrading thread-group divergence in the context of GPU applications. Detecting either of these anti-patterns requires an overview of the application control flow, while the former additionally requires an overview of **allocations** in the context of that control flow.

3 Control Augmented Trace Structure – C.A.T.S.

Based on the aforementioned requirements, we specify the Control Augmented Trace Structure (C.A.T.S.), which is able to capture and provide this information. The overview in Fig. 2 shows that the required information falls into three categories: information about memory usage (allocations and accesses), control flow, and data dependencies. Since data dependencies can be obtained efficiently through various program analysis techniques [6, 10, 25, 72], the trace format must primarily capture *memory usage* and *control flow* information.

3.1 Memory Events

A C.A.T.S. trace can capture the necessary memory usage information with two types of *memory events*: memory allocation events, and memory access events. Memory *allocation events* occur each time memory is being allocated or de-allocated. An allocation event records a unique identifier for each allocation and corresponding de-allocation, together with the number of bytes and the type of memory being allocated. A memory *access event* occurs on each access to allocated data. Each access event records a unique identifier of the allocated data being accessed, which corresponds to the identifier recorded by an earlier allocation event. Memory access events further record whether the access reads or writes memory, the offset of the access to the base address of the allocated memory, and the number of bytes written or read.

3.2 Control Flow Stack

To capture control flow information, tracing and profiling tools often utilize call stacks as a proxy [2, 9, 33, 62, 78]. Tracking function invocations and constructing a call graph, or calling context tree, contextualizes performance metrics and gives a structural overview of the application. However, HPC applications are often dominated by loops, which are core to many performance anti-patterns, including repeated allocations or, in the context of GPUs, data-dependent branching that causes divergence. Detecting these anti-patterns requires a more fine-grained structural view of the application control flow that includes loops and branching.

To capture this information, each C.A.T.S. event is associated with a detailed *control flow stack* that captures the exact control flow context at the time of the event. The control flow stack contains a reference not only to each function in the call leading to the event, but also each surrounding loop, conditional branch, or parallel region. Together with each reference, information about each construct is stored, such as function parameters or loop iterators. This gives a global picture of the control flow leading to each measurement or event, which helps detect common anti-patterns such as allocations inside of loops.

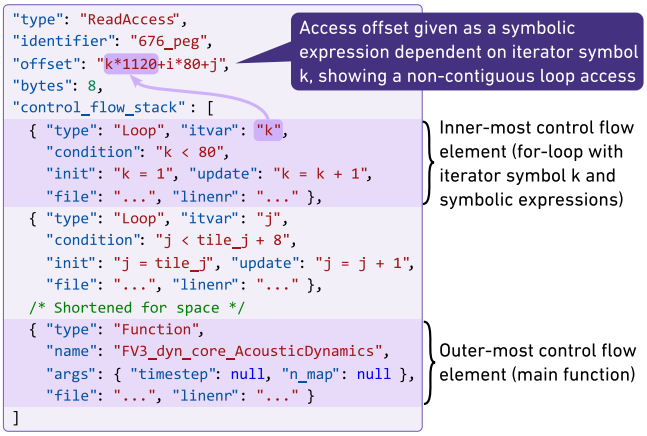


Figure 3: Example of a C.A.T.S. memory access event, including a control flow stack with symbolic offset information.

3.3 Recording C.A.T.S. Traces

Program tracing is typically performed in one of two ways: either at program runtime through instrumentation [2], or through symbolic analysis [24, 38]. Recording C.A.T.S. traces through either method has its advantages and disadvantages. Runtime-based tracing has the benefit of recording empirical, concrete program behavior, including any effects of compiler optimizations or data-dependent execution. Events gathered during runtime may also uncover effects such as pointer aliasing, something that is challenging to handle with symbolic program analyses. The downsides of runtime-based tracing are the time and space complexity involved. HPC applications may take a long time to execute, require specific hardware to run, and generated traces may become prohibitively large.

Control flow stacks allow us to address the space complexity by associating each C.A.T.S. event with a detailed control flow context. Allocations are tracked with the intention of detecting anti-patterns and obtaining the maximum memory footprint, so we can capture *unique* instances, in which an allocation occurs based on a unique control flow stack profile. Control flow stacks encode the necessary information to extrapolate if an instance is executed repeatedly.

Similarly, since data accesses are recorded to understand where buffers may be reused or pooled, or how data layout changes may affect data locality, only unique instances of each data accessing instruction needs to be recorded. For example, assuming the memory access event shown in Fig. 3 has been recorded, any subsequent firing of the same event, i.e., the same identifier with the same offset and numbers of bytes read, where the control flow stack matches the already recorded event does not need to be recorded. This effectively means that the size of a C.A.T.S. trace **correlates with the size of a program**, i.e., the number of source lines of code, and does not grow for a longer-running, heavily loop-based application.

Detecting data access anti-patterns, specifically loop-based non-contiguous accesses or indirect accesses, can be difficult with runtime traces. Identifying these accesses and classifying them correctly requires establishing a semantic connection from an access to surrounding loops, or to other accesses. Specifically, it must be possible to determine how the offset from a given base address associated with a data access depends on its surrounding loop iterators, or if it depends on a different data access. This is easily possible with *symbolic* trace construction by capturing the offsets

with allocations are accessed as symbolic expressions. Such a symbolic offset expression may contain various constants and symbols, where some symbols may be the identifier of a different data access, exposing indirect accesses, or may correspond to loop iteration variables. Additionally capturing the start, stride, and end of loops in the control flow stack as symbolic expressions, where possible, shows how loops surrounding a data access influence the offset, consequently exposing possible non-contiguous accesses. An example of a C.A.T.S. memory access event with symbolic offset and loop information is shown in Fig. 3, demonstrating how the symbolic information exposes a loop-based non-contiguous data access.

Both runtime-based and symbolic trace construction are viable options for the effective whole-program performance analysis. However, to gain a global picture of the type of data access patterns in the application, the described symbolic information on data access offsets and loops is required. With recent works [11] it is also feasible that symbolic information can be added at runtime, potentially enabling a combined approach. We discuss the implementation of both runtime and static C.A.T.S. tracing in Section 5.

4 Visualizing Whole-Program Performance

With the varying program insights required to inform different whole-program optimizations, a single performance visualization that aims to show all that information would result in an "information overload". On the other hand, simply providing an overview over individual program metrics may not be enough context to inform concrete optimizations. To avoid this, we aim to provide a *task-oriented* performance visualization that provides *actionable* information with respect to specific whole-program optimization tasks. We do this by introducing three separate visualizations, each designed for a specific subset of the common whole-program optimization tasks identified in Section 2, as shown in Fig. 2. Each of these visualizations is discussed in detail in the following sections.

4.1 Memory Timeline

The memory footprint of HPC applications is an important factor, since a large memory consumption may introduce significant communication overhead or lead to prohibitive resource requirements when large problem sizes are tackled. Resource availability thus imposes an upper bound on the problem size that can be solved, as we will observe in a case study on km-scale weather modeling in Section 6.2. Heap profilers [15, 62, 71, 74] are frequently used when trying to optimize an application's memory footprint to track heap memory usage over time and store snapshots at various intervals. These snapshots are then visualized to identify large allocations or data structures, informing where most of the application's memory footprint stems from. Some heap profilers visualize memory usage with (stacked) area charts [12, 62, 71] to view memory usage over time, while others use hierarchical views such as trees or icicle graphs [3] to identify memory usage distribution in hierarchical data structures. While this helps identify large allocations, answering the question of if and how the footprint can be reduced usually requires careful examination of the source code to identify how and where those allocations are being used — or, more importantly, not being used.

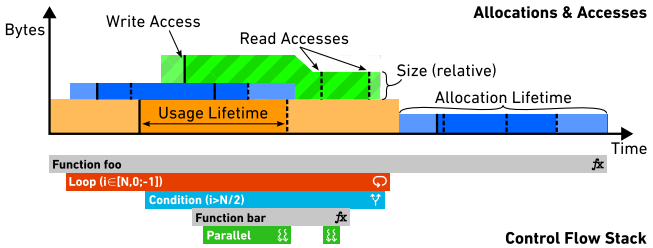


Figure 4: A schematic of the memory timeline view.

Apart from leading to an excessive memory footprint, large allocations are also associated with performance anti-patterns. The most common anti-pattern centered around memory allocation is repeated, costly allocations inside loops or loop nests. A timeline-based traditional heap profiler may capture those allocations and show them as repeated spikes on a footprint timeline, but only if heap snapshots are taken at a fine enough granularity and the program is observed for a certain duration. Both increasing granularity and observed duration lead to a significantly larger trace size than if only the largest snapshots are stored for inspection.

To address these issues, we introduce a novel *Memory Timeline* visualization based on C.A.T.S. traces to visualize allocations in context with the relevant data accesses and control flow information. The Memory Timeline is a 2D chart where the x-axis represents time, and the y-axis represents the amount of allocated memory in bytes. Each tick on the x-axis represents an individual event in the visualized C.A.T.S. trace, uniformly spaced to provide an abstract linear timeline view.

Memory allocations are visualized by drawing a colored region on the chart for each pair of corresponding allocation and de-allocation events, i.e., events sharing the same identifier. An example of this can be seen in the top half of Fig. 4, which shows four regions on the timeline, corresponding to four pairs of events. The x-coordinates at which the region starts and ends are determined by the corresponding allocation and de-allocation events' position in the trace, respectively. The width of the region thus indicates the logical timespan for which the memory is allocated. The height of each region is relative to the number of bytes being allocated. To improve readability, region colors are picked from a rotating color palette based on Kelly's 22 colors of maximum contrast [49], skipping white and black, and conditionally occurring allocations receive a striped pattern. With each new event, regions are stacked vertically on top of each other as they appear, starting at $y = 0$. When a de-allocation occurs and a region ends, any regions stacked on top of it drop down to the top of the next lower region.

Data Accesses. To help inform concrete footprint reduction optimizations, such as memory pooling or reusing buffers, we add information about where which allocations are being used, i.e., accessed, into the visualization. For each data access event recorded in the visualized C.A.T.S. trace, a vertical line is drawn on the corresponding allocation at the time of the event. Write or read accesses are distinguished by drawing solid or dashed lines, respectively. The example in Fig. 4 shows a total of 5 data writes (solid lines) and 8 reads (dashed lines) drawn on top of the visualized allocations.

By highlighting the portion of an allocation between the first and last access to it, we indicate the *usage lifetime* of an allocation. This addition allows us to immediately see if an allocation is stale

and may be reused for a set of other accesses. For program inputs and outputs the highlighted portion is extended to the start or the end of the timeline, respectively. Using the accesses we can further derive the median reuse distance [59] for a given allocation by taking the median of the differences in x-positions between any subsequent accesses to the allocation. This metric, together with the allocation's identifier, size, and the ratio between the usage lifetime and the allocation lifetime of an allocation is provided contextually through tooltips.

Control Flow Stacks. To spot performance anti-patterns associated with allocations requires knowledge about what control flow surrounds individual allocations. We include this information within the same visualization using events from the C.A.T.S. trace, building a second chart that shares the same x-axis below the allocation graph. This graph is constructed as an icicle plot [52] from the control flow stacks associated with each of the events recorded in the timeline. Each entry in the stack is given a separate horizontal, fixed-height bar, from the top down. Its left end is determined by the position of the event where it first appears in a control flow stack, and the right end by the last event where it appears in the stack. An example of this can be seen in the lower half of Fig. 4.

By additionally coloring any bars corresponding to loops (🔁) in red, and bars corresponding to conditional branches (🔀) in blue, the entire graph immediately exposes common allocation anti-patterns. At a glance, it becomes evident when allocation scheduling is suboptimal. In the example in Fig. 4, for instance, the presence of a red bar corresponding to a for loop indicates that two of four shown allocations occur within a loop and may benefit from being rescheduled.

4.2 Data Access Statistics

Data layouts or storage formats that do not align well with the surrounding program schedule are a common performance anti-pattern, causing poor cache utilization or uncoalesced accesses. This is particularly the case with indirect accesses or non-contiguous accesses. To optimize data layouts and improve caching behavior, one has to detect any data accesses that negatively impact performance. However, since changes to data layouts may improve cache utilization in one place and worsen it in another, it is not sufficient to point out locations where data layouts lead to suboptimal data access patterns. Rather, a full overview over all accesses to a given data structure is needed to analyze the potential tradeoff resulting from a layout or storage format change.

We provide a visualization that aggregates this information in a global *Data Access Statistics* view. The statistics view consists of a histogram that provides an overview over the number and types of all data accesses recorded in the C.A.T.S. trace of an application. It then further provides various breakdowns of this statistic by grouping accesses into sub-categories according to the data being accessed, or which control flow construct (e.g., loop, function, etc.) an access occurs in. These sub-categories are displayed as a collapsible tree view, where expanding a tree view node reveals a histogram summarizing the accesses in that specific sub-category and a list of corresponding access events. Sorting tree view items according to the number of a particular type of access helps identify problematic data structures or program locations. This visual ranking and grouping further helps navigate traces for large-scale applications.

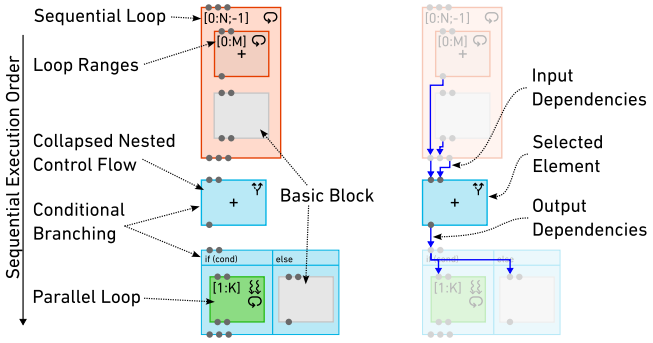


Figure 5: Control flow data dependency view, showing dependencies of a selected conditional block (right).

Access Types. We classify each recorded data access event in a trace into one of four categories based on its symbolic offset expression. These categories form the histogram buckets, each bearing significance with respect to data locality:

(1) **Constant Accesses** are accesses where the symbolic offset expression contains only constants. They may be an indication of data that can be privatized into scalar values.
(2) **Stride-1 Accesses**, also known as linear or contiguous accesses, are accesses where the offset expression depends on the loop variables of one or more surrounding loops. Accesses with such a dependency are considered stride-1, if the inner-most such loop has a stride of ±1 and its loop variable appears only as an additive term in the offset expression. Stride-1 accesses are not associated with a clear negative performance impact.
(3) **Stride-k Accesses**, or non-contiguous/strided accesses, are the inverse of stride-1 accesses. These accesses similarly depend on surrounding loop variables, but the conditions set by stride-1 accesses are not met. They indicate a suboptimal data layout w.r.t. the surrounding control flow.
(4) **Indirect Accesses** are accesses where a different data container is found in the offset expression. The view shows which data containers are being used in the offset expressions. Indirect accesses lead to hard-to-predict access patterns, causing uncoalesced accesses.

An example of this visualization can be seen in Fig. 6, where the visualization helps to quickly expose suboptimal data layouts that can be changed for an easily attainable speedup.

4.3 Control Flow Data Dependency

Long-ranged data reuse or communication overhead in an application is frequently optimized with some amount of program restructuring, for instance through increasing the overlap between communication and computation. Such program reordering optimizations require awareness of data dependencies in an application, as those restrict what can be reordered and how. Data dependencies further dictate where task parallelism can be extracted, where communication or synchronization is required, and they indicate where data is reused with what reuse distance.

To expose this information, we introduce a *Control Flow Data Dependency* visualization, which provides a structural overview of a program together with data dependencies between individual structural components. Using the event sequence in a C.A.T.S. trace in conjunction with the control flow stacks attached to each event, we first construct a hierarchical control flow graph of the application. This graph consists of basic blocks and three key structural components, which we discuss below and show in an example graph in Fig. 5:

- **Loops** are represented with a single loop graph node with the loop body represented in the form of a separate flow graph nested inside the loop node.
- **Conditional branching** is similarly represented with a single conditional graph node, which contains nested flow graphs for each of the conditional branches.
- **Parallel blocks** are graph nodes that represent regions of parallelism, such as parallel loops/kernels or task parallelism. For the latter, each parallel task is represented with a separate graph nested inside the parallel block.

Each of these structural components which contain a nested flow graph can be visually collapsed or expanded to hide or show the contained flow graph through double clicking. When collapsed, a tooltip is shown on the component when hovering that summarizes the contents with respect to points of interest for whole-program analysis. Specifically, the tooltip summarizes:

(1) Depth of the deepest loop nest
(2) Total number of loops
(3) Number of stride-k and indirect data accesses, respectively
(4) Memory footprint and number of allocations

Visualizing Data Dependencies. Using this control flow graph, we can visualize data dependencies between structural components by overlaying data dependency edges on the control flow graph. An edge is drawn between two nodes in the control flow graph for any dependency between the pair of nodes, indicating the direction of the dependency with the arrow direction. Hovering over an edge indicates which accesses are involved in a given dependency. Since a large program contains thousands of dependencies, dependency edges are shown on-demand when a node is activated through clicking, and are removed again if the node is deactivated. An example of how data dependencies are drawn between components can be seen in Fig. 5 (right), and we demonstrate how this view can be used to improve communication-computation overlap and gain noticeable speedups in a distributed setting in Section. 6.1.

5 C.A.T.S. Tracers

Capturing the behavior of highly dynamic applications, such as programs containing sparse computations, typically requires runtime tracing. However, since whole-program optimizations are by their nature rarely isolated to a specific program part, evaluating their impact and informing the next program change usually requires repeatedly running analyses for the entire application between optimizations. With runtime-based tracing and the scale of many HPC applications, this is a time consuming process that consequently slows down the optimization workflow. As discussed in Section 3.3, an alternative to runtime-based tracing is to use symbolic analysis,

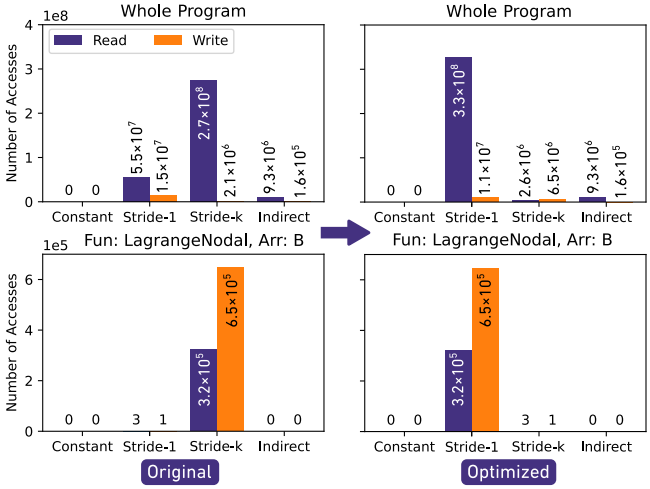


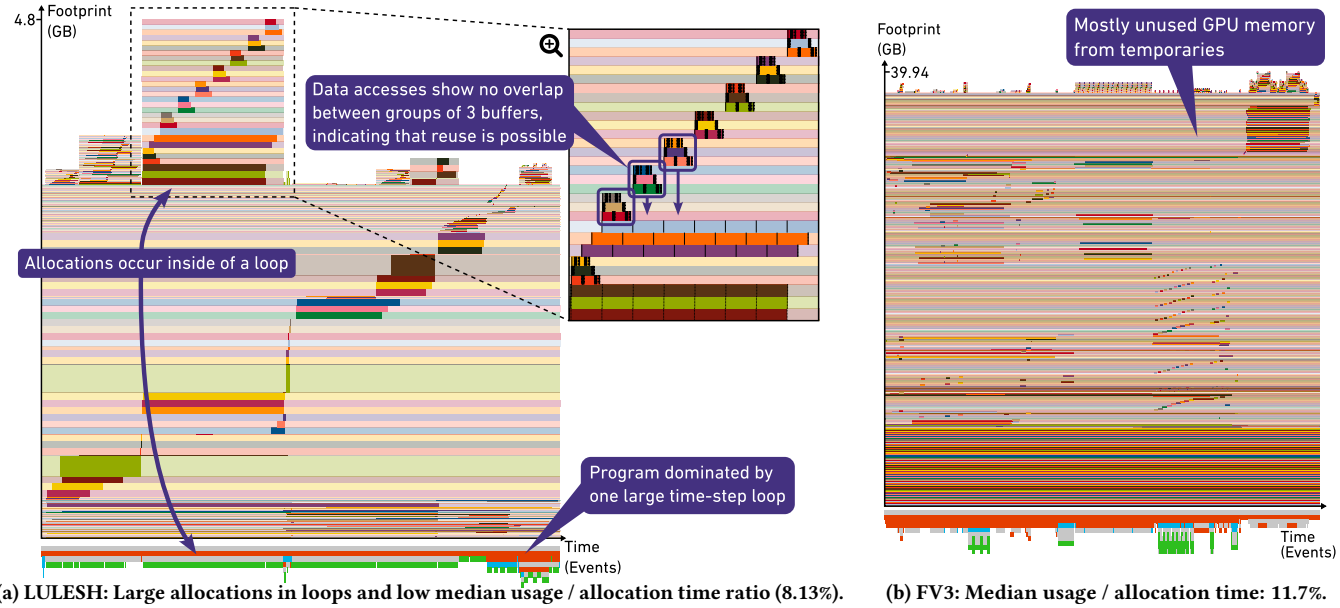
Figure 6: Access statistics visualization showing a histogram of the types of data accesses in LULESH. Data layout changes reduce the number of stride-k accesses program-wide by 30×.

which, in addition to natively exposing symbolic offsets for data access categorization, can be performed without program execution. Given their respective benefits, both means of trace construction are relevant for HPC applications. We demonstrate either method of constructing C.A.T.S. traces with two proof-of-concept tracer implementations, which we both discuss in the following paragraphs and demonstrate on case studies in Section 6.

Runtime Tracer. To construct C.A.T.S. traces at runtime, we implement an instrumentation runtime library and a series of instrumentation passes in LLVM, which insert calls to the runtime library for dispatching trace events. Trace events are dispatched for each allocation, load and store operation, and for each change of control flow scopes, i.e., entering and exiting of loops, branching, functions, and parallel regions (by detecting OpenMP runtime calls). As discussed in Section 3.3, the runtime library is able to filter generated events based on control flow stacks to keep the memory footprint and generated trace size minimal. The library collects traces on a per-thread basis, and when threads are joined, user-defined criteria (e.g., highest footprint, most accesses, or specific thread-id) are used to aggregate the traces of individual threads. Using LLVM, this allows for the construction of traces from any language supported by LLVM, and instrumentation can be performed per translation unit, before linkage.

Static & Symbolic Tracer. The static collection of C.A.T.S. traces is implemented through as a series of compiler analysis passes in the DaCe compiler [6]. The trace collector operates on the compiler's dataflow intermediate representation (IR) to generate trace events with symbolic data access information. We selected DaCe for this since the internal IR lends itself well to symbolic analysis and natively expresses data dependencies, and because the compiler has frontends for Python, GridTools [35], and Fortran, which are the backbone of a wide variety of HPC applications.

Since the compiler IR expresses control flow as a state machine with no specific notion of loops or conditional constructs, a first analysis pass performs graph-analysis based control flow detection. This pass identifies loops, including symbolic expressions for



(a) LULESH: Large allocations in loops and low median usage / allocation time ratio (8.13%). **(b) FV3: Median usage / allocation time: 11.7%.**

Figure 7: The memory timeline view helps reduce the memory footprint of LULESH and FV3 by 1.6× and 3×, respectively.

loop variables, strides, and bounds, and conditional branches including symbolic condition expressions. A second pass constructs the C.A.T.S. event trace by traversing the program's instructions in the order in which the compiler generates instructions during lowering of the IR. For each access event, the offset is recorded with a symbolic offset expression, which is how the offsets are expressed in the IR. A third pass performs data provenance analysis for access events. The pass extracts the data containers used in indirect accesses by tracing data dependency information of the IR backwards until the origin of symbols used in access expressions can be found.

Using these passes, C.A.T.S. traces can be constructed from Python and Fortran on consumer hardware, independently of the resource requirements and execution time of an HPC application.

6 Evaluation

We demonstrate the effectiveness of C.A.T.S. traces and the new performance visualizations using our static trace collector on three case studies in shock hydrodynamics and Earth system modeling. An analysis of the HPCG benchmark [39] demonstrates collection of C.A.T.S. traces at runtime for a highly dynamic application.

6.1 Shock Hydrodynamics

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [48] is a popular proxy application that models representative workloads in HPC systems. We optimize the Python version of the application [55], which uses NumPy [37] for array computations. The program is compiled using a development build of DaCe based on version 1.0.2, with Python 3.12.2 and GCC 8.4.1. The output of the DaCe compiler's automatic optimization pipeline serves as our baseline, and we statically extract a C.A.T.S. trace using our analysis passes. All single-node performance numbers reported are the median over 100 executions on a dual-socket compute node with 2×18 core Intel Xeon Gold 6154 (3GHz) and 384GB RAM, using a problem size of $S = 30$ and 72 OpenMP threads.

Data Access Statistics. To obtain an initial overview, we inspect the data access statistics visualization, shown in the top left of Fig. 6. This view indicates that, over the entirety of the application, there are more stride-k data accesses than there are stride-1 accesses. This possible anti-pattern can be further investigated by sorting the tree-view, grouping by arrays in descending order of stride-k accesses. The top-most entry is an $S^3 \times 3 \times 8$ array B, where S represents the input problem size. Based on the corresponding histogram, shown in the bottom left of Fig. 6, this array has a total of 9.7×10^5 stride-k accesses and only 4 stride-1 accesses to it, indicating that the data layout is likely not ideal. Indeed, looking at the accesses to that array in the program reveals that most of them carry an offset expression of the form $24i+c$, where i is an iterator of a surrounding loop, and c represents different constants. This shows that loops are used to index into the 1st (S^3-sized) dimension of the array, but the 3rd (size 8) dimension is the contiguous one in memory. Based on this, we change the layout to make the 1st dimension contiguous in memory, and as shown in the bottom right of Fig. 6, this improves the data access behavior significantly. We repeat this process for each array in the program that has more stride-k accesses to it than stride-1 accesses, reducing the total number of stride-k accesses by a factor of 30× (top right of Fig. 6). This allows us to obtain a 1.18× single-node speedup over the baseline.

Memory Timeline. Creating a new trace after layout optimizations and visualizing it in the memory timeline view reveals a maximum memory footprint of $4,776 \cdot S^3$ bytes, where S is the problem size. The symbolic nature of the C.A.T.S. trace allows this to be used for a scaling analysis, indicating that the maximum footprint for a problem size of $S = 30$ will be 130.44MB, but rises to 4.78GB for a $S = 100$ problem size. The usage lifetime for data containers, indicated with the saturated portions of the bars drawn for each allocation in Fig. 7a, reveal at a glance that there is likely potential for reducing this memory footprint. Most data containers have a usage lifetime that is short compared to the time they are allocated.

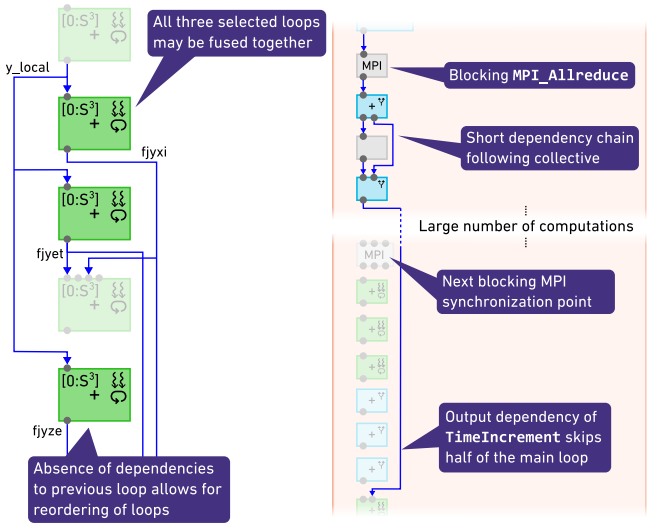
Additionally, many containers of similar size do not overlap in their usage lifetime, meaning that buffers may be reused to save space. For instance, a view at the portion of the application where the footprint is highest, shown zoomed-in to the right of Fig. 7a, exposes a collection of 24 allocations of exactly the same size. The sequence of accesses to these allocations and the resulting usage lifetime indicates that all accesses can in fact be placed in only 3 of the 24 allocations, removing the need for 21 allocations and reducing the peak footprint by 21% to $3,768 \cdot S^3$ bytes.

At the same time, the solid red bar at the bottom of the timeline at the time of these allocations indicates that they are being repeatedly allocated and de-allocated inside of a loop. This corresponds to $528 \cdot S^3$ bytes being allocated and de-allocated in each iteration of the loop, which can be costly and thus represents a possible performance anti-pattern. By moving these remaining allocations and de-allocations to right before and after the loop, together with removing the need for 21 of these allocations, we gain a speedup of 1.44× over the baseline.

By repeating this optimization pattern of identifying buffers which may be fused together, we are able to reduce the maximum footprint of the application by a factor of 1.6× down to $2,928 \cdot S^3$ bytes, given a problem size S. After additionally moving any allocations occurring inside of loops according to the timeline to before the main program loop, this allows us to obtain a total speedup of 2.38× over the DaCe optimized code.

Control Flow Data Dependency. As the timeline view already demonstrates in the control flow stacks at the bottom, the application consists of a large number of parallel loops which occur one after another. The control flow overview, a section of which is shown in Fig. 8a, reveals that most of these parallel loops run over the same loop bounds, i.e., over the range $[0, S^3 - 1]$. This means that any two consecutive parallel loops over the same bounds may be fused together if there are no data dependencies preventing that. To investigate whether there are such loops that can be fused, we inspect any neighborhood of two or more adjacent parallel loops in the visualization, clicking each of the loops and revealing edges indicating data dependencies to other control flow elements. Consequently, any two consecutive loops without any edge between them can be fused together. An example of three selected parallel loops can be seen in Fig. 8a. Two of the loops shown are consecutive with the same loop bounds and no dependencies between them, and can consequently be fused. A third loop, separated by a different loop, has the same loop bounds and also no dependencies to the other loop pair. Since there is no dependency between it and the loop separating it from the other two loops, the code may be reordered to allow all three loops to be fused together.

In the presence of an edge between two loops, a quick look at the corresponding code locations can reveal if the dependency is a simple producer-consumer relationship, e.g., where one loop generates N elements of an array and the subsequent loop reads those N elements. In those cases, the loops can also be fused together, and if the resulting single loop no longer has that array as an output dependency in the control flow view, the array can additionally be privatized to a single scalar value, further reducing the memory footprint. By doing this for all fusible loops we can find in the visualization, we are able to further reduce the memory footprint



(a) Exposed loop fusion options. **(b) Unnecessary blocking collective operation at start of loop.**

Figure 8: Control flow data dependency view of LULESH.

down to $2,400 \cdot S^3$ bytes, which is only *half of the original memory footprint*. In addition to that, fusing loop nests together allows the compiler to perform more vectorization, giving us a **total speedup of 2.6×** over the DaCe optimized code.

Optimizing MPI Communication. When run in a distributed setting, the C++ implementation of LULESH [54] communicates between ranks 4 times in each main time-step loop iteration. The main time-step loop corresponds to the longest red bar at the bottom of Fig. 7a. Communication with grid neighbors occurs three times towards the midpoint of the timeline, while the first communication occurs in the `TimeIncrement` procedure right at the start of each iteration, in the form of a blocking `MPI_Allreduce`. This effectively introduces a barrier at the start of each time-loop iteration. When analyzing the control flow data dependency view, shown in Fig. 8b, the edges of the `TimeIncrement` procedure indicate that the procedure's output is only read significantly later, when the rest of the communication occurs. This implies that a non-blocking `MPI_Iallreduce` could be used instead, with a corresponding wait only where the result is needed according to the dependency view. Not only does this increase the overlap between computation and communication, it also moves all barrier points closer together. We implement this change in the C++ version of LULESH and measure the performance over 10 runs with a problem size of $S = 30$ on 256 HPE Cray EX nodes with 4 NVIDIA GH200 superchips (i.e., 4×72 Arm Neoverse V2 Cores at 3.4GHz with 4×128GB RAM) each. Running with 2197 ranks and 32 OpenMP threads each, we observe a 5% speedup over 1000 time steps, which corresponds to a 10% reduction in the overall communication overhead. The speedup varies with the amount of work imbalance in the grid, with more imbalance leading to a more significant speedup.

Trace Construction. Constructing the traces used for these analyses takes only 24 seconds on a consumer-grade development machine, and generates a trace of 395KB. Running the data provenance analysis to facilitate construction of the access statistics takes another 70 seconds. Performing a memory footprint analysis with

conventional heap profilers, such as Valgrind [62], takes 41 minutes on the same machine and generates a trace at least 30× larger, slowing the program down by two orders of magnitude over execution with no profiler. Using HPCToolkit [2], constructing a trace takes less time with 24 minutes when sampling every 10,000 instructions, but the generated trace is 4× larger than that of Valgrind. An overview of the different tracing times and generated trace sizes for different tracers when applied to LULESH can be seen in Table 1.

6.2 Kilometer-Scale Weather Modeling

The Geophysical Fluid Dynamics Laboratory's (GFDL) Finite-Volume Cubed-Sphere Dynamical Core (FV3) is a numerical weather prediction code that forms the foundation of the Next Generation Global Prediction System project [45]. One crucial challenge with Earth system modeling remains increasing the grid resolution. Since the dynamical core is mostly memory-bound, increasing the number of grid points per GPU increases the performance, both due to individual GPU utilization and effective grid size with the same HPC resources. We analyze and optimize the memory footprint for Pace [61], which is the GFDL's actively developed Python implementation of FV3 using GridTools [35], currently the *fastest performing version of the model*. Even running Pace at just a 51.2km resolution with 79 atmospheric levels, which corresponds to roughly 15 million grid points, requires 60GB of GPU memory. Scaling up to a 25.6km resolution increases the memory footprint to 240GB (39.94GB per rank), requiring the use of at least 3 NVIDIA H100 GPUs with 96GB of HBM3 memory each to run. This leads to the available hardware either limiting the model resolution, or significant underutilization of GPU compute abilities due to the high GPU memory consumption.

By statically analyzing a 25.6km resolution configuration of Pace with our memory timeline visualization, a global view of which is shown in Fig. 7b, it becomes clear at a glance that memory is not being used efficiently. The visualization shows only a small band of allocations that are being used for the entire duration of the application, indicated by the region of saturated colors at the bottom of the timeline. These allocations correspond to the inputs and outputs of the program and only make up 6.56GB, or 16.4% of the maximum memory footprint for a single rank. The remaining 83.6% of the consumed memory is made up of temporary allocations, which are allocated for the entire duration of the program lifetime to avoid the anti-pattern of allocating inside of a loop nest. Each of these temporary allocations is only used for a fraction of their allocated time, and they are all similar in size, most with a size of 93.3MB. The pattern of the saturated versus unsaturated color regions in the visualization allows us to quickly identify that the actual maximum memory requirement, assuming optimal reuse, is thus significantly lower than 39.94GB per rank.

To manually optimize the footprint, we use the visualization to first identify an array A with a low usage lifetime to allocation lifetime ratio $R_A^{U/A}$ by looking for regions with a small amount of saturated colors. We note the size of the array, $|A|$, and visually scan the allocation graph horizontally from the end of the allocation's usage lifetime to the end of the graph. Each time we encounter a non-overlapping usage lifetime of a different array B with the same size $|B| = |A|$, we greedily move all accesses to B to instead access

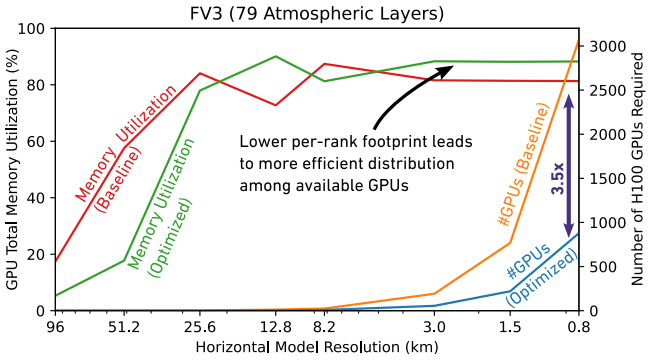


Figure 9: Footprint reduction helps achieve 3.5× reduction in the number of H100 GPUs required to run FV3 at km-scale.

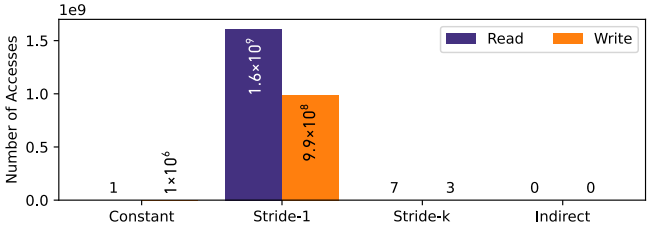
A with a find-and-replace operation on the code. This enables the removal of B, saving on memory footprint while increasing $R_A^{U/A}$ and simultaneously improving memory reuse through coalescing more data accesses. In case any of the accesses to B occur within a sequential loop, which is identified through the presence of a red bar at that point below the x axis, we check the control flow data dependence view to ensure there is no loop-carried dependency preventing the access to be moved. Once the end of the graph is reached, we perform the same visual scan to the left, starting from the beginning of the usage lifetime of A to the start of the graph.

By repeating this process for as long as there appear to be similar sized arrays with non overlapping usage lifetimes, we are able to eliminate a total of 306 allocations through buffer reuse optimizations. This reduces the total memory footprint of the application to **33.6% of the original footprint**, with a new maximum of 13.41GB of memory being required per rank. Consequently, the entire 25.6km resolution program with six ranks now fits onto a single H100 GPU. This allows us to improve resource utilization, while allowing us to now use the same 3 H100 GPUs to double the model resolution to 12.8km with 41 million grid points, which, with our buffer reuse optimizations, now has a per-rank footprint of just over 45GB compared to the original 140GB. Fig. 9 shows how this memory footprint reduction affects scaling and GPU memory utilization as the model resolution moves towards a sub-1km scale.

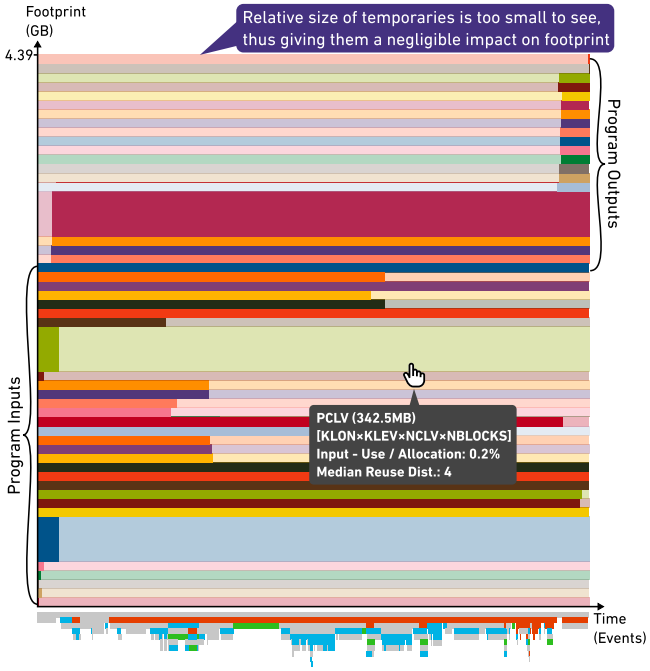
Generating the initial C.A.T.S. trace for this analysis, which is only 1.2MB in size, takes 65 seconds on consumer hardware and does not require any GPUs or other specialized hardware.

6.3 A Case for Local Optimization

We investigate the whole-program performance characteristics of a part of the European Centre for Medium-Range Weather Forecasts' (ECMWF) Integrated Forecasting System (IFS) – specifically, the Fortran implementation of the cloud microphysics scheme (CLOUDSC) [23]. On a compute node with 2×18 core Intel Xeon Gold 6154 clocked at 3GHz and 384GB RAM, running with 72 threads, a roofline analysis indicates that the application is heavily memory bound and only reaches 30% of peak memory bandwidth. This indicates that data locality could be improved. We statically construct a trace for the application and examine it using our data access statistics view. As shown in Fig. 10a, almost all data accesses are performed either with constant index expressions or contiguously. As such, there do not appear to be any anti-patterns that



(a) CLOUDSC's program-wide access statistics show no anti-patterns.



(b) The timeline view shows inefficient memory usage, but long-range reuse distance with a program-wide median of 18 accesses.

Figure 10: Visualizations of CLOUDSC's C.A.T.S. trace.

could be resolved through data layout or storage format changes. Similarly, the memory timeline view (Fig. 10b) shows that there are no clear anti-patterns with respect to allocations, given that the relative size of temporary allocations is so small (max 5.4KB) that they are invisible in the timeline without zooming in significantly. While the timeline shows a generally poor utilization of the allocated memory, it also indicates that all large allocations are part of the program input or output and thus cannot be reused without further domain knowledge. Lastly, the timeline shows that the overall logical long-range data reuse is low with a median reuse distance of only 18 accesses between any two accesses to the same allocation. This indicates that no large-scale code reordering is necessary to improve data reuse, but rather that fine-grained kernel- or loop-level data reuse optimizations such as loop tiling are necessary. By obtaining the program trace statically, this insight is gained early in the optimization workflow and thus quickly reduces the search space for possible optimizations and more costly runtime analyses.

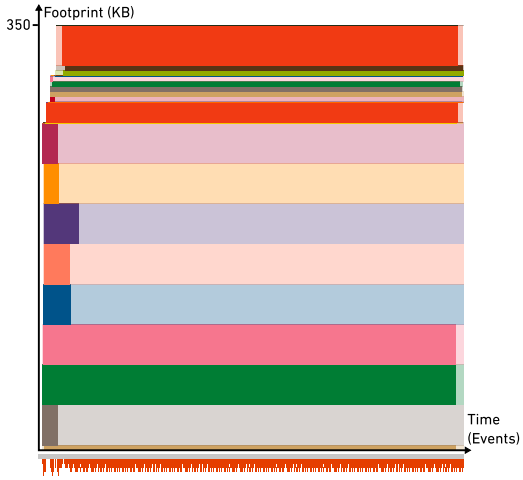


Figure 11: Timeline view based on a runtime C.A.T.S. trace for the sparse computations found in the HPCG benchmark.

6.4 Dynamic Applications

The High Performance Conjugate Gradient (HPCG) benchmark [39] performs computations and access patterns that are typically found in HPC applications. This includes sparse computations, whose dynamic nature poses a challenge for symbolic or static tracing. We thus use our LLVM-based runtime tracer discussed in Section 5 to construct a C.A.T.S. trace for the application. While tracing adds a runtime overhead over running the application without tracing, this allows for construction of a trace in 5.3 seconds on consumer hardware, and the resulting trace is only 2.1MB in size. As seen in the memory timeline view in Fig. 11, this trace can be used to perform an in-depth analysis of the memory footprint over time. Performing the same analysis with Valgrind, for instance, generates a more than 2× larger trace. A detailed comparison between different tracing approaches and tools with respect to tracing time and generated trace sizes can be seen in Table 1.

The visualization in Fig. 11 reveals a repeating pattern in the loops shown in the control flow stacks timeline at the bottom of the figure, when function scopes are not drawn. If functions are shown, the control flow timeline reveals a triangular shape that grows to the bottom (◥). This shape, combined with the gray color indicating function scopes, indicates recursion, where the height of the triangle represents recursion depth. Since each recursive call changes the control flow stack, it leads to a new set of unique events that are kept, meaning that trace size for highly recursive programs may grow with problem size. This can be avoided by not just filtering events for unique control flow stacks during tracing, but also for repeating patterns found in recursive function calls.

7 Related Work

Extensive research has been conducted in the field of performance visualizations to help understand application performance [42]. HPC performance analysis frameworks such as HPCToolkit [2, 57, 89], Paraver [69], TAU [78], Scalasca [28, 51], or VTune [41] all have specialized visualization components to understand an application's performance characteristics. As such, they typically offer a global view of an application with respect to specific metrics,

	C.A.T.S.				
	Static	**Runtime**	Valgrind	HPCToolkit	
Time	24s	39s	41min	24min	**LULESH**
Size	395KB	191KB	11.8MB	47.2MB	
Time	N/A	5.3s	2s	26s	**HPCG**
Size	N/A	2.1MB	5.2MB	20.49MB	

Table 1: Comparison of tracing time and generated trace size between established approaches on LULESH and HPCG.

but are typically geared towards locating hot-spots and bottlenecks, rather than performing more coarse, whole-program optimizations.

Control Flow and Data Dependence. Decisions around larger code restructuring to improve communication or long-range data require insights into the structure of an applicaton. Visualizations such as Vampir [60], Jumpshot [86, 87], or Boxfish [43, 53] provide this by visualizing inter-process or inter-node communication. An alternative technique is the use of Calling Context Trees, such as in Trevis [1], VIPACT [65], or Callflow [50, 64]. EXTRAVIS [16] employs a circular bundle view to group logical program constructs and dynamically show relationships between them. In this work, we have identified that for many performance anti-patterns, the structural granularity offered by call stacks is not sufficient to reliably detect them. As such, we use full control flow stacks including loops, function calls, and branching to provide more context. Only very few performance tools use a comparable granularity [2, 19], and they typically have to recover this information from analyzing program binaries or instruction sequences.

Memory Timeline. Heap profilers [74] such as Valgrind [62] or GCspy [15, 71] were created to help reduce application memory footprint. Over the years, the research community has developed numerous intuitive visualization schemes to expose inefficient data allocations [8]. For hierarchical data structures, Memory Cities [85] proposed using a city metaphor to identify problematic heap object groups or memory leaks. Heapviz [3] takes a different approach to hierarchical data, breaking allocations down in a radial tree diagram. Byma and Larus [12] provide contextual information by visualizing allocations using a flame graph in the context of callstacks. However, despite these visualizations, deciding where buffers can be reused or memory may be pooled still requires detailed inspection of the corresponding code [26]. By visualizing data accesses and control flow information alongside allocations, we provide the necessary context to inform reuse and pooling decisions.

Data Access Overview. Due to the importance of data locality in HPC applications, numerous frameworks and visualizations have been proposed to analyze data reuse and caching behavior, especially w.r.t. application scaling [58]. Various techniques have been employed, ranging from detailed cache and data access simulations [34, 36, 40, 75], to trace-based visualizations, such as MACPO [73] or MemAxes [31]. Ayers et al. [4] have demonstrated how classifying memory accesses by types gives enough insight into locality behavior to inform effective prefetching decisions. By classifying data accesses by types and grouping them for program regions or specific allocations, we provide an interactive and quick baseline for deciding where data layout changes help data locality, and how program scaling affects this decision.

Static and Symbolic Traces. While constructing program traces statically to avoid input-dependence and runtime costs is a frequently explored technique [21, 22, 81], the idea of adding symbolic program information to program traces has not been explored much. Tensorize [11] recently demonstrated the importance of symbolic program information, and the challenges associated with capturing this information from source code representations. Works that have explored this in the past, such as Code Vectors [38], which uses symbolic traces to learn code word embeddings, typically rely on symbolic program execution to construct traces. Symbolic program execution is a popular technique in program correctness checking, with tools such as Klee [13] or SymCC [70]. However, HPC codes pose a challenge for symbolic execution, which suffers from state space explosion in deep loop nests or difficulties with operating on floating point values [5]. We solve these challenges by building symbolic traces from the symbolic intermediate representation of a compiler designed for HPC application optimization.

8 Conclusion

In this work, we identify the key components necessary to inform effective whole-program optimization decisions, and introduce the C.A.T.S. tracing format to capture this information. In particular, we illustrate the importance of fine-grained contextual control flow information in performance results to detect and resolve common performance anti-patterns. Based on the trace format, we introduce three novel visualizations to provide a whole-program view of application characteristics that impact the global program performance. We highlight the benefits of both runtime-based and static trace construction, and implement compiler analysis passes to construct symbolic and runtime C.A.T.S. traces on consumer hardware. Using control flow stacks, the size of generated traces is limited to a few megabytes even for large, long-running applications.

In case studies with applications written in Python, Fortran, and C++, we demonstrate how our visualizations help detect where whole-program optimization is necessary and how to inform subsequent optimization decisions. Using our memory timeline, we are able to reduce the memory footprint of the FV3 dynamical core by a factor of 3×, drastically reducing the hardware resource requirements for sub-km scale Earth system modeling. In a case study on optimizing LULESH, the global data access statistics view helps us improve data layouts, leading to a total speedup of 2.6×. With the help of our control flow data dependency view, we are additionally able to improve the communication and computation overlap leading to speedups in a distributed environment. By providing visualizations to statically understand program performance characteristics, we extend the toolbox of performance engineers to enable optimization workflows in the absence of clear local bottlenecks.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047), and from the Swiss State Secretariat for Education, Research and Innovation (SERI) under the SwissTwins grant agreement. Work by Lawrence Livermore National Laboratory was performed under the auspices of the U.S. Department of Energy under contract DE-AC52-07NA27344 (LLNL-CONF-2003780). T.B.N. was supported by LLNL LDRD 23-ERD-022.

References

[1] Andrea Adamoli and Matthias Hauswirth. 2010. 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*. ACM, New York, NY, USA, 73–82. doi:10.1145/1879211.1879224

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (4 2010), 685–701. doi:10.1002/cpe.1553

[3] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proceedings of the 5th international symposium on Software visualization*. ACM, New York, NY, USA, 53–62. doi:10.1145/1879211.1879222

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 513–526. doi:10.1145/3373376.3378498

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2019. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (5 2019), 1–39. doi:10.1145/3182657

[6] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, New York, NY, USA, 1–14. doi:10.1145/3295500.3356173

[7] Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. 2011. Exploiting Data Similarity to Reduce Memory Footprints. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 152–163. doi:10.1109/IPDPS.2011.24

[8] Alison Fernandez Blanco, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. 2023. Software Visualizations to Analyze Memory Consumption: A Literature Review. *Comput. Surveys* 55, 1 (1 2023), 1–34. doi:10.1145/3485134

[9] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Vol. 0. IEEE, 550–560. doi:10.1109/SC.2016.46

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[11] Alexander Brauckmann, Luc Jaulmes, José W. de Souza Magalhães, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2025. Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, New York, NY, USA, 15–30. doi:10.1145/3696443.3708956

[12] Stuart Byma and James R. Larus. 2018. Detailed heap profiling. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, Vol. 13. ACM, New York, NY, USA, 1–13. doi:10.1145/3210563.3210564

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 209. doi:10.5555/1855741.1855756

[14] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. 1999. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*. ACM, New York, NY, USA, 444–453. doi:10.1145/305138.305231

[15] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, and J. Nystrom-Persson. 2006. Visualising dynamic memory allocators. In *Proceedings of the 5th international symposium on Memory management*, Vol. 2006. ACM, New York, NY, USA, 115–125. doi:10.1145/1133956.1133972

[16] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. van Wijk, and A. van Deursen. 2007. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*. IEEE, 49–58. doi:10.1109/ICPC.2007.39

[17] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM Trans. Math. Software* 41, 4 (10 2015), 1–20. doi:10.1145/2699470

[18] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. 2014. PTG: An Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE, 21–30. doi:10.1109/WOLFHPC.2014.8

[19] Wim De Pauw and Steve Heisig. 2010. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*. ACM, New York, NY, USA, 143–152. doi:10.1145/1879211.1879233

[20] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM, New York, NY, USA, 245–257. doi:10.1145/781131.781159

[21] T. Eisenbarth, R. Koschke, and G. Vogel. 2002. Static trace extraction. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE Comput. Soc, 128–137. doi:10.1109/WCRE.2002.1173071

[22] Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel. 2005. Static object trace extraction for programs with pointers. *Journal of Systems and Software* 77, 3 (9 2005), 263–284. doi:10.1016/j.jss.2004.04.028

[23] European Centre for Medium-Range Weather Forecasts. 2003. *CLOUDSC cloud microphysics scheme.* https://github.com/ecmwf-ifs/dwarf-p-cloudsc

[24] Thomas Fahringer and Bernhard Scholz. 2003. Symbolic Analysis of Programs. In *Advanced Symbolic Analysis for Compilers*. Springer, Berlin, Heidelberg, 13–40. doi:10.1007/3-540-36614-8_2

[25] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (2 1991), 23–53. doi:10.1007/BF01407931

[26] Alison Fernandez Blanco, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. 2018. Effective Visualization of Object Allocation Sites. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 43–53. doi:10.1109/VISSOFT.2018.00013

[27] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-matrix Multiplication. *Comput. Surveys* 55, 12 (12 2023), 1–36. doi:10.1145/3571157

[28] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (4 2010), 702–719. doi:10.1002/cpe.1556

[29] Evangelos Georganas, Jorge Gonzalez-Dominguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11. doi:10.1109/SC.2012.32

[30] Lukas Gianinazzi, Alexandros Nikolaos Ziogas, Langwen Huang, Piotr Luczynski, Saleh Ashkboosh, Florian Scheidl, Armon Carigiet, Chio Ge, Nabil Abubaker, Maciej Besta, Tal Ben-Nun, and Torsten Hoefler. 2024. Arrow Matrix Decomposition: A Novel Approach for Communication-Efficient Sparse Matrix Multiplication. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 404–416. doi:10.1145/3627535.3638496

[31] Alfredo Gimenez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. 2018. MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors. *IEEE Transactions on Visualization and Computer Graphics* 24, 7 (7 2018), 2180–2193. doi:10.1109/TVCG.2017.2718532

[32] Antônio Tadeu A. Gomes, Enzo Molion, Roberto P. Souto, and Jean-François Méhaut. 2021. Memory allocation anomalies in high-performance computing applications: A study with numerical simulations. *Concurrency and Computation: Practice and Experience* 33, 18 (9 2021). doi:10.1002/cpe.6094

[33] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: a Call Graph Execution Profiler. *ACM SIGPLAN Notices* 17, 6 (6 1982), 120–126. doi:10.1145/872726.806987

[34] Thomas Grass, Cesar Allande, Adria Armejach, Alejandro Rico, Eduard Ayguade, Jesus Labarta, Mateo Valero, Marc Casas, and Miquel Moreto. 2016. MUSA: A Multi-level Simulation Approach for Next-Generation HPC Machines. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 526–537. doi:10.1109/SC.2016.44

[35] GridTools. 2025. *GT4Py: GridTools for Python.* https://github.com/GridTools/gt4py

[36] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2017. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. In *Tools for High Performance Computing 2016*, Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, 1–22. doi:10.1007/978-3-319-56702-0_1

[37] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (9 2020), 357–362. doi:10.1038/s41586-020-2649-2

[38] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

Vol. 18. ACM, New York, NY, USA, 163–174. doi:10.1145/3236024.3236085
[39] Michael Heroux and Jack Dongarra. 2013. *Toward a New Metric for Ranking High Performance Computing Systems.* Technical Report. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (United States). doi:10.2172/1089988
[40] Roman Iakymchuk and Paolo Bientinesi. 2012. Modeling performance through memory-stalls. *ACM SIGMETRICS Performance Evaluation Review* 40, 2 (10 2012), 86–91. doi:10.1145/2381056.2381076
[41] Intel. 2025. *Intel VTune Profiler.* https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html
[42] Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-timo Bremer. 2014. State of the Art of Performance Visualization. *EuroVis 2014* (2014), 141–160.
[43] Katherine E. Isaacs, Aaditya G. Landge, Todd Gamblin, Peer-Timo Bremer, Valerio Pascucci, and Bernd Hamann. 2012. Abstract: Exploring Performance Data with Boxfish. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 1380–1381. doi:10.1109/SC.Companion.2012.202
[44] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. *Proceedings of Machine Learning and Systems* (2021).
[45] Ming Ji and Frederick Toepfer. 2016. *Dynamical Core Evaluation Test Report for NOAA's Next Generation Global Prediction System (NGGPS).* Technical Report. doi:10.25923/ztzy-qn82
[46] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 194–206. doi:10.1109/MSR59073.2023.00037
[47] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. 1998. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Comput. Soc, 285–296. doi:10.1109/MICRO.1998.742790
[48] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.
[49] Kenneth L Kelly and Deane Brewster Judd. 1976. *Color: universal language and dictionary of names*. Vol. 13. US Department of Commerce, National Bureau of Standards.
[50] Suraj P. Kesavan, Harsh Bhatia, Abhinav Bhatele, Stephanie Brink, Olga Pearce, Todd Gamblin, Peer-Timo Bremer, and Kwan-Liu Ma. 2023. Scalable Comparative Visualization of Ensembles of Call Graphs. *IEEE Transactions on Visualization and Computer Graphics* 29, 3 (3 2023), 1691–1704. doi:10.1109/TVCG.2021.3129414
[51] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91. doi:10.1007/978-3-642-31476-6_7
[52] J. B. Kruskal and J. M. Landwehr. 1983. Icicle Plots: Better Displays for Hierarchical Clustering. *The American Statistician* 37, 2 (5 1983), 162–168. doi:10.1080/00031305.1983.10482733
[53] A. G. Landge, J. A. Levine, A. Bhatele, K. E. Isaacs, T. Gamblin, M. Schulz, S. H. Langer, Peer-Timo Bremer, and V. Pascucci. 2012. Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (12 2012), 2467–2476. doi:10.1109/TVCG.2012.286
[54] Lawrence Livermore National Laboratory. 2025. *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH).* https://github.com/LLNL/LULESH
[55] Lawrence Livermore National Laboratory. 2025. *Python Port of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH).* https://github.com/LLNL/pylulesh
[56] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-Avoiding Parallel Strassen: Implementation and performance. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11. doi:10.1109/SC.2012.33
[57] Xu Liu and John Mellor-Crummey. 2013. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, New York, NY, USA, 1–12. doi:10.1145/2503210.2503297
[58] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Vol. 15-20-Nove. ACM, New York, NY, USA, 1–12. doi:10.1145/2807591.2807648
[59] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117. doi:10.1147/sj.92.0078
[60] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer* (1996).
[61] National Oceanic and Atmospheric Administration - Geophysical Fluid Dynamics Laboratory (NOAA-GFDL). 2025. *Pace.* https://github.com/NOAA-GFDL/pace
[62] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavy-weight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007), 89–100. doi:10.1145/1250734.1250746
[63] Neungsoo Park, Bo Hong, and V.K. Prasanna. 2003. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* 14, 7 (7 2003), 640–654. doi:10.1109/TPDS.2003.1214317
[64] Huu Tan Nguyen, Abhinav Bhatele, Nikhil Jain, Suraj P. Kesavan, Harsh Bhatia, Todd Gamblin, Kwan-Liu Ma, and Peer-Timo Bremer. 2021. Visualizing Hierarchical Performance Profiles of Parallel Codes Using CallFlow. *IEEE Transactions on Visualization and Computer Graphics* 27, 4 (4 2021), 2455–2468. doi:10.1109/TVCG.2019.2953746
[65] Huu Tan Nguyen, Lai Wei, Abhinav Bhatele, Todd Gamblin, David Boehme, Martin Schulz, Kwan-Liu Ma, and Peer-Timo Bremer. 2016. VIPACT: A Visualization Interface for Analyzing Calling Context Trees. In *2016 Third Workshop on Visual Performance Analysis (VPA)*. IEEE, 25–28. doi:10.1109/VPA.2016.009
[66] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 90–106. doi:10.1145/3503221.3508431
[67] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 37–48. doi:10.1109/HPCA.2014.6835955
[68] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. 2015. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. Lecture Notes in Computer Science, Vol. 9137. Springer International Publishing, Cham, 48–57. doi:10.1007/978-3-319-20119-1_4
[69] V Pillet, J Labarta, T Cortes, and S Girona. 1995. Paraver: A tool to visualize and analyze parallel code. *Proceedings of WoTUG-18: transputer and occam developments* February (1995), 17–31.
[70] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SYMCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau
[71] Tony Printezis and Richard Jones. 2012. GCSPY: An adaptable heap visualisation framework. *ACM SIGPLAN Notices* 37, 11 (2012), 343–358. doi:10.1145/583854.582451
[72] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*. ACM Press, New York, New York, USA, 4–13. doi:10.1145/125826.125848
[73] Ashay Rane and James Browne. 2014. Enhancing Performance Optimization of Multicore/Multichip Nodes with Data Structure Metrics. *ACM Transactions on Parallel Computing* 1, 1 (10 2014), 1–20. doi:10.1145/2588788
[74] Colin Runciman and Niklas Rsjemo. 1996. Heap Profiling for Space Efficiency. In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, 159–183.
[75] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2022. Boosting Performance Optimization with Interactive Data Movement Visualization. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Vol. 2022-November. IEEE, 1–16. doi:10.1109/SC41404.2022.00069
[76] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, New York, NY, USA, 53–64. doi:10.1145/1854273.1854286
[77] Kenshu Seto, Hamid Nejatollahi, Jiyoung An, Sujin Kang, and Nikil Dutt. 2019. Small Memory Footprint Neural Network Accelerators. In *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 253–258. doi:10.1109/ISQED.2019.8697641
[78] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications* 20, 2 (5 2006), 287–311. doi:10.1177/1094342006064482
[79] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2018. Computing with Near Data. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 3 (12 2018), 1–30. doi:10.1145/3287321
[80] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S.

Nikolopoulos. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (4 2018), 1422–1434. doi:10.1007/s11227-018-2238-4

[81] P. Tonella and A. Potrich. 2003. Reverse engineering of the interaction diagrams from C++ code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE Comput. Soc, 159–168. doi:10.1109/ICSM.2003.1235418

[82] Didem Unat, Anshu Dubey, Torsten Hoefler, John Berkeley Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H.J. Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericas. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. doi:10.1109/TPDS.2017.2703149

[83] Ronald Veldema, Ceriel J. H. Jacobs, Rutger F. H. Hofman, and Henri E. Bal. 2005. Object combining: a new aggressive optimization for object intensive programs. *Concurrency and Computation: Practice and Experience* 17, 5-6 (4 2005), 439–464. doi:10.1002/cpe.836

[84] Chao Wang, Chengjie Cao, Liyu Ye, Chunhui Wang, and ChunYu Guo. 2023. An efficient peridynamic method and its MPI parallelization for simulating the continuous icebreaking process. *Ocean Engineering* 279 (7 2023), 114460. doi:10.1016/j.oceaneng.2023.114460

[85] Markus Weninger, Lukas Makor, and Hanspeter Mossenbock. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *2020 Working Conference on Software Visualization (VISSOFT).* IEEE, 110–121. doi:10.1109/VISSOFT51673.2020.00017

[86] C.E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. 2000. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *ACM/IEEE SC 2000 Conference (SC'00).* IEEE, 50–50. doi:10.1109/SC.2000.10050

[87] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. 1999. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications* 13, 3 (8 1999), 277–288. doi:10.1177/109434209901300310

[88] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 1089–1102. doi:10.1109/ISCA45697.2020.00092

[89] Keren Zhou, Mark W. Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 34th ACM International Conference on Supercomputing.* ACM, New York, NY, USA, 1–12. doi:10.1145/3392717.3392752

[90] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Transactions on Architecture and Code Optimization* 14, 1 (3 2017), 1–26. doi:10.1145/3023362