ABSTRACT

Title of Dissertation: DATA-CENTRIC PERFORMANCE MEASUREMENT AND MAPPING FOR HIGHLY PARALLEL PROGRAMMING MODELS

Hui Zhang, Doctor of Philosophy, 2018

Dissertation directed by: Professor, Jeffrey K. Hollingsworth,
Department of Computer Science

Modern supercomputers have complex features: many hardware threads, deep memory hierarchies, and many co-processors/accelerators. Productively and effectively designing programs to utilize those hardware features is crucial in gaining the best performance. There are several highly parallel programming models in active development that allow programmers to write efficient code on those architectures. Performance profiling is a very important technique in the development to achieve the best performance.

In this dissertation, I proposed a new performance measurement and mapping technique that can associate performance data with program variables instead of code blocks. To validate the applicability of my data-centric profiling idea, I designed and implemented a profiler for PGAS and CUDA. For PGAS, I developed ChplBlamer, for both single-node and multi-node Chapel programs. My tool also provides new features such as data-centric inter-node load imbalance identification. For CUDA, I developed CUDABlamer for GPU-accelerated applications. CUDABlamer also attributes performance data to program variables, which is a feature that was not found in any previous CUDA profilers. Directed by the insights from the tools, I optimized several widely-studied benchmarks and significantly improved program performance by a factor of up to 4x for Chapel and 47x for CUDA kernels.

DATA-CENTRIC PERFORMANCE MEASUREMENT AND MAPPING FOR HIGHLY PARALLEL PROGRAMMING MODELS

by

Hui Zhang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Dedication

To my wife,

For continuously loving me and driving me forward.

To my mother,

For support and never asking why it is taking so long to finish.

To my grandmother,

How I wish you could see this, but I know you are always here for me.

Acknowledgments

I would like to begin by thanking my advisor, Dr. Jeffrey K. Hollingsworth. His guidance, inspiration, and support are invaluable to my research, my career, and my life. His serious and professional attitude and the commitment to the work set a great example for me. Without him, this dissertation would not have been possible.

I want to thank my advisory committee: Professor Donald Yeung, Professor Bruce Jacob, Professor Alan Sussman, and Professor William Dorland. Each member of the committee contributed valuable suggestions that helped me finish this work.

I would also like to thank my colleagues in the research group: Ray Chen, Richard Johnson, Mike Lam, Sukhyun Song, and Yoav Segev. Thank you Ray, you are like a second advisor to me, enlightening me when I was lost in the research. Thank you Richard, for sharing ideas of research, thoughts about life, and points on geopolitics; and of course, for the hop-on day trip and Canadian dollar bills. Thank you Mike, for encouraging words through email when I struggled for the prelim examination.

I want to thank all other friends I made since I came to the States in 2011, such as Nuttiiya Seekhao, Peixin Gao, Xingjian Ling, Teng Long, Yuan Zhou, Yuhan Rao, and Lei Wang, for the time we spent together and the support when it was needed.

To my wife, Lianglei Zhang, I cannot thank you enough for all your understanding and support, although we were apart in different countries for most of the time. I promise I will put you in front of work at any time.

Finally, I want to express my respect and gratitude to my mom, Wenjuan Jin. Your endless love is truly what keeps me going and you are a lifetime role model to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the computing power of distributed systems escalates, the complexity of scientific and engineering problems that can be solved by these systems also increases. However, there is a divide between system designers who know how to utilize these distributed systems efficiently and the people who have real problems to solve. There has been much effort made from different directions to help take full advantage of the power of parallel architectures. One effort to better utilize large-scale, highly parallel and increasingly heterogeneous supercomputers is to develop parallel programming models that have better productivity and higher performance. This dissertation studies the performance of two highly parallel programming models: PGAS (Partitioned Global Address Space) and CUDA (Compute Unified Device Architecture), which are popular on recent supercomputers. Specifically, PGAS is a promising productive programming model for systems with tens of thousands of CPUs; CUDA is currently the mainstream programming model for CPU-GPU heterogeneous computing systems. Newer and higher-level programming models usually ease the programming for end-users, but how to associate the performance issues back to original program elements is also a critical issue in understanding the program performance characteristics.

Performance tools can help users take full advantage of the power of parallel architectures by providing insights into where and why a program fails to obtain peak performance. Currently, there are few profilers for highly parallel programming models and most of them are code-centric, meaning they can only associate performance data to computation regions in the source, such as functions, loop, and code blocks. The conventional code-centric view of performance data is helpful in pinpointing hot spots at the granularity of from the instruction-level to the procedure-level in the program. However, the traditional code-centric view of performance data lacks the capability to find performance problems associated with different variables accessed by specific lines of the code. In many cases, it is the data and its movement that cause the greatest performance loss instead of the computation. Additionally, data-centric profiling can aggregate performance statistics from all memory accesses that are associated with the same variable via full data flow analysis, which a code-centric profiler cannot accomplish. Data-centric approaches are especially important for HPC applications since memory allocation and data movement are usually the bottlenecks of the overall performance. Therefore, a profiling tool that can identify these inefficiencies and associate them with memory regions and source-level data abstractions is highly desirable. My thesis shows that a new data-centric performance mapping technique can greatly help PGAS and CUDA programmers to improve the performance of their programs.

Partitioned Global Address Space (PGAS) [23] is an alternative programming model that marries the good performance scalability of message passing with the good programmability of a shared memory model. PGAS improves the productivity in HPC by introducing an additional layer of abstraction, described as "global address space" for a cluster. It defines a global memory address space that is logically partitioned and a portion of it is local to each process, thread, or processing element

[40]. The novelty of PGAS is that portions of the shared memory space may have an affinity for a particular process, thereby allowing programmers to exploit locality of reference. Because of this additional layer of abstraction, users do not have to explicitly handle the communication between nodes as is required in traditional message-passing based programs, such as MPI. However, a major challenge of higher level semantics is that it also significantly increases the difficulty of diagnosing performance bottlenecks that are now hidden from users.

Among the languages that have PGAS features, I chose Chapel [1] for evaluation mainly for two reasons:

1. Chapel is a promising language in productively solving large HPC problems with hybrid parallelism, but currently, it still has much room for performance improvement, especially in distributed systems with heavy communication between multiple nodes.
2. Currently, there are only a few performance analysis tools that are Chapel-specific and user-friendly with valuable performance insights for user-level performance optimization.

Even though this work is built for Chapel, the methods and ideas I used can be applied to other programming languages. This is because: first, the tool framework and top-level abstractions in implementing this data-centric idea are generic to most programming languages; second, my analysis is based on a machine-independent intermediate representation LLVM. Therefore, both my idea and my tool can be extended to support a number of other programming languages.

Besides multi-thread, multi-node computing, the use of accelerators, such as GPUs, is growing rapidly in supercomputers and thus CUDA programming also plays an important role in achieving high performance. The application of GPUs in general-purpose-computing other than traditional graphics processing is known as GPGPU. The main challenge in developing such programs is that they often do not fit in the model required by GPUs, limiting the scope of applications that may benefit from the massive parallelism provided by GPUs. Even if the application fits the GPU model, obtaining optimal performance using heterogeneous architectures is non-trivial. Therefore, it is important to create performance tools that assist the development and guide programmers to write efficient code.

Although a handful GPU profilers exist [57, 58, 59, 75], most traditional tools, unfortunately, simply provide programmers with a number of different kinds of measurements and metrics obtained by running applications. It is very hard for users to map these metrics back to source code to understand the root causes of slowdowns, much less decide what next optimization step to take to alleviate the bottlenecks and improve the overall performance. Therefore, I applied the same data-centric idea to GPU-accelerated applications, implementing a tool prototype for programmers to easily understand the causes of performance degradation and hotspots that consume most GPU time during execution.

In this dissertation, I describe new performance mapping techniques that employ sampling in conjunction with static analysis to study the performance of highly parallel programs running on modern architectures with many CPU and GPU cores.

In the rest of this chapter, I briefly describe the contributions of this work, the dissertation organization, and the thesis statement.

Contributions This dissertation describes the design and implementation of two performance measurement and mapping tools for PGAS and CUDA. The fundamental uniqueness of my tools is that they provide a data-centric profiling feature that can link the runtime performance data back to program variables and data structures with a complete user-level calling context. The way I determine the mapped variables is not to simply link the accessed memory addresses with the specific variables that are created with the allocation within that memory range but to figure out the program-wide variables that should be really "responsible" for that executed statement. This inclusive data-centric performance analysis technique can also propagate the performance loss due to the use of third-party libraries up to user-visible variables so that programmers are able to focus on optimizing user code instead of trying to improve the performance of the library.

Specifically, building on Blame by Rutar [18], I implemented two performance tools. First, I designed and developed ChplBlamer, for both single-node and multi-node Chapel environments. It supports most Chapel language features and provides hierarchical profiling over program abstractions and call path profiling in a user-level context. It pinpoints performance losses due to data distribution and remote data accesses, and provides new features such as data-centric inter-node load imbalance identification. It solves problems like multi-threading and asynchronous tasking that Blame [18] does not solve. The combination of the tool's data-centric and code-centric profiling provides insights into inter and intra node communication bottlenecks that could not be discovered by previous Chapel profilers. Secondly, I developed an effective performance tool for GPU-accelerated HPC applications. It is also able to map performance statistics to program variables and it provides complete calling context from the device stack to the host stack for the sampling-based runtime data. The tool is able to provide different performance insights into kernels against existing CUDA profilers, such as Nvidia Visual Profiler (NVP) [57] and TAU [3].

I conducted several benchmark experiments for each tool to validate the functionality and utility of the tool, and manually optimized those benchmarks with the guidance of my tools. By doing this tuning, I achieved significant speedups of up to 4x for Chapel and 47x for CUDA kernels.

Dissertation organization There are total 7 chapters in this dissertation. Among those, the contents of Chapter 4 and Chapter 5 are adapted from my previous conference publications [68, 69]. Chapter 2 describes the background in PGAS programming model, Chapel language features, GPGPU, and several widely-used performance analysis techniques that show us the big picture of the field that I've been working in. Chapter 3 introduces the unique performance metric "Variable Blame" used in my tools and illustrates its calculation with a few simple examples. Chapter 4 describes the design and implementation of a single-node Chapel profiler for end users. Chapter 5 shows the implementation of a multi-node Chapel profiler that supports detecting communication bottlenecks. Chapter 6 describes the design and implementation of a data-centric profiler for guiding CUDA optimization in GPU-accelerated applications. Finally, Chapter 7 presents some conclusions from my work and identifies a few directions for future research.

Thesis statement Using static analysis, plus hardware counter based sampling in conjunction with call path profiling, one can develop data-centric profilers with reasonable overhead to analyze program executions on a parallel architecture with many hardware threads, deep memory hierarchies, and GPU accelerators. These methods can provide rich information to guide code optimization.

Chapter 2

Background

Performance is always essential to HPC. The efforts to make programs run faster lie in two broad categories: hardware and software. On the one hand, hardware innovations have been impressive, especially in the recent two decades as the supercomputer systems are becoming more and more heterogeneous, and as a result, more complicated. On the other hand, software innovations are often not in the spotlight but the problems are actually becoming more and more critical. With each new hardware feature, the challenge of how to utilize new hardware architectures to make programs run faster and more efficient is getting harder, not easier. Researches from different fields have made many attempts to improve system software, runtime libraries, compiler-based programming model, and domain-specific application libraries. Programming model can abstract away the complexity of the machine and abstract away the implementation differences, thus letting programmers focus on the algorithmic choices. Evaluating the performance of a new programming model is significantly important in its development. This thesis focuses on providing performance insights for two highly parallel programming models: PGAS and CUDA. The rest of this chapter introduces prior work about these two programming models and their performance tools.

2.1 PGAS Language

To exploit locality and scalability in High Performance Computing (HPC), while alleviating the burden on programmers and improving the productivity in parallel programming, researchers continue to look for new programming models other than the traditional C plus MPI/OpenMP model. Partitioned Global Address Space (PGAS) [23] emerged in the last twenty years as an alternative, and it has been actively studied in both academia and industry. PGAS provides users with a flat address space atop a possible physically distributed computer memory system, so users can write programs as if coding for a shared memory system and let the underlying infrastructure take care of the onerous and error-prone work such as the communication between nodes, memory address translation, and data motion. PGAS attempts to combine the advantages of an SPMD programming style for distributed memory systems and the data referencing semantics of shared memory systems.

HPF (High Performance Fortran) [10] is a forerunner of PGAS. It is an extension of Fortran 90 with constructs that support parallel computing. Similar to PGAS, it is a global view language that allows users to express the global semantics of data structures. Operators on the whole data structure are defined, allowing concise and expressive programs to be written. It was popular on SIMD and MIDM style architectures. However, the chief disadvantage of HPF is its limited power of expression, which limited its further application.

There are a set of languages using the PGAS model as the basis. One such language, Unified Parallel C (UPC), originally developed by Lawrence Berkeley National Laboratory and others, is an extension of ISO C that boasts multiple proprietary and open source compilers [41]. Another PGAS language, Titanium [42], centered at Berkeley is a dialect of Java designed for high-performance scientific computation. Compilers for both these languages use a source-to-source compilation strategy that translates the parallel languages to C with calls to a communication layer called GASNet. Beyond UPC and Titanium, there are X10 [43] developed by IBM, Fortress developed by Sun (now Oracle), Global Array [44] developed by Pacific Northwest National Laboratory (PNNL), and Chapel developed by Cray Inc., etc. Among these languages, Chapel [1] and X10 [43] also support asynchronous task creation, which is referred to as asynchronous partitioned global address space (APGAS) [45].

Currently, many PGAS languages offer comparable or even better performance in the single-node systems compared against OpenMP but suffer a great performance loss on multiple-node parallel architectures. There has been a continuing effort made to improve the performance of PGAS languages. For example, the Chapel team has been putting a major effort on correctness instead of performance of programs over the past ten years. Obviously, to make those languages successful, the performance issues must be solved. To tackle the performance problems, a good profiler provides an important first step.

2.2 Chapel

Chapel [1] is an emerging parallel language whose design and development have been led by Cray Inc. Chapel supports a multithreaded execution model, permitting the expression of more general and dynamic styles of computation than the typical

Single Program, Multiple Data (SPMD) programming models that became dominant during the 1990's.

Chapel has several features that are distinct from previous parallel languages and libraries. Chapel diverges from most HPC languages by supporting what its designers refer to as a global view of data structures and control flow. Chapel distributes variables as global data and accesses them using global indices, freeing programmers from calculating process-based subarray indices and local access ranges. On the other hand, Chapel also supports the global view of control, which refers to the fact that a Chapel program begins executing using a single task and then introduces parallelism through the use of additional language constructs. This is in contrast to SPMD programming models in which users write their program under the assumption that multiple copies of *main*() will execute simultaneously. Consider Figure 2.1:

```
writeln("The original task print this");
begin {
  writeln("A seoncd task is created to print this");
  compute();
  writeln("The second task ends");
}
writeln("This may print before the 2nd task completes").
```

Figure 2.1: Code snippet for Chapel "begin" keyword

Here, the *begin* keyword creates a new task that will execute the statements within braces. The original task goes on to execute the statements that follow. In this way, Chapel creates a flexible and asynchronous parallelism. Synchronization variables and *sync* statements are used to ensure the task synchronization whenever necessary.

```
cobegin {
  producer(numUpdates);
  consumer();
}
writeln(“Print until producer and consumer are done”);

coforall elem in Mat do
  processElement(elem);
writeln(“Print until all elements have been processed’);
```

Figure 2.2: Code snippet for Chapel “cobegin” and “coforall” keywords

The above two statements (*cobegin, coforall*) in Figure 2.2 create groups of tasks in a structured manner. *cobegin* is a compound statement in which a distinct task is created for each of its component statements. The original task also waits until its child tasks complete before proceeding. The *coforall*-loop is like a traditional for-loop except that it creates a distinct task for every loop iteration. Like the *cobegin* statement, *coforall* has an implicit join that causes the original task to wait for its children to finish before proceeding.

The data parallelism features of Chapel provide a more abstract and implicit style of parallelism, consider the example in Figure 2.3:

```
const MatSpace: domain(1) = {-N..N};
const MatSpaceD: domain(1) dmapped block(boundingBox=MatSpace);
var Mat: [MatSpace] real;
var MatD: [MatSpaceD] real;

forall m in Mat do
    m = 0;
forall m in MatD do
    m = 0;
```

Figure 2.3: Code snippet for Chapel “domain”

The forall-loop is Chapel's central concept for expressing data parallelism. Forall-loops are similar to coforall-loops in that both are parallel variants of Chapel's for-loop. However, where a coforall-loop creates a concurrent task per iteration and a for-loop is executed serially by a single task, a forall-loop may use an arbitrary number of tasks to execute the loop. As a result, it may execute serially using a single task, or it can use any number of tasks up to the number of iterations (or even beyond, though that is unusual). For typical iterands, this choice is based on the amount of hardware parallelism available. *Domain* is a first-class language concept that represents an index set. In the examples above, there are two domains: *MatSpace* and *MatSpaceD*. Every Chapel domain is defined in terms of a domain map that specifies how to distribute the array indices. When no domain map is specified, like *MatSpace*, a default domain map is used, and it maps the domain’s indices and the array’s elements to the current node. For a multi-node system, the second domain *MatSpaceD* distributes elements and array indices to all nodes. By comparing two forall-loops, you will see that Chapel provides a global view data distribution by using domains and locales so that programmers can only change the domains they use rather than modifying process-specific array indices and subarray accessing methods. Another important feature is that Chapel supports the expression of locality. Chapel was designed to execute on the largest-scale parallel machines where locality and affinity are crucial for performance. Locality features provide control over where tasks execute so that users can explicitly leverage the data locality and this feature largely contrasts HPF [10] in the effective expression of data locality.

```
on Locales[1] do
    writeln("I'm on Locale 1");
```

Figure 2.4: Code snippet for Chapel "on" keyword

The core feature for Chapel's locality feature is the locale type. For most conventional parallel architectures, a locale tends to describe a compute node, such as a multicore or SMP processor. The *on*-clause in Figure 2.4 is used to specify that a statement should execute on a specific locale. *Locales* is a built-in array representing system resources that can be queried directly in the user code. Combining locality manipulation and task parallelism, programmers can create tasks and access data in local or any remote nodes from a global view.

Moreover, Chapel is an elegant language because of the conciseness in expression. In practice, an implementation of a popular benchmark LULESH takes just 1,288 lines of code (plus 266 lines of comments and 487 blank lines), while the corresponding C+OpenMP+MPI version is nearly four times bigger [2].

2.3 Existing HPC Performance Tools

The prerequisite for optimizing an application is to understand its execution characteristics. There are many established performance tools that measure and analyze program performance on parallel architectures, ranging from simple shell utilities, timers, and profilers, trace analysis tools, to sophisticated full-featured graphical toolsets.

This section presents a brief review of the recent efforts made in mapping performance measurement data to a different level of abstractions in the program. I start from the general profilers for HPC applications to very specific profilers that can analyze certain PGAS languages. I also summarize a few tools having a data-centric way of viewing the profiling results.

2.3.1 HPC Profiling Tools

Much prior work has analyzed High Performance Computing (HPC) applications, based on different profiling methods such as simulation, sampling, and direct instrumentation. For instance, the Tuning Analysis Utilities (TAU) from the University of Oregon [3] is one of the most popular profiling tools; the popularity comes partly from the fact it is available on most platforms and supports a variety of languages, including FORTRAN, C, C++, Java, and Python. TAU also handles language extensions such as OpenMP and MPI implementations on supported platforms. The framework for TAU has three layers: Instrumentation, measurement, and analysis. The instrumentation is primarily source-based, but the binary instrumentation is also supported by DyninstAPI [19].

Scalasca [4] is a performance toolset that has been specifically designed to analyze parallel application execution behavior on large-scale systems with many thousands of processors. It offers an incremental performance-analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. Distinctive features are its ability to identify wait states in applications with very large numbers of processes and to combine these with efficiently summarized local measurements.

Vampir provides interactive visualization and exploration of parallel event traces [5]. It consists of the run-time measurement system VampirTrace and the visualization tools Vampir and VampirServer.

HPCToolkit [9] is another popular performance analysis system in the HPC field. It is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to the world's largest supercomputers. It relies on periodic sampling to capture the dynamic runtime behavior of parallel target applications. It also uses PAPI [20] library to read hardware counters as the performance metrics.

Some profilers focus on solving a very specific performance issue in HPC. ThreadSpotter [6] is a commercial tool that focuses on the memory access behavior of applications, specifically on how the application's memory access patterns interact with processor caches. With ThreadSpotter, users can sample an application from the beginning to the end, or attach to the application while it is running and sample it for a while then detach. It calculates cache metrics, such as cache miss ratios, cache fetch ratios, cache line utilization, and hardware prefetching probabilities. It can operate only at the binary code level, but the source code is needed to map performance data back to source lines. However, the responsibility of interpreting the performance information and using it to optimize the application lies with the programmer.

MemProf [7] instruments thread and memory management operations with a user library and a kernel module. It leverages AMD's Instruction-Based Sampling (IBS) [76] to associate latency with data structures to identify costly memory accesses to remote sockets. MemProf exclusively focuses on the NUMA locality problem. It allows precise identification of the objects that are involved in remote memory access and corresponding causes. However, there are also some limitations to this tool: first, it relies on programmers to establish a diagnosis and devise a solution; second, it is mostly used for applications that are not cache-efficient and perform a large number of memory accesses; third, it records a trace of each IBS sample and variable allocation rather than collapsing it on-the-fly into a compact profile. The resulting high data volume makes it hard to scale to a cluster with a large number of nodes. Finally, it doesn't map performance metrics to individual static variables; instead, it treats all static variables from a load module as one group and coarsely attributes metrics to these groups.

Buck and Hollingsworth developed CacheScope [8] to perform a data-centric analysis on Itanium 2. The Itanium processor provides a set of event address registers (EARs) that record the instruction and data address of data cache misses for loads, the instruction and data address of data TLB misses, the instruction addresses of instruction TLB and cache misses [24].

2.3.2 PGAS Profiling

Only a few profilers partially support PGAS programs and fewer have data-centric features. Tallent and Kerbyson [11] proposed a method to profile PNNL's Global Arrays based on HPCToolkit [9]. Their tool provides a code-centric, data-centric and time-centric view, using a hybrid tracing and profiling approach. It shows reads and writes of global data objects with respect to time, rank, and calling context. It will attribute performance metrics such as "bytes accessed per read" and "average latency per blocking write" to global data objects in their full static and dynamic context. To collect profiles and traces at scale and with minimal overhead, it combines the sampling of global reads and writes with the sampling of program behavior.

Parallel Performance Wizard (PPW) is another tool that supports both UPC and SHMEM. One important feature of this framework is the use of generic operation types instead of model-specific constructs whenever possible. Thus it has the potential to support multiple PGAS languages [12]. To accommodate the many instrumentation techniques appropriate for various PGAS model implementations, without introducing multiple measurement components, the developers proposed an Instrumentation-Measurement interface called the Global Address Space Performance (GASP) interface. For each model, an event type mapper maps the model-specific constructs to their appropriate generic operation types. By using the GASP interface, a potential disadvantage of PPW is that PPW requires a PGAS application to be compiled with a GASP-aware compiler.

Seisei Itahashi, et al., from the University of Tokyo, is working on a profiler for X10. It consists of a profiler and visualizer; the profiler is a modified version of the x10 compiler built with Polyglot. It is a source-to-source translator that inserts probe code into a target x10 program code. It supports analysis of activities involved in synchronization, but not on implicit data transfer. For example, unlike a typical MPI program, some data transfers in X10 are implicit. When an activity is moved among places, data transfers are implicitly executed (specifically, all local variables and arrays declared before the move will be copied and transferred to the destination place), but not explicitly described in the source code. It aims at better visualizing the implicit behavior in X10 [13].

Sebastian, et al. is working on extending Vampir to support the OpenSHMEM standard for parallel programming [14]. They proposed a theory about the mapping of OpenSHMEM communication primitives to generic event records that is compatible with a range of PGAS libraries. They also demonstrated an experimental extension for Cray-SHMEM in VampirTrace and Vampir and first results with a parallel example application.

2.3.3 Chapel Performance Analysis

As for profiling Chapel code, there are few established performance analysis tools that have deep integration with Chapel language features. The TAU suite [3] demonstrated its support for Chapel with a simple program [15], but there are no papers about profiling Chapel production codes using TAU published as far as I know. HPCToolkit [9] can be used to profile the Chapel runtime library, but it does not associate the work offloaded to worker threads to the full calling context at the source level.

For multi-threaded programs, it is especially important to have the full calling context of performance bottlenecks in the code because otherwise, programmers will probably lose track of root causes of the performance issue and miss opportunities for optimization. Let's say you have a function *bar* which is called in many different places and consumes a large portion of time during the execution. However, the time spent on *bar* is determined by actual parameters it receives and most time it consumes comes from certain bad calls. Therefore, without a full calling context, it will be very difficult to find the real problem. Besides, the tasks completed by worker threads are usually wrapped into several task functions with generated function names; simply attributing performance data to those functions would be of little help to users in locating problematic user code blocks. Pprof from Google's gperftools [16] partially supports the traditional code-centric profiling of Chapel. It is code-centric and more useful in profiling the runtime library from a Chapel developer's perspective. Consider Figure 4.10 and its following explanation, the output of Pprof on LULESH [74] shows it's insufficient for end-user purposes.

HPCToolkit-Data-Centric component [17], derived from the original HPCToolKit, has been used to profile several HPC benchmarks, either for single-locale or multi-locale environments. Nonetheless, it only tracks the memory allocation and deallocation of static variables (data allocated in the .bss section in load modules), heap-allocated variables that have size over 4K bytes, and no local variables. Therefore, it lacks complete data-centric information of the whole program. Additionally, after the Chapel compiler's translation, the global variables in Chapel source code are not treated as "static data" in the view of HPCToolkit's data-centric component. Therefore, most variables in Chapel benchmarks are regarded as "unknown data", which cannot provide useful information to programmers. Chplvis [26] is built for Chapel programmers. It visualizes the inter-locale communication and task computation of Chapel programs that help the user to discover the pitfalls of certain uses of parallelism in their code. However, it needs source modifications and it shows only the phenomena but not the causes of performance issues.

Besides the automated performance measurement tools, there exist several studies of the performance of Chapel. Chamberlain et al. studied Chapel's performance with HPCC benchmarks including STREAM as well as random access and FFT. They report that STREAM performance was near optimal at the date of writing [22]. Nan and Kenjiro analyzed Chapel performance and compared single-locale Chapel execution to C execution; they report that Chapel performance can get as close as 70% performance as C [46]. A study by Chamberlain et al. details what productive features a PGAS language should have and gives examples of Chapel's such features [1]. Johnson and Hollingsworth also conducted several case studies of Chapel's performance for single-node environments [21]. They chose OpenMP as a point of comparison and hand-tuned the generated code of four competitive Chapel benchmarks and gained a speedup of up to 6x. Most recently, a study by Kayraklioglu and El-Ghazawi examined several language optimizations provided by Chapel on a set of benchmarks using multiple locales and analyze their impact on programmer productivity quantitatively [47]. The optimization methods that they studied achieved improvements over non-optimized versions ranging from 1.1 to 68.1 times depending on the benchmark. Haque and Richards [55] implemented CoMD in Chapel. They demonstrated that optimizing data access through replication and localization is crucial for achieving performance comparable to the reference implementation. They discussed limitations of existing scope-based locality optimizations and argue instead for a more general (and robust) type-based approach. It's because Chapel's type system currently lacks a notion of locality i.e. a description of an object's access behavior in relation to its actual location. This often necessitates programmer intervention to avoid redundant non-local data access. Moreover, due to insufficient locality information, the compiler ends up using "wide" pointers that can point to non-local data for objects referenced in an otherwise completely local manner, adding to the runtime overhead.

2.4 GPU-accelerated Computing

Emerging supercomputers are increasingly employing GPU accelerators and integrated many-core devices. Not only do these GPU-accelerated systems deliver higher performance than their counterparts built with conventional multicore processors alone, but these accelerated systems also deliver improved energy efficiency because they are optimized for throughput and performance per watt and not absolute performance [65]. Nowadays, GPGPU (general-purpose computing on graphics processing units) is used to speed up parts of applications that require intensive numerical computations. In 2003, Mark Harris recognized the potential of using graphical processing chips for general purpose applications and started gpgpu.org to while he was still a Ph.D. student at UNC for those working in the field to share and discuss their work [50]. Traditionally, these parts of applications are handled by the CPUs but GPUs have floating points arithmetic rates much higher than CPUs. The reason why GPUs have FLOP rates much better than multicore CPUs is that the GPUs are specialized for highly parallel intensive computations and they are designed with much more transistors allocated to data processing rather than flow control or data caching [51]. As the use of such GPU-accelerated computing systems increases, it has also motivated researchers to develop new techniques to analyze the performance of these systems. To date, much work on performance analysis of heterogeneous systems focuses on data copying and communication optimization between GPU and CPU, I am more interested in kernel optimization.

2.4.1 GPU

GPUs (Graphics Processing Unit) were designed as a specialized integrated circuit to handle graphics processing, video decoding, image rendering, and shading, etc. With the GPU's rapid evolution from a configurable graphics processor to a programmable parallel processor, GPUs are increasingly used in scientific computing. Today's GPUs use hundreds of cores executing tens of thousands of parallel threads to rapidly solve large problems that have substantial inherent parallelism. They're now the most pervasive massively parallel processing platform available, as well as offer the most cost-effective for those applications that can effectively use them [52]. Figure 2.5 from NVIDIA website [53] shows the importance of GPUs in continuing the Moore's Law in microprocessor development.

GPU computing may be the path forward for HPC. NVIDIA powers the most advanced systems in Europe and Japan. U.S.-based Summit is the world's fastest supercomputer, with over 200 petaFLOPS for HPC and 3 exaOPS for AI. Summit includes over 27,000 NVIDIA Volta Tensor Core GPUs [56].

The importance of GPU acceleration is escalating as the HPC applications are more data-intensive. Data parallel programming paradigm can be found in many applications like linear algebra routines, computational biology, computational finance or econometrics. All these applications can obtain high speedups by mapping data elements to GPU threads that are executed in parallel.

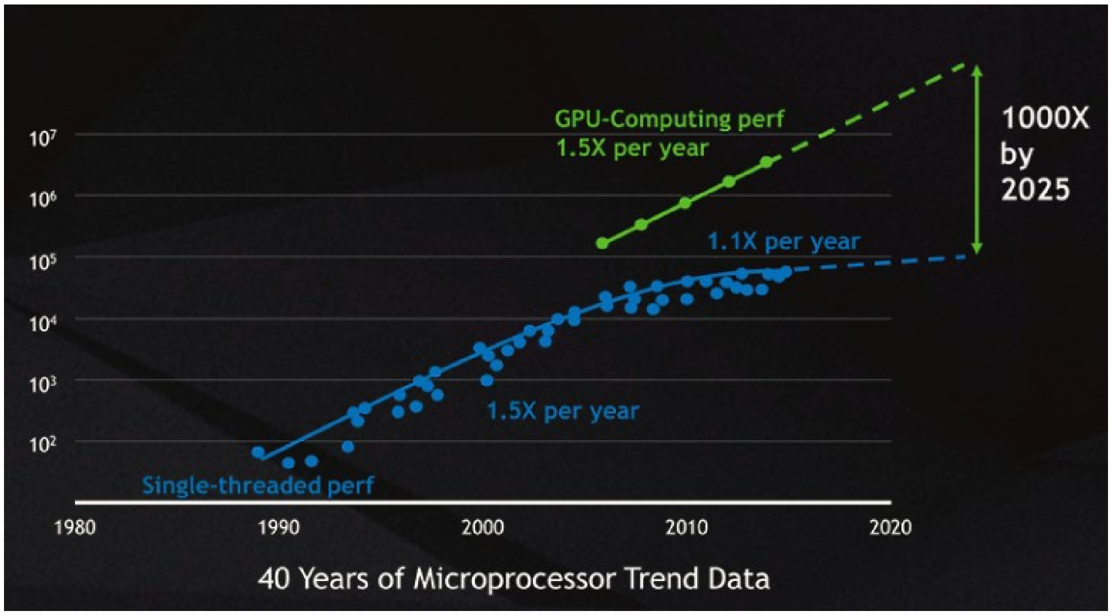


Figure 2.5: Microprocessor Performance Growing Trend over 40 years

2.4.2 CUDA

GPUs are designed to solve problems that can be formulated as data-parallel computations – the same instructions are executed in parallel on many data elements with a high ratio between arithmetic operations and memory accesses. This is similar to the SIMD approach. CUDA (Compute Unified Device Architecture) was introduced in 2006 by NVIDIA [54]. It is a general purpose parallel programming model that uses the parallel compute engine in NVIDIA GPUs to solve complex computational problems. At the time of its introduction CUDA supported only the C programming language, but nowadays it supports FORTRAN, C++, Java, and Python. The CUDA parallel programming model has three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions are exposed to the programmer as language extensions. They provide fine grain data parallelism and thread parallelism together with task parallelism that can be considered as coarse-grain parallelism. The CUDA parallel programming model requires programmers to partition the problem to be solved into coarse tasks that can be independently executed in parallel by blocks of threads and each task is further divided into finer pieces of code that can be executed cooperatively in parallel by the threads within the block. This model allows threads to cooperate when solving each task, and also enables automatic scalability.

GPU programming needs to explicitly handle data movement between CPU and GPU memories and know GPU hardware limitations, such as the GPU memory capacity and the number of registers, to effectively utilize GPUs. For most entry-level CUDA programmers, not getting good speedups or sometimes even worse performance than pure-CPU programs is a common issue. The synchronization between threads, the overlapping of computation and data movement, and poor kernel performance are major problems in CUDA programming.

2.4.3 GPU Profiling

GPU-accelerated computing is becoming the mainstream for modern supercomputers. Therefore, the performance analysis of hybrid architectures becomes more critical. Generally, there are two types of GPU profilers: one focuses on kernel-level performance analysis and the other is more system-wide, exploring potential benefits by coordinating CPU and GPU tasks more effectively.

With regard to profilers that are focused more on kernel performance, NVIDIA has provided Nvidia Visual Profiler (NVP) [57] using measurement-based approaches. NVP logs the execution characteristics of each GPU task, including method name, start and end times, launch parameters, CPU status (idleness or in-execution) and

GPU hardware counter values, etc. Some useful functionality, such as Unified CPU and GPU Timeline, CUDA API trace, Power and thermal profiling and Guided Application Analysis provide users with abilities to trace the execution of the entire program and identify inefficiencies in computation/communication overlapping, synchronization, and load imbalance problems. Besides the visual profiler, Nvidia also has a command line based profiler – nvprof, which can produce most of the same performance information as the visual profiler, and generate runtime data that can be fed into the visual profiler if desired. Although NVP is the most popular GPU/CUDA profiler, it lacks some features, such as source-level, in-depth analysis of kernels, mapping the performance issues to specific variables and functions, and the complete user-level calling context, which can be of great use to CUDA programmers.

S. S. Baghsorkhi and I. Gelado, et al. [58] use modeling and simulation to provide insights into the performance of individual kernels. The proposed analysis is based on memory traces collected for snapshots of an application execution. A trace-based memory hierarchy model with a Monte Carlo experimental methodology generates statistical bounds of performance measures without being concerned about the exact inter-thread ordering of individual events but rather studying the behavior of the overall system. The statistical approach overcomes the classical problem of disturbed execution timing due to fine-grained instrumentation. However, simulation-based methods have an inherent limitation that they do not reflect the actual execution profiles. S.J. Pennycook and S.D. Hammond, et al. [59] presented a performance study of a port of the LU benchmark from the NAS Parallel Benchmark suite [60] to CUDA. They conducted comprehensive performance comparison among a selection of GPUs, ranging from workstation-grade, commodity GPUs to NVIDIA's HPC center products, as well as between the CUDA port and the original FORTRAN 77 implementation. They also compared GPU cluster performance to that of existing CPU clusters using performance modeling. The analytic model they employed is a reusable model of pipelined wavefront computations by Mudalige, et al. [61], which abstracts parallel behavior into the generic model. Jaewoong Sim and Aniruddha Dasgupta, et al. [62] developed a performance analysis framework for identifying potential benefits when applying several commonly-used optimizations in GPGPU applications. They first develop an analytical performance model – the MWP-CWP model [63] that can precisely predict performance and provide user-interpretable metrics. Then they apply static and dynamic profiling to instantiate the performance model for a particular input code and show how the model can predict the potential performance benefits gained from each independent optimization or combined optimizations. This static and dynamic combined approach resembles my tool in understanding the root causes of slowdowns, but their model-based technique cannot be applied to all cases. Their performance advisor is an interesting component, which estimates the potential gains from reducing or eliminating these bottlenecks.

With regard to performance tools focused on both GPU and CPU activities, several interesting works have been done. The NVP [57] provides coarse CPU activity information with a unified timeline with GPU activities, which also indicates the potential benefits of adjusting the order and the workload of CPU tasks and GPU tasks. G-HPCTOOLKIT [64] characterizes kernel behavior by looking at idleness analysis vial blame-shifting and stall analysis for performance degradation. It quantifies CPU code regions that execute when a GPU is idle, or GPU tasks that execute when a CPU thread is idle, and accumulate blame to the executing task proportional to the idling task. Their approach does a good job in identifying the root cause of slowdowns instead of simply the performance phenomena and the use of sampling-based method enables it to scale for real applications. Coincidently, my tool also uses the concept of “blame”, but with a completely different meaning and quantification. The Vampir performance analysis toolset [5] keeps track of program executions on heterogeneous clusters. VampirTrace monitors GPU tasks using CUPTI [66], logging information including kernel launch parameters, hardware counter values, and details about the memory allocations. For CPU activities, Vampir traces the entry/exit of functions like TAU [3, 75] does.

TAU is another performance analysis toolset that has support for hybrid architectures. It employs more specific instrumentation for performance measurement, which could incur high overhead. Robert Lim [67] presented extensions to TAU that characterize the behavior of GPU application kernels and their performance at the node level. It also uses CUPTI sampling function to sample the instruction mixes for kernel execution runs, which reveals a variety of intrinsic program characteristics relating to computation, memory and control flow. The work demonstrates the effectiveness of their proposed techniques with two case studies on a variety of GPU architectures. However, they did not identify any optimizations based on the insights they obtained.

Chapter 3

Blame Definition

My data-centric approach builds on a performance mapping technique called "variable blame", proposed by Nick Rutar [18]. A variable's blame is a percentage that indicates the share of certain performance metrics, such as time, cache misses, or I/O operations due to individual variables.

Blame is an inclusive data-centric method that utilizes the control flow and full data flow information to map performance data to variables in the source code. During the runtime, I use event-driven sampling. If a sample is triggered for an instruction that is part of the data flow of a given variable, then that particular variable will be blamed for the sample.

Blame is particularly useful in analyzing scientific computing applications that have multi-level abstractions and complex data structures. These objects are often inherently distributed and contain calls to message passing and third-party libraries that are mostly hidden from users, masking both data motion and parallelism. Such hiding makes it easier for programmers to write parallel programs but also far more difficult to diagnose performance issues. Blame is a profiling tool that can attribute performance data to these abstractions, from the very bottom internal data to higher level concepts. The following sections explain the most important parts in the blame calculation using a single-thread program as an example.

3.1 Blame Calculus

Formally, "blame" is presented in terms of values for each variable for one run of a program. Let *S* be the set of all samples gathered during the run of the program. For a given sample *s* within *S*, *W* is the set of all statements containing a write to the memory region allocated to the variable *v*, the aliases of *v*, and all fields of *v*. For a structure, this includes all sub-fields within the hierarchy of *v*. The blame set for *v* is the union of all the statements in the backward slices [38] for each of the statements in set *W*:

$$BlameSet(v, W) = \bigcup_{w \in W} BackwardsSlice(w)$$

Variable *v* is blamed for sample *s* in the cases where *s* is a member of the BlameSet(*v*, *W*). The result is computed with the following function:

$$isBlamed(v, s)\{if\big(s \in BlameSet(v)\big)\ then\ 1\ else\ 0\}$$

The blame percentage for a variable for the entire program is the number of samples that are attributed to a particular variable divided by the total number of samples. This is calculated by the following formula:

$$BlamePercentage(v, S) = \frac{\sum_{s \in S} isBlamed(v,s)}{|S|}$$

After computing the BlamePercentage of variable *v*, I can declare that *v* is responsible for that fraction of whatever performance metric I chose to generate the samples. For example, if I chose CPU clock cycles as the performance metric and the

BlamePercentage was x, I can declare that *v* was responsible for the x fraction of all CPU clock cycles consumed over the entire course of the program execution.

In order to properly calculate the blame of each variable, I need a set of rules that dictate how the blame is transferred from one variable to another. Blame can be propagated through variables in three ways: explicit transfer, implicit transfer, and transfer functions.

3.1.1 Explicit Blame Transfer

Explicit blame transfer occurs when there is an explicit data dependency between variables. This is the most common transfer that's reflected by most assignment operations. For instance, consider the C code below:

```
int a, b, c, d;
a = 6;
b = 8;
c = a + b;
d = a + c;
```

Figure 3.1: Code snippet for Explicit Blame Transfer

From the snippet, I see two variable dependencies from two assignments (I do not care about constant values, so the first two assignments are ignored). Clearly, the blame of *a* and *b* will be explicitly transferred to c because the values of *a* and *b* are calculated directly for the purpose of having their sum stored in *c*. The last assignment causes *d* to have all the blame of the snippet because there is an explicit transfer between *d* and *c*, *d* and *a*, even though *b* does not appear in the direct assignment of *d*, *b*'s blame will also be indirectly transferred to *d* through *c*. The variable *d* may be used in another computation and will be subsequently included in the BlameSet of the variables that use it.

3.1.2 Implicit Blame Transfer

Implicit transfer is a little more complicated. It happens when there is no direct value assignment between two variables, but there exists a variable used in a control dependency. For example, a loop index is incremented for every iteration but is never explicitly involved in any calculation in the loop body. However, all variables that are within the loop body will inherit the blame from the index variable. The same situation happens to the standard conditional statements.

```
int i, a, b
for (i=0; i<10; i++) {
    a = 6;
    if (a>7)
       b = 8;
    else
       b = 9;
}
```

Figure 3.2: Code snippet for Implicit Blame Transfer

In this case, the variable *i* is used as a loop index, even though it is not in the direct assignment of variable *a* or *b*, it would be assigned to both *a* and *b* through an implicit transfer. The variable *a* also acts as a conditional variable in the if statement, so even though there isn't a direct assignment from *a* to *b*, what value b will be assigned still depends on *a*'s value. Therefore, variable *b* and *a* also have an implicit dependency relationship between them; the blame of *a* needs to be transferred to b as well. Such dependency relationships are obtained by doing a control flow analysis of the program so that all variables inside a loop will have blame from the loop's index variables and variables inside a conditional statement will have blame from the conditional variable

3.1.3 Transfer Function and Exit Variable

The above two transfers reside within each function for the blame transferring between local variables and possible global variables used there, but what about the blame transferred through function parameters? I use a transfer function based on "Exit Variables" to properly propagate blame up along the call trace.

A transfer function serves as a link to transfer blame between the callee and the caller function in my performance data mapping system. To transfer blame between functions, "Exit Variables" of the callee are kept and used as intermediate transmitters to transfer blame to the caller's local variables. "Exit Variables" are those whose values can affect the program outside the scope of the analyzed function.

Categories of exit variables include:

- Parameters that can result in side effects outside the analyzed function (pass by reference, pointers) are eligible for exit variables. The cases where a pointer is passed in and has its elements read instead of written won't be counted since blame is transferred only through write operations.
- If return value exists, and it is assigned to a variable in the caller function, the transfer function is checked to see whether the blame needs to be transferred.
- Global and static variables (in C/C++ context) are handled by transfer functions as well, but they are not treated as other variables at a per-function level. They are hoisted to a program level and all blame from each function will be aggregated to the single instance of that variable.

3.2 Simple Example

This subsection demonstrates how to calculate the blame for each variable with a simple example. The example is written in C for better readability, not for those who are familiar with Chapel, but my tool handles Chapel programs besides C programs.

The BlameSet of each variable represents a set of line numbers, so whenever a sample point falls on that line during runtime, variables that have a BlameSet including that line will be blamed for the sample. I list the BlameSet for each variable after each step in Table 3.1. The sample program is displayed in Figure 3.3.

In this example, there are 4 variables of interest. Three of them are local variables: *i*, *temp*, *med* and the other one is a pointer parameter *x*, which is counted as an exit variable as well. Note that *x* is the name of a formal parameter; therefore its blame will be propagated to the real parameter, which could be a local variable in the caller or a global or static variable defined in the global space.

First, you can easily find out the lines for each variable that is either a declaration or a write to that variable. The variable *i* and *temp* both have line 7 as their declaration line, so 7 is included in the BlameSet of *i* and *temp*. Line 8 is both a declaration and an assignment for *med*, so 8 is included in *med's* BlameSet. Line 9 is the head of *a* for loop and *i* is the loop index and gets incremented for every iteration, thus 9 will be attributed to *i*. Line 10 is included in *temp's* since it's a direct write to *temp*. Line 11 won't be assigned to anyone for now since it's neither a declaration nor a writes to some variable, and neither will line 13. Line 12 is assigned to *x* and so is line 14. Line 16 won't be assigned to anyone either since it's just a read to *temp*.

```
5    int foo(int *x,  int increment)
6    {
7      int i, temp;
8      int med=increment;
9      for(i=0; i<N ;i++){          //Sample Point 1
10       temp = i+3;                //Sample Point 2
11       if(temp%2 == 0)
12         x[i] = med + 1;
13       else
14         x[i] = med - 1;          //Sample Point 3
15     }
16     return temp;
17   }
```

Figure 3.3: Example code for blame calculation

Table 3.1: Blame Calculation Result for the Example in Figure 3.3

Variable	BlameSet		
Name	declaration and writes	after Explicit Transfer	after Implicit Transfer
i	7,9	7,9	7,9
med	8	8	8
temp	7,10	7,9,10	7,9,10
x	12,14	8,12,14	7,8,9,10,11,12,13,14

Secondly, I analyze the explicit blame relationship between these variables and do the transferring accordingly. The variable *med* depends on the value of *increment*, which is a non-pointer parameter, so it is not counted as an exit variable. The variable *temp* depends on *i* in Line 10, so it will have everything in *i*'s. The same situation for *x* and *med's* will be merged to *x's* as well.

Thirdly, I have one *for* loop and one *if* statement in this function, so an implicit blame dependency exists. First of all, all variables that are written inside a *for* loop will be blamed for *i*'s; this includes *x* and *temp*. Then since *temp* is used as a conditional variable, *x* will also be blamed for each line in *temp's* BlameSet.

Lastly, I find that the function *foo* has a return value *temp,* which is an exit variable. The variable that gets assigned the return value of foo will accept all the blame assigned to *temp* within *foo* and as a result, counter lives outside the scope of *foo*.

The procedure that is described so far is a simplified blame analysis and it does not represent the real analysis' order and steps. Real blame analysis is far more complicated and works on an LLVM intermediate representation (IR) [48] instead of source code.

Now that I have the BlameSet for each variable, I can combine it with the given runtime data (samples) to calculate the final blame percentage for each variable. From Table 3.1, I notice that Sample Point 1 (short as "S1") falls on Line 9, which is an element of the BlameSet of the variable *i*, *temp* and *x*; therefore *i*, *temp* and *x* will be blamed for S1. S2 corresponds to line 10, which is included in *temp's* and *x's*, so these two variables are blamed for S2. S3 on Line 14 is only attributed to *x* since only x's BlameSet contains 14. According to the formula of blame I displayed in Section 3.1, I can calculate the blame percentage for each variable in *foo* as the following: 33% for *i*, 66% for *temp*, 100% for *x* and 0% for *med*.

For any given sample, multiple variables may be blamed. For example, S1 is attributed to the variable *i, temp,* and *x* as I explained before. Therefore, in a given function, the total percentage assigned to all variables can be more than 100%.

Chapter 4

Data-centric Profiling for Single-locale Chapel Programs

Chapel is an emerging PGAS (Partitioned Global Address Space) language whose design goal is to make parallel programming more productive and generally accessible. To date, the implementation effort has focused primarily on correctness over performance. I designed and implemented a performance tool *ChplBlamer* for single-locale[1] Chapel programs based on my data-centric idea and the blame metric. The same idea is also applicable to other PGAS models. I also included a case study on three well-known benchmarks and manually optimized the code with insights into the programs. The optimized versions improved the performance by a factor of 1.4x for LULESH, 2.3x for MiniMD, and 2.1x for CLOMP with simple source modifications. This chapter is adapted from a paper that has been presented at the International Parallel and Distributed Processing Symposium (IPDPS'17) [68].

4.1 Implementing the Tool

The framework of my data-centric profiling system is presented in Figure 4.1. It consists of 4 steps, combining the static (pre-run) information and dynamic (runtime) information of a binary, to map performance data to variables in the source code. My approach leaves most work to the static analysis and postmortem processing, to minimize the perturbation to the program execution. Step 1 runs intraprocedural static

[1] "Locale" is a Chapel abstraction that represents a compute node, such as a multicore or SMP processor in most typical parallel architectures.

analysis, including complete control flow analysis and data-flow analysis to get BlameSet for each variable in each function. Step 2 is the program execution under a monitor process. Step 3 is a postmortem processing step, which can be concurrently executed on each node in the environment when the tool is extended to support multi-locale. Step 4 is responsible for profiling data aggregation, processing, and presentation to the user via a GUI.

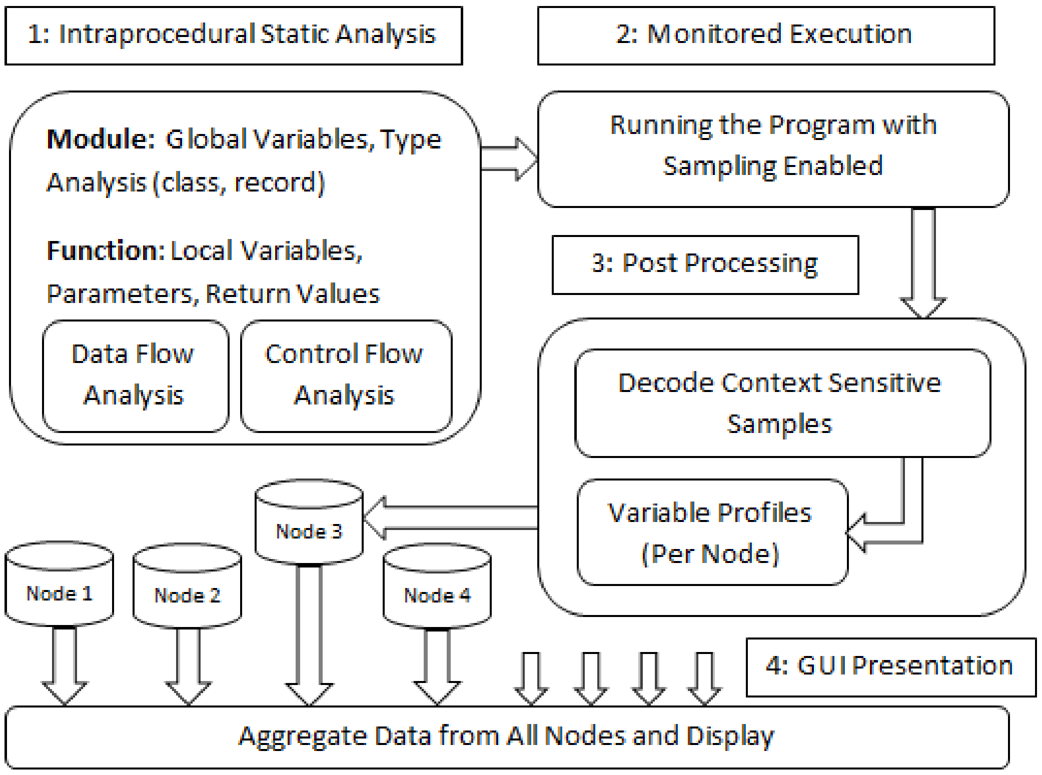

1: Intraprocedural Static Analysis
2: Monitored Execution
Module: Global Variables, Type Analysis (class, record)
Function: Local Variables, Parameters, Return Values
Data Flow Analysis
Control Flow Analysis
Running the Program with Sampling Enabled
3: Post Processing
Decode Context Sensitive Samples
Variable Profiles (Per Node)
Node 1
Node 2
Node 3
Node 4
4: GUI Presentation
Aggregate Data from All Nodes and Display

Figure 4.1: Process of Calculating the Performance Data for Variables

4.1.1 Debug Information Support for Chapel LLVM Frontend

My static analysis component runs an analysis pass on the LLVM Intermediate Representation (IR) of Chapel programs. LLVM [48] is a widely used language-agnostic compiler infrastructure, originally designed for C/C++, but later has spawned a variety of front ends: ActionScript, Ada, C#, Fortran, Haskell, Java bytecode, Objective-C, Python, R, Ruby, Rust, CUDA, Scala, and Swift, etc. The great advantage of LLVM is that you can write a language-specific front using its C++ library to get a language-independent intermediate representation, which is also hardware-independent. At this level, you have the same IR grammar and syntax for different languages; therefore you can apply the same compiler optimization passes to different languages. LLVM greatly simplifies the development of a compiler for a new language and also exposes language-independent optimization opportunities.

I chose to analyze the LLVM IR of Chapel for several reasons: first, LLVM supports a rich language-independent library for program analysis and transformation, so the implementation of my static analysis pass will be relatively easier and more generic; second, there are many existing LLVM-based optimization passes that can be plug-and-play for Chapel programs. For example, Hayashi, Akihiro, et al. [25] developed several LLVM-based communication optimizations for Chapel. They first enabled some existing LLVM optimization passes for Chapel's LLVM IR, such as Loop Invariant Code Motion (LICM) and Scalar Replacement; then they created Chapel-specific optimization "Aggregation", which combines sequences of loads/stores on adjacent memory location into a single *memcpy* operation.

Chapel had a basic LLVM frontend with very limited support for debugging. Originally, it was only able to produce the debug information of modules and functions from the Chapel standard modules. It did not produce any debug information for variables, function parameters, and composite types, nor did it produce any debug information for anything from the user code (including modules, functions, variables, etc.). However, I need the debug information to associate low-level instructions and memory addresses with source lines in the user code. Therefore, I implemented the complete debug information generation in the code generation pass of the Chapel compiler.

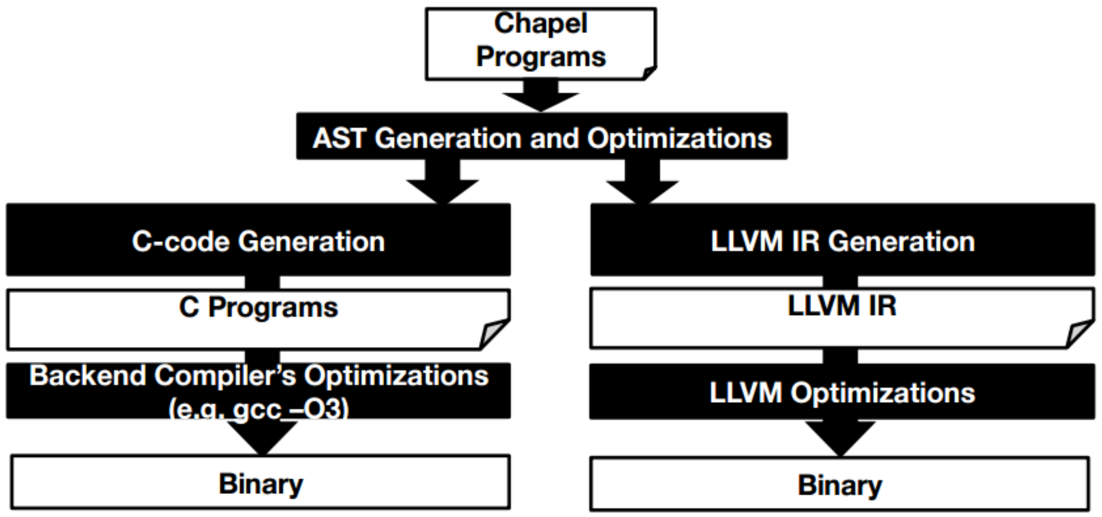


Figure 4.2: Chapel Compilation Flow

Figure 4.2 [39] shows two ways of compiling Chapel programs. The default method that Chapel compiler uses will first convert the user code and Chapel standard modules into C code and then use gcc to compile the generated C code and produce the ultimate binary. On the other side, the LLVM frontend is used to generate the LLVM IR for both the user code and standard modules and clang is used to build the binary. As I mentioned before, the debug information generation for the LLVM track is added during the code generation step of the compiler. Everything in the user code including compile units, variables, functions, types, instructions, etc. will have the corresponding debug information attached. Among all the debug information, the context information (including the file name and line number) of each program element is the most important metadata for my work since it is key to mapping blame data back to source code variables.

4.1.2 Static Analysis

The First step of my performance data mapping system is static analysis, which is the core part of my system. LLVM follows the SSA (Static Single-Assignment) convention for its registers. Registers are temporary holders of intermediate values represented by "%#", where "#" will be replaced by a unique integer in each function. LLVM represents variables from the original compiled code as memory locations, so no "MOVE" operations exist in LLVM; registers are used to move data to and from memory through load and store operations.

Although my complete analysis is interprocedural, I limit the analysis to intraprocedural at this point for two reasons. First, I need runtime information for interprocedural data-flow information and alias analysis. I can perform some interprocedural analysis before running the program, but it would be incomplete. Second, by limiting the static analysis to intraprocedural information, I can easily reuse analyses from run to run (or among programs that use shared libraries). Also, static analysis is parameter-independent, which means users are free to change real parameters and configurable arguments during execution without running the static analysis again.

Instruction Parsing

The blame calculation needs full dataflow information so that I have to thoroughly analyze all the instructions within a function one by one. While parsing instructions, I categorize each variable/register by their different roles in different types of instructions. I also need to store the information of function parameters to create exit variables, which will be used to create transfer functions. The details about transfer function are explained in Section 3.1.3.

Graph Representation

Variable blame is calculated based on dataflow interactions; therefore, to formalize the propagation of information, I use sets to present the elements removed or inserted in each category for each variable/register after parsing every instruction. It ends up with a BlameSet for each variable that associates them with blamed lines in the source code. Along with BlameSet, I have several auxiliary sets for each variable/register that are used in the data-flow analysis to transfer blame between variables. This leads to redundant data from set to set. I use a graph to represent the blame relationship between variables. I represent the dataflow interactions as edges within a graph, and variables and registers and function calls as vertices. The edges are directed, from the blamed vertices to blamee vertices. For example, if it is a load instruction, then the node representing the return value will have a directed edge to the node representing the address from where the load instruction gets its value.

To show how my graph representation encodes the blame calculus, here I will go through a simple Chapel code snippet shown in Figure 4.3 with the corresponding generated LLVM IR in Figure 4.4. Figure 4.5 shows the original graph representation and Figure 4.6 shows the compact graph that removes redundant data from the original.

```
var a : int = 6;
var b : int = 7;
var c : int = a + b;
```

Figure 4.3: Sample Code for Graph Representation

```
store i32 6, i32* %a_chpl
store i32 7, i32* %b_chpl
%0 = load i32* %a_chpl
store i32 %0, i32* %call_tmp_chpl
%1 = load i32* %b_chpl
store i32 %1, i32* %call_tmp_chpl2
%2 = load i32* %call_tmp_chpl
%3 = load i32* %call_tmp_chpl2
%4 = add i32 %2, i32 %3
store i32 %4, i32* %c_chpl
```

Figure 4.4: LLVM Instructions for the snippet in Figure 4.3

The LLVM IR in Figure 4.4 is a simplified version while the complete IR has many more instructions, making the generated graph far more complicated.

Consider Figure 4.5, the graph is constructed in the following way: the SSA registers generated by LLVM are presented with white ovals; the constant values are represented by rounded rectangles filled with light blue; the variables from user code or temporary variables generated by Chapel are represented with white rectangles. If there is a blame relation between vertex *a* and *b*, I add an edge directed from *a* to *b* if *a* is blamed for *b*.

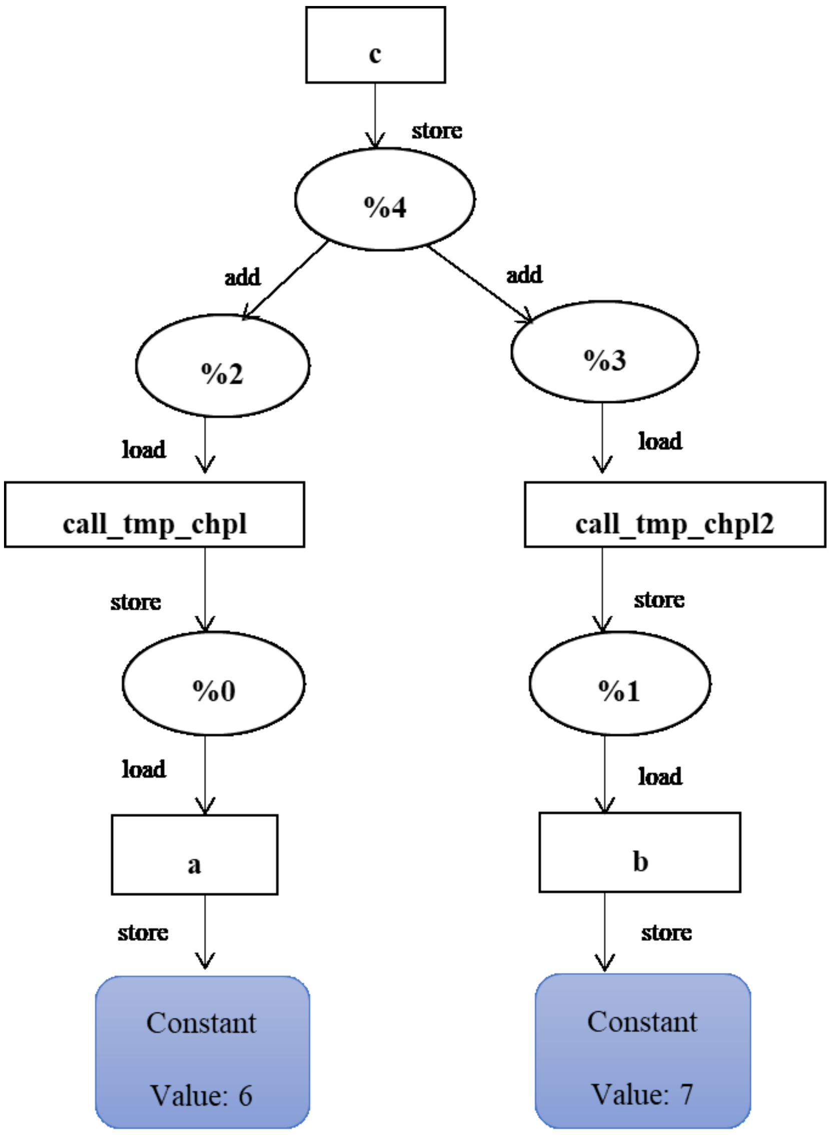


Figure 4.5: Original LLVM IR Graph Representation

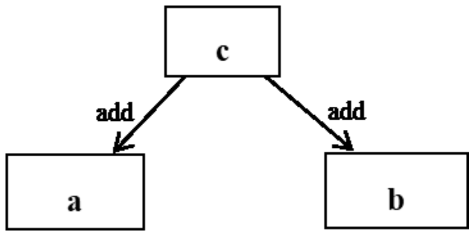


Figure 4.6: Compact LLVM IR Graph Representation

In most cases, the raw graph consists of a lot of redundant data and I can compress the graph to get a clearer blame relationship between variables from user code. I migrate vertices and edges to the compact graph once they meet certain criteria. Local variables and function parameters are migrated automatically. Any registers that are a pointer to one of the elements listed above are eligible to migrate. Any registers that are the receiver of a store operation are migrated as well. According to Chapel's naming pattern, variables starting with "*call_tmp_chpl*" will be recognized as temporary variables and will be removed. The deleted vertices need to transfer their connected edges to other remaining variables. After a variety of operations, I get a compact graph as shown in Figure 4.6.

Hierarchical Processing of Structures

The most important instruction of the LLVM IR in static analysis is the "getelementptr" (GEP) instruction [70]. The GEP instruction gets the address of a complex data structure or array but does not actually access memory. In GEP instructions, the first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers. The remaining arguments are indices of the elements of the aggregate object that are accessed. The interpretation of each index is dependent on the type being indexed. The first index always indexes the pointer value given as the second argument, the second index indexes a value of the type pointed to (not necessarily the value directly pointed to, since the first index can be non-zero), etc. The first type indexed must be a pointer value, subsequent types can be arrays, vectors, and structs [71]. A load instruction usually follows a GEP instruction to get the value from the calculated address. In some other cases, the retrieved address will be first stored into some other temporary variables and later loaded and de-referenced, in that case, I need to handle the aliasing relationship between the original GEP instruction and the ones that are loaded and used later. I do back-tracing along GEP and load/store operations to resolve the pointer relationships. In the graph representation, a GEP instruction is illustrated with several nodes representing the base, the field index, and field address, respectively.

The tool adds edges between each field and their structure base or "Parent" (this is a recursive process since a data structure can be multi-level so the "Parent" may have its own "Parent", which becomes the "Grandparent" of the original field representation). The nodes in the graph that actually point to the same field of a data structure will be collapsed to a single node so that the distributed blame to all the representations can coalesce. Later, a "Parent" can absorb all the blame of its "Children". In this way, the blame can bubble up along the data hierarchy. And if a user-defined high-level abstraction employs a low-level library data structure, the blame on that library structure can be reflected on the user-defined variable and provides programmers with insights for user-level optimization.

4.1.3 Runtime Information Acquisition

The execution step involves running the program under a monitoring process and generating raw sampling data.

The sampling mechanism uses hardware support via PMU (performance monitoring unit) that exists in most current processors. A PMU can be configured to trigger an interrupt when a marked event count reaches some threshold.

I use the PAPI [20] library as the interface to utilize PMUs. PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance hardware counter in most major microprocessors. When a PMU triggers a sample event, the profiler receives a signal and reads PMU registers to extract the precise instruction pointer of the sampled instruction. The marked PAPI event I used as the performance metric is PAPI_TOT_CYC. One issue with measurements involving interrupts is “skid”: once an overflow interrupt happens, it takes a CPU (especially modern complex out-of-order designs) some amount of time to stop the processor and pinpoint exactly which instruction was active at the time of the overflow. Often there is an offset between the instruction indicated versus the one causing the interrupt (this offset is called skid). PEBS [88] and IBS [76] provide support for low-skid sampling, at the expense of some additional overhead. Some previous work [49] has been done to avoid this problem by sampling instructions instead of events. I do not account for skid in my current implementation, but I plan to add a skid compensation feature in the future.

For each sample that is triggered during the program execution, I need to perform a stackwalk to get the call path using libunwind [82]. Libunwind is a portable and efficient C programming interface (API) to determine the call-chain of a program. The API supports both local (same-process, first-party) and remote (cross-process, third-party) operation. I used the remote/third-party stack walk by creating a monitoring process to ensure thread safety because the online stackwalk of a multi-threaded program within a signal handler is currently not safely supported by libunwind. Whenever the sampling process receives an overflow signal indicating it is time to take a sample, the monitoring process will perform a stack walk on the associated thread, which guarantees the memory space of the target process not being corrupted. To implement my monitoring process, I use DyninstAPI [19]. DyninstAPI is a widely-used Application Program Interface (API) that permits the insertion of code into a computer application that is either running or on disk. When running under Dyninst, all signals are first sent to the Dyninst monitoring process.

To support multi-threading, I instrument the Chapel tasking layer and enable the PAPI initialization whenever a new thread is created so the corresponding PMU can start counting on that thread. For parallel blocks in Chapel (i.e., forall and coforall), the master thread spawns worker threads to execute distributed tasks. To get the full call paths for the samples for each worker thread, I keep a unique tag for each spawn operation and record the stack trace before the spawn operation begins. It allows us to combine the pre-spawn stack trace with the post-spawn stack trace to produce a full call path for worker threads.

4.1.4 Postmortem Analysis

The postmortem analysis is the key step to combine the static information with the runtime information and produce a list of blamed variables for each sample. Several important passes in this step such as address parsing, full call path construction, and transfer function application are explained in the following subsections.

Address Parsing

First of all, I need to parse the raw sample data. Each sample is constructed as an "instance" object. Each instance consists of an instance index and a vector of frames obtained by walking the stack periodically during runtime. Each frame has three basic fields: frame number, address, and frame name (the name of the procedure that created the stack frame). I can resolve the raw instruction memory address to the precise file name and line number with the debug information.

Parsing is done not only for the samples generated by the sampling mechanism but also for pre-Spawn stack traces I got from the instrumentation in the Chapel runtime library. These partial stack traces are linked together to constitute full stack traces in the next step.

Glue Stack Traces and Assign Blame to Variables

To get a full call path for each sample, I need to glue the pre-spawn and post-spawn stack traces based on the unique spawn tag. All the context information, including module name, file name, line number, and frame number are stored in an abstraction named "instance" for each sample.

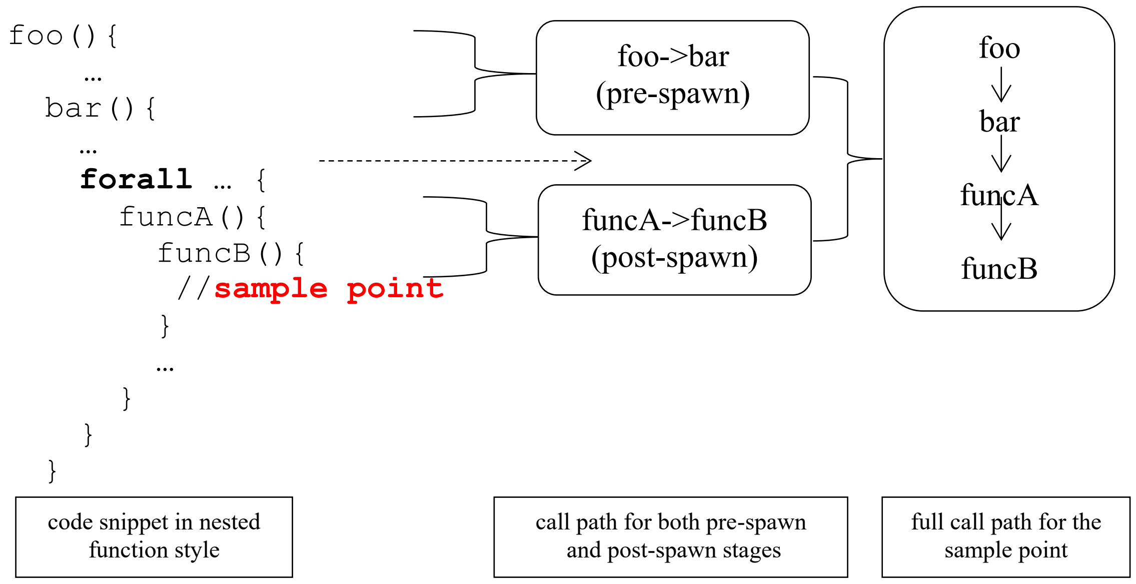

foo(){
…
bar(){
…
forall … {
funcA(){
funcB(){
//sample point
}
…
}
}
}
foo->bar
(pre-spawn)
funcA->funcB
(post-spawn)
foo
bar
funcA
funcB
code snippet in nested function style
call path for both pre-spawn and post-spawn stages
full call path for the sample point

Figure 4.7: Process of getting full call path for the sample inside a parallel block

Consider Figure 4.7 as an illustration of concatenating partial stack traces; the code is written in a nested function style to show the call chain in a more intuitive way.

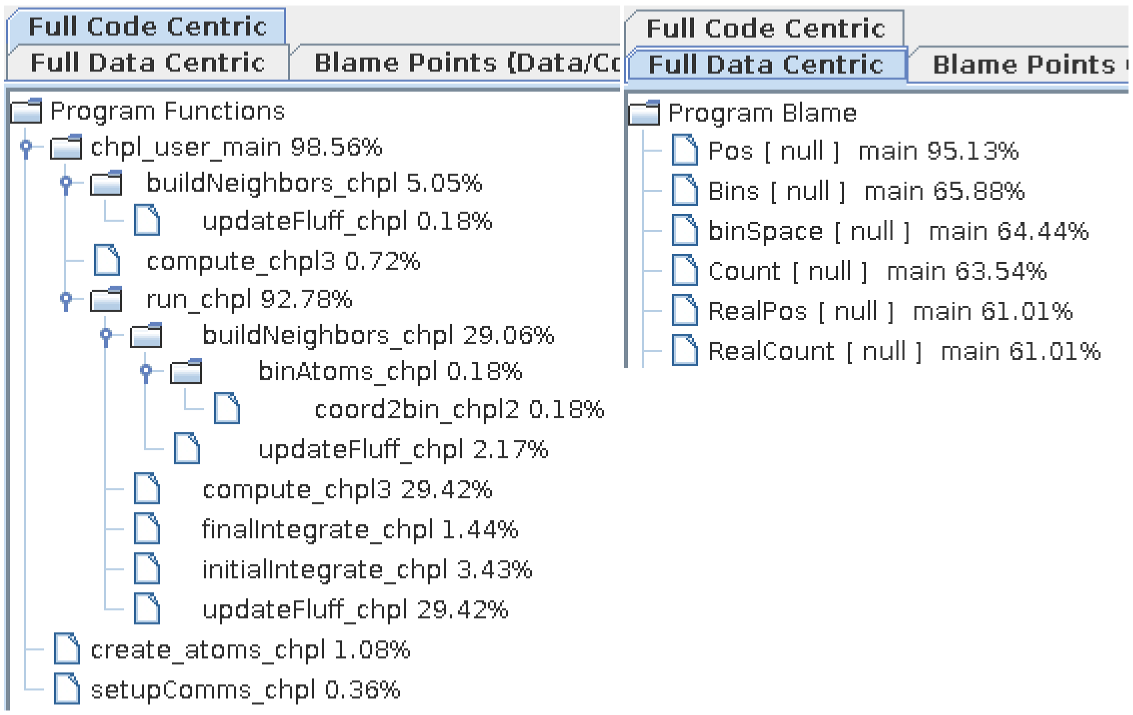
Later, I combine the stored intraprocedural analysis results with the runtime data ("instances") to determine the blamed variables for each sample. After resolving the addresses to functions and line numbers, I can utilize the predetermined exit variables to apply transfer functions at each level of the call trace. It means I can bubble the blame up as far as I need to assign blame to appropriate variables in each function along the call chain.

After this step, I have a list of blamed variables for each sample on the current compute node. For multi-locale programs, it will be one file per node to store the postmortem processing result and that part will be discussed in Chapter 5.

4.1.5 GUI Display

The last component of ChplBlamer is the Graphic User Interface (GUI). There are three different windows to view the data: a flat data-centric view, a traditional code-centric view, and a hybrid way. The flat data-centric view is the default window. It provides a flat view of all the variables defined in the program, ranked in descending order by the blame they are assigned to. The second view is a hybrid view based on "blame points". Blame points are points in the program deemed to have interesting variables; the most common one is the main function since variables in there cannot be bubbled up any further in the call stack. The third window is a traditional code-centric view that attributes samples to different functions instead of variables, in an inclusive way with complete calling context. Since I have processed all the context-sensitive samples in data-centric view, this code-centric view incurs almost no additional overhead.

In the data-centric view, I also show the declaration point of the variable, a call trace starting from the main function. For now, I do not distinguish local variables from global variables. All variables from all functions are laid out in the same window. Therefore, this location information is critical to identify the exact variables blamed in the code since local variables from different functions can share the same names and as well as the global variables. Figure 4.8 is a screenshot of the GUI with a flat data-centric and code-centric main display for one run of MiniMD. The right side is the unique data-centric result that my tool provides using blame analysis while the left side is the inclusive view of the regular sampling based code-centric result.



```
Program Functions
  chpl_user_main 98.56%
    buildNeighbors_chpl 5.05%
      updateFluff_chpl 0.18%
    compute_chpl3 0.72%
    run_chpl 92.78%
      buildNeighbors_chpl 29.06%
        binAtoms_chpl 0.18%
          coord2bin_chpl2 0.18%
        updateFluff_chpl 2.17%
      compute_chpl3 29.42%
      finalIntegrate_chpl 1.44%
      initialIntegrate_chpl 3.43%
      updateFluff_chpl 29.42%
  create_atoms_chpl 1.08%
  setupComms_chpl 0.36%
```

```
Program Blame
  Pos [ null ]  main 95.13%
  Bins [ null ]  main 65.88%
  binSpace [ null ]  main 64.44%
  Count [ null ]  main 63.54%
  RealPos [ null ]  main 61.01%
  RealCount [ null ]  main 61.01%
```

Figure 4.8: GUI screenshots of MiniMD

4.1.6 Exclusive Blame

So far as presented, the blame calculation is an inclusive data-centric profiling approach. This means that the blame value of a certain variable will absorb all the blame of its dependent variables. Therefore, the variables that hold the ultimate results will stand out in terms of weight. For example, variable *c* in Figure 4.9 has a blame value of 100% since the whole block of code is to compute the value of *c*. After examining the code, I find that the largest contribution to *c* is from *b* since *c* depends on *b* and *b* is responsible for all previous computation except for the last assignment to *c*. Therefore, optimizing the computation of *b* (line 2 to 4) may potentially provide better speedup than optimizing the single assignment to *c* (line 5).

```
1       a = 8;                     //Sample 1
2       b = a * a;                 //Sample 2,3
3       for (i = 0; i < N; i++)    //Sample 4
4          b = b + i;
5       c = a + b;                 //Sample 5
```

Figure 4.9: Code snippet for Inclusive and Exclusive Blame Calculation

Table 4.1: Inclusive and Exclusive blame calculation for Figure 4.9

Variable	a		b		c		i	
Result Type	inc	exc	inc	exc	inc	exc	inc	exc
BlameSet	1	1	1,2,3,4	2, 4	1,2,3,4,5	5	3	3
Blame Samples	s1	S1	S1,2,3,4	S2,3	S1,2,3,4,5	S5	s4	S4
Blame	20%	20%	80%	40%	20%	20%	20%	20%

To supplement the original inclusive blame, I provide another way of evaluating the weight of variables in terms of the potential of optimization: exclusive blame. Exclusive blame only attributes a line to a variable if there is a direct write to the variable at that line. Therefore, more blame will be aggregated to computation-intensive variables, where usually more optimization opportunities exist. Table 4.1 shows the process and result of exclusive blame calculation for the code snippet in Figure 4.9. Now the most blamed variable is *b*, I can quickly locate *b*'s first write statement with the multiplication as a potential performance bottleneck and optimize it. In the following subsections, only the inclusive blame results are presented.

4.2 Case Studies

I have chosen three Chapel benchmarks to demonstrate the utility of my tool. Two of them, LULESH [74] and MiniMD are from the Chapel source distribution, the third benchmark CLOMP, was ported by my group member Johnson [21] from the C version of the Livermore OpenMP benchmark on the Coral Collaboration Benchmark Codes website [71]. All experiments were done on a single locale. I used a 12-core SMP system, each is a 2.53GHz Xeon CPU. The Chapel version I used in experiments is 1.11. The threshold[2] I utilized with PAPI to trigger samples is 608,888,809, which is a large prime.

All programs were compiled with "--llvm --no-checks -g" (meaning the llvm frontend with no redundant boundary checks). Specifically, I did not use "--fast" (equivalent to

[2] Threshold is set to be a prime since it's important to avoid sampling bias caused when the sampling interval is a multiple of loop trim cause.

"-O3" in GNU compilers) since my intraprocedural analysis heavily depends on the generated LLVM bitcode of the Chapel program. Using "--fast" option in compilation would result in an LLVM intermediate representation with too many functions removed or renamed, variables optimized out and instructions reordered. These optimizations would make it nearly impossible to map the performance data from the IR nodes (temporary variables and registers) back to the source level variables with real names. A production version of Chapel should use augmented debug data to allow higher level optimizations. To validate that the information supplied by my tool *w/o* "--fast" provides useful guidance, I reran all of the original and optimized benchmarks with the "--fast" option and show that I get similar gains when using this option as without it.

As for the overhead of my tool, taking LULESH as an example: the typical cost per stack walk is 0.051ms while the interval is about 241ms (or a total overhead of 0.02%); the sizes of the datasets generated during runtime are 6MB to 20MB depending on the problem size; post-processing analysis takes an average of 16ms to process one sample.

4.2.1 MiniMD

MiniMD, short for "Mini Molecular Dynamics", is a proxy application from Sandia's Mantevo group. It represents key idioms of their real applications. Molecular Dynamics codes compute physical properties like energy, pressure, and temperature for a simulated space containing moving atoms. MiniMD was previously implemented in C++ using MPI and OpenMP, requiring about 5,000 lines of code, while the Chapel version only takes about 2,000 lines.

I picked this application for two reasons: first, it is an important strategic benchmark; second, it has several variables with multi-level data structures that are responsible for most computation so data-centric information is particularly useful. The problem size I tested for the benchmark is (16, 16, 16) unit cells (16,384 atoms). All other input parameters are pre-defined in the source by default.

I ran the test 10 times and reported mean values to eliminate run to run variance in the data. Variables with the largest blame values are listed in Table 4.2. First, I describe the roles of the variables that have a large blame. Then I explain how I used this information to optimize the program.

Table 4.2: Variables and Their Blame for THE Run of MiniMD

Name	Type	Blame	Context
Pos	[DistSpace][perBinSpace] v3	96.3%	main
Bins	[Space][perBinSpace] atom	84.2%	main
RealCount	[binSpace] int(32)	80.8%	main
RealPos	[binSpace][perBinSpace] v3	80.8%	main
Count	[DistSpace] int(32)	54.9%	main
binSpace	domain	49.4%	main

The record named "atom" is the most important data structure in the benchmark. It is an abstraction of atoms in the Molecular Dynamics simulation and contains two basic attributes: *velocity (v)* and *force (f),* both are *(x, y, z)* 3-D real values. It also includes the storage for the neighbor list, which stores the bin and index of a neighboring atom. Therefore, in the global space, which is initialized before the main function runs, I have the two most important variables: *Pos*, which is an array of positions, and *Bins*, which is an array of atoms.

Pos (96.3%): Pos serves as one of the root variables for the entire program. It stores all the position data of the atoms in the space. "v3" is a created type, using a 3*real tuple. *DistSpace* is a domain that defines the bounds of the arrays (*Pos, Bins*) and distributes them across locales in the multi-locale environment, while here it is simply the expanded domain of *binSpace*. *perBinSpace* is a one-dimensional domain. *Pos* takes a lot of blame because the positions are accessed and updated frequently in the program for the computation of the atom forces as well as neighboring *atoms'* attributes.

Bins (84.2%): The variable *Bins* is a collection of atoms based on spatial position. The benchmark is to simulate the space by calculating the attributes of each atom, thus this variable is read and written frequently and continuously throughout the entire program. The domains of this variable are basically the same as *Pos*, except the first domain '*Space'* is exactly equal to *binSpace* instead of *binSpace.expand()* in the single locale environment.

RealCount/RealPos (80.8%): These two variables are array aliases of *Count* and *Pos*, respectively. In Chapel, array slices alias the data in arrays rather than copying it. They are accessed and updated frequently in the time-critical code.

Count (54.9%): The variable Count is an array that keeps the count of bins in the space. For domain remapping reason, which I will address later, this variable is "written" (not at the source code level, but at the LLVM instruction level) during the main calculations, so is its array alias *RealCount*.

binSpace (49.4%): As I introduced earlier, *binSpace* is a domain whose range is determined by the problem size I set, which tells us the number of bins I need in each direction in the simulation space.

Discussion and Optimization: After a brief review of the benchmark source, I found three functions that handle most of the computation workload inside the real simulation function: *run*. They are *buildNeighbors*, *updateFluff*, and *ForceLJ.Compute*. Combined with my profiling results, it was discovered that the hot spots of these three functions are inside the nested for loop, where Bins and Pos are calculated after several domain remapping operations. The function *buildNeighbors* is used to put atoms into correct bins and rebuild neighbor lists; *updateFluff* is to update ghost information of Pos and Bins, and *ForceLJ.compute* is to compute forces between atoms.

The original code uses succinct zippered iteration expressions to do domain remapping in nested loops. Specifically, zippered iteration refers to a way in which Chapel for-loops can be driven by multiple iterands in a coordinated manner. However, based on my experience, that could produce significant overhead, especially in a large nested loop. Johnson's work [21] has done some optimizations on substituting for those zippered iterations in MiniMD. I applied their modifications to the source and obtained a significant improvement in the performance. The full details about the modifications can be found in Appendix A of [21].

The optimization opportunity found by Johnson [21], et al. was based on a manual performance analysis of the generated code, a complicated and painful process that required examining over 50 C source files mixed with user code and Chapel library code. It is very difficult to identify bottlenecks by hand and even harder to map them back to Chapel source code and then optimize them. Using my tool, programmers can quickly identify the problematic variables in the source code. In the case of MiniMD, by searching for the two most blamed variables, *Pos* and *Bins*, I was able to quickly locate those forall loops that contain zippered iteration and domain remapping. Based on my experience of the poor performance when using zippered iteration and domain remapping, I could apply those transformations to improve the performance.

Table 4.3: Results w/ or w/o "--fast" Flags

	Original(s)	**Optimized(s)**	**Speedup**
w/o –fast	20.87	9.23	2.26x
w/ --fast	6.41	2.50	2.56x

Table 4.3 shows the performance improvement using my optimization. I gained a speedup of 2.26x on a small-sized problem (size=16). To show that my optimization works no matter which compilation flags I use, I applied the same optimization to the benchmark recompiled with the "--fast" option. The result of "--fast" version shows that with compiler optimization, my optimization still produces similar speedups.

4.2.2 CLOMP

CLOMP stands for C version of the Livermore OpenMP benchmark [71] and is used to simulate a typical scientific application to measure the overhead of different usage of OpenMP primitives. I selected variables with blame larger than 10% in Table 4.4.

CLOMP is a simple benchmark. After the initialization, the application starts the simulation by calling function *update_part* over and over again through the top loop function *do_parallel_version*, inside which, there is only one function: *parallel_cycle*. The function *parallel_cycle* calls four subprograms: *parallel_module1, parallel_module2, parallel_module3* and *parallel_module4.* The difference between these subprograms is simply the number of the parallel forall loops in each function. Besides this dominating calculation, there are multiple *re-initializations* and *calc_deposit* function calls, which only consume a small portion of the total run time. The roles of blamed variables are introduced below.

Table 4.4: Profiling Result for the Run of CLOMP

Name	Type	Blame	Context
partArray	[partDomain] Part	99.5%	main
->partArray[i]	Part	99.5%	main
->partArray[i].zoneArray[j]	Zone	99.0%	main
->partArray[i].zoneArray[j].value	real	99.0%	main
->partArray[i].residue	real	12.3%	main
remaining_deposit	real	11.8%	update_part

"->" symbol is used to represent field to its parent struct relation, with the parent struct variable listed above it in the table

partArray (99.5%): Variable *partArray* is the top-level data structure in the benchmark that holds everything of importance. It is created as a global variable, so that the final blame value from all the points wherever a piece of it gets written, that portion of blame will be aggregated to one single variable in the last step of my tool. The *partArray* is defined on a *partDomain*, which is based on the configurable constant *CLOMP_numParts* so that I can control the size of the domain in the execution command line. Besides other attributes in the *Part* data structure, I have an array of zones, which is created with the self-defined *Zone* type and runtime configurable array size: *CLOMP_zonesPerPart*.

partArray[i].zoneArray[j](99.0%): This variable is the element of the *zoneArray* in the global variable *partArray*. By following the hierarchical symbol "->", I'm able to find which field of the complex data structure is actually responsible for the most computation. I can see the value in *Zone* takes most credits so I have a basic idea that the whole program is trying to compute the field value for each zone in *partArray*.

partArray[i].residue (12.3%): The residue is a field of the *Part* class, it is calculated in the *update_part* function. It's another important field that needs to be updated frequently for each part besides all the zones.

remaining_deposit (11.8%): This is simply a local variable defined in function *update_part*. Since the function *update_part* is called frequently, so this variable is accessed a lot as well. It is used as a temporary variable for computing the value of *partArray[i].residue.*

Discussion and Optimization: Since the number of parts and the number of zones per part are determined on the command line, I can use a large 2D array to hold those values, like Johnson and Hollingsworth did [21]. Accessing elements in one big array is much faster than through nested structures. The performance improved by up to a factor of 2.13x. The details can be found in Table 4.5. I also list the result with compiler optimization enabled, in which case I get even better speedups with the same manual optimization.

Table 4.5: Results w/ or w/o "--fast" Flags

Flag	Problem Size	Original (s)	Optimized (s)	Speedup
w/o --fast	1024/64,000	4.02	2.18	1.84x
	65536/10	4.79	4.40	1.09x
	12/640,000	3.87	1.82	2.13x
	65536/6400	7.88	7.14	1.10x
w/ --fast	1024/64,000	3.72	1.44	2.59x
	65536/10	5.13	2.14	2.40x
	12/640,000	3.75	1.41	2.65x
	65536/6400	7.98	4.07	1.96x

numThreads=12, allocThreads=12, flopScale=1, timescale=100

4.2.3 LULESH

LULESH was first implemented by Lawrence Livermore National Lab (LLNL) and has since become a widely studied proxy application in DOE co-design efforts for exascale. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It has a collection of implementation versions based on most modern HPC programming models and languages, including Chapel. The problem size I chose is 15 elements per edge for the time limit considering my current sampling threshold.

LULESH Chapel source was designed to mirror the overall structure of the C++ LULESH but use Chapel constructs wherever they can to help make the code more concise and compact. It was implemented as a single locale, multi-threading program.

The function *LagrangeleapFrog* in main is responsible for 96% running time and most work is done underneath the forall parallel blocks inside its subroutines, therefore traditional code-centric profiler would only give limited insight while my profiler provides more insights about what and where to look for optimizations.

To compare my tool to prior approaches, I used Pprof from gperftools [16], an existing code-centric profiler that works for Chapel, to profile the benchmark. The profile output of the top ten functions is shown in Figure 4.10. The columns are:

1. Number of profiling samples in this function

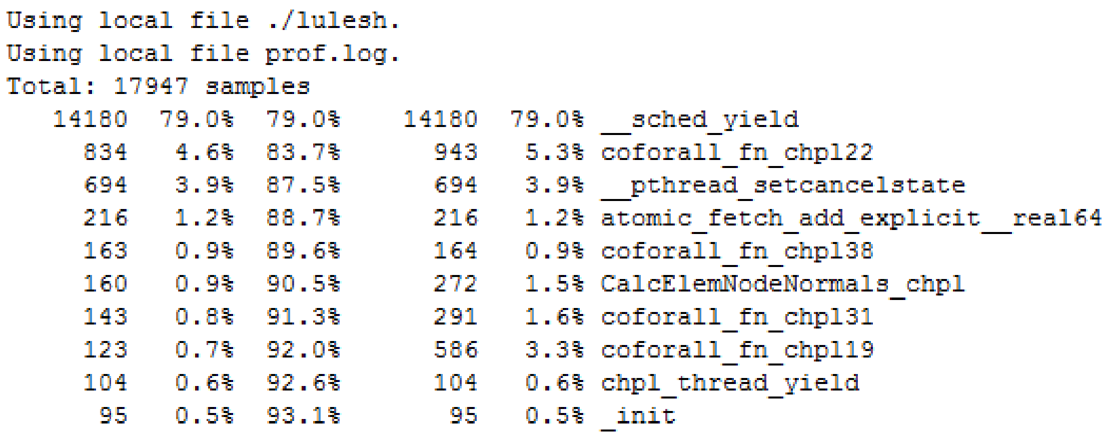
2. Percentage of profiling samples in this function

3. Cumulative percentage of samples

4. Number of samples in this function and its callees

5. Percentage of samples in this function and its callees

6. Function name



```
Using local file ./lulesh.
Using local file prof.log.
Total: 17947 samples
   14180  79.0%  79.0%    14180  79.0% __sched_yield
     834   4.6%  83.7%      943   5.3% coforall_fn_chpl22
     694   3.9%  87.5%      694   3.9% __pthread_setcancelstate
     216   1.2%  88.7%      216   1.2% atomic_fetch_add_explicit__real64
     163   0.9%  89.6%      164   0.9% coforall_fn_chpl38
     160   0.9%  90.5%      272   1.5% CalcElemNodeNormals_chpl
     143   0.8%  91.3%      291   1.6% coforall_fn_chpl31
     123   0.7%  92.0%      586   3.3% coforall_fn_chpl19
     104   0.6%  92.6%      104   0.6% chpl_thread_yield
      95   0.5%  93.1%       95   0.5% _init
```

Figure 4.10: Pprof output for LULESH

The output of Pprof is a bit confusing. First, it mixes functions from the Chapel runtime libraries and user code. The function that takes the largest portion of time on the list is *__sched_yield*, a system call that's referenced by Chapel threading layer.

It's used to force the running thread to relinquish the processor and move the thread to the end of the queue; time spent in this function is often due to load imbalance or lack of parallelism elsewhere in the program. The only function that can be recognized by users on the list is *CalcElemNodeNormals*, which only consumes 0.9% of the total time and reveals limited optimization opportunities. Second, the information isn't fine-grained enough to identify the specific performance bottlenecks in the code. In comparison, the blame profiling result in Table 4.6 provides richer variable-specific information, thus giving better insights into the user code optimizations.

The function *main* primarily serves as the highest level structure: initializing the test, starting the main loop that contains the core work, timing, and printing results. Almost all the samples fell within the function *LagrangeLeapFrog*. The "Context" column in Table 4.6 only lists the subroutines where the corresponding variables are defined. Note the sum of the blame for all variables is over 100%. As I briefly explained in Section 3.2, multiple variables could be blamed for a single sample. In this case, all the samples that were counted for *hourmodx's* will be assigned to *hx*, *shx*, and *hgfx* as well, thus these variables all get their own number of samples incremented. Therefore, the overall blame is larger than 100% for almost all programs as long as there is data dependency between variables. The roles of variables that are blamed most are introduced below.

hgfx (29.5%): LULESH is a symmetric 3-D simulation, thus I'll just use the x-axis variable to represent corresponding variables in all 3 dimensions later in the paper. Here, *hgfx* is an *8*real* tuple, defined in *CalcFBHourglassForceForElems*. Together with *shx, hx, hourgam,* and *hourmodx*, they compute the hourglass control force for each element.

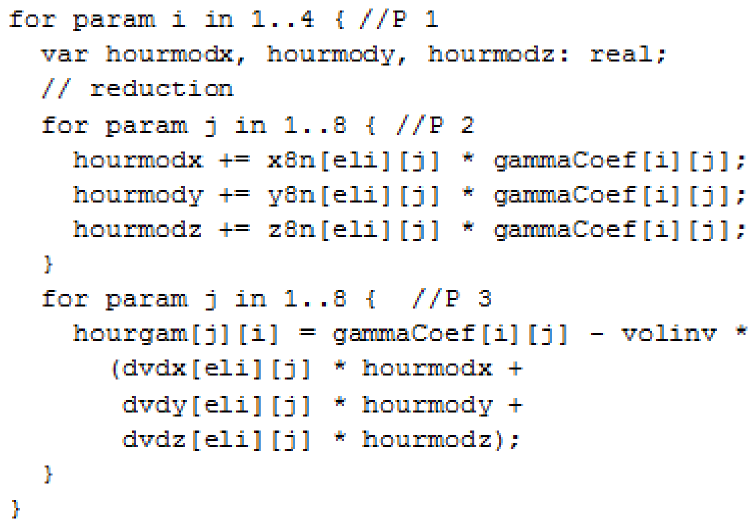
Table 4.6: Variables and Their Blame for the Run of LULESH

Name	Type	Blame	Context
hgfz	8*real	30.8%	CalcFBHourglassForceForElems
hgfx	8*real	29.5%	CalcFBHourglassForceForElems
hgfy	8*real	29.2%	CalcFBHourglassForceForElems
shz	real	27.9%	CalcElemFBHourglassForce
hz	4*real	27.6%	CalcElemFBHourglassForce
shx	real	26.9%	CalcElemFBHourglassForce
shy	real	26.6%	CalcElemFBHourglassForce
hx	4*real	26.6%	CalcElemFBHourglassForce
hy	4*real	26.6%	CalcElemFBHourglassForce
hourgam	8*(4*real)	25.0%	CalcFBHourglassForceForElems
determ	[Elems] real	15.7%	CalcVolumeForceForElems
b_x	8*real	9.7%	IntegrateStressForElems
b_z	8*real	9.7%	IntegrateStressForElems
b_y	8*real	8.7%	IntegrateStressForElems
dvdx(y/z)	[Elems] 8*real	8.3%	CalcHourglassControlForElems
hourmodx	real	5.8%	CalcFBHourglassForceForElems
hourmody	real	5.1%	CalcFBHourglassForceForElems
hourmodz	real	4.8%	CaclFBHourglassForceForElems

determ (15.7%): The variable determ is a higher level data abstraction defined in CalcVolumeForceForElems. It's a local array with a domain being dynamically allocated on the heap every time the function is called. The same situation happens to the variable dvdx, which is defined in *CalcHourglassControlForElems*. I will explore the potential optimization opportunities of these variables later in the discussion.

b_x (9.7%): The variable *b_x* is also a 8*real (floating point double) tuple declared in *IntegrateStressForElems* and passed into *CalcElemNodeNormals* as a reference. The value of *b_x* is assigned through a nested function inside *CalcElemNodeNormals*.

hourmodx (5.8%): The variable *hourmodx* is a local variable defined inside a nested for loop in function *CaclFBHourglassForceForElems*. It is used to calculate the value of *hgfx*. There is only one write to this variable in the code, but it is updated frequently due to the loop and it acts as an important role in transferring blame.



```
for param i in 1..4 { //P 1
  var hourmodx, hourmody, hourmodz: real;
  // reduction
  for param j in 1..8 { //P 2
    hourmodx += x8n[eli][j] * gammaCoef[i][j];
    hourmody += y8n[eli][j] * gammaCoef[i][j];
    hourmodz += z8n[eli][j] * gammaCoef[i][j];
  }
  for param j in 1..8 {  //P 3
    hourgam[j][i] = gammaCoef[i][j] - volinv *
       (dvdx[eli][j] * hourmodx +
        dvdy[eli][j] * hourmody +
        dvdz[eli][j] * hourmodz);
  }
}
```

Figure 4.11: Code Snapshot of LULESH hotspot

Discussion and Optimization: From Table 4.6, I discovered variables that hold the most blame in the program. After examining the code, I found that *hgfx(y/z), shx(y/z),* *hx(y/z), hourgam* and *hourmodx(y/z)* have direct data dependency between them: *hgfx(y/z)* depends on the value of *shx(y/z); shx(y/z)* depends on *hourgam* and *hx(y/z); hx(y/z)* depends on *hourgam*, which ultimately depends on the value of *hourmodx(y/z).* By further checking the code-centric data, it was discovered that over 21% of the total time came from the loop block in Figure 4.11. Therefore, optimizing this for loop is a good way to improve the overall performance.

Table 4.7: Results for Loop Unrolling Methods

Unrolling tag	Run time (s)	Speedup
Original	12.47	1.00x
0 params	12.04	1.04x
P 1	11.65	1.07x
P 2	12.95	0.96x
P 3	11.78	1.06x
P1+P2	12.59	0.99x
P1+P3	11.89	1.05x
P2+P3	12.60	0.99x
P1+U2	12.10	1.03x
P1+U3	12.33	1.01x
P1+U2+U3	12.75	0.98x

‘U x’ means I manually do the unrolling for that for loop in place x

The keyword “param” before the loop iterator in Chapel causes the compiler to optimize the code by unrolling the loop. However, sometimes it would be counterproductive since it enlarges the code size. Therefore, I did control tests by preserving or eliminating these keywords in each location (denoted as “P #”). I further combined it with manual unrolling to see if that would be beneficial as well. The experiment results are displayed in Table 4.7.

Among all the options, I can see that simply keeping "param" for the outermost loop (P 1) gives us the best performance for this block of code. By shortening the execution time of this loop block, I expect to decrease the blame of those variables used in the loop, e.g., *hourmodx* and *hourgam*. This worked well and the result is shown in Table 4.8.

The second optimization I made to the benchmark is from observing the variables *determ* and *dvdx*. At first, they seemed hard to optimize since the calculations of their values are deep inside the subroutines after their declarations. Without changing the algorithm, I can't simplify the computation. Fortunately, inspired by the optimization in Johnson's paper [21], I did Variable Globalization (VG). This optimization moves the declarations of several safe local variables to the global space so that they won't be dynamically allocated every time when the function is called. In this way, I saved about 19% execution time.

Another optimization I found through analyzing the profiling result is to work on variables *b_x, b_y, and b_z*. The values of *b_x*, *b_y*, and *b_z,* representing the "normal" from each face in the program, are computed in function *CalcElemNodeNormals* ("CENN" for short). Inside CENN, partial results are calculated through the nested function ElemFaceNormal and stored in temporary variables. Finally, the partial results from multiple *ElemFaceNormal* calls are added up through an addition operation on tuples. Since all temporary variables use tuple type, it involves tuple constructions and destructions, which are not cheap when they are nested deeply inside a big loop. I optimized this part by directly assigning intermediate results to the passed-in variables, thus avoiding redundant tuple constructions and destructions. This optimization denoted as "CENN" is able to reduce the execution time by 7%.

Table 4.8: Profiling Results Comparison between Different Optimizations

variable name	**Blame (%)**			
	Original	***P1***	***VG***	***CENN***
hgfx	29.5%	20.5%	31.3%	26.4 %
hgfy	29.2%	18.8%	31.3%	27.4%
hgfz	30.8%	19.8%	28.0%	27.1%
shx	26.9%	18.1%	27.7%	23.08%
shy	26.6%	17.0%	28.0%	24.8%
shz	27.9%	17.4%	27.0%	24.4%
hx	26.6%	17.0%	27.7%	23.1%
hy	26.6%	16.3%	27.0%	23.4%
hz	27.6%	17.0%	27.0%	23.8%
hourgam	25.0%	13.2%	25.7%	22.1%
hourmodx	5.8%	2.8%	7.3%	6.4%
hourmody	5.1%	3.8%	6.1%	6.7%
hourmodz	4.8%	2.4%	8.3%	6.0%
dvdx(y/z)	8.3%	7.3%	8.2%	7.0%
determ	15.7%	20.8%	14.8%	16.1%
b_x	9.7%	10.4%	9.0%	6.0%
b_y	8.7%	10.1%	9.0%	6.0%
b_z	9.7%	10.8%	9.3%	6.0%

Table 4.8 shows a profiling result comparison between each optimization I applied to the program. Instead of the default descending order, I group the variables that are affected by the same optimization. It gives us a clearer view of how a particular optimization would affect the profiling result of the relevant variables. The first optimization "P 1" reduces the computation time of that for loop, which directly affects variables *hourgam*, *hourmodx(y/z),* indirectly affects variables like *hgfx(y/z),* etc. Therefore, there is a decrease in the ratio of the above variables between the 3rd and 2nd columns in Table 4.8. The second optimization "VG" relates to *determ* and *dvdx* since it would reduce the number of times that these variables are declared and initialized. The total reduction in time brought by this optimization was achieved by hoisting many similar variables. The last optimization "CENN" focuses on simplifying the calculation of *b_x(y/z).* By comparing the 5th and 2nd column in Table 4.8 of these three variables, there is an obvious drop in their weight, which also meets my expectation.

Table 4.9 summarized the timing results of all versions of LULESH benchmark. The speedup column is the exclusive effect that the corresponding option achieves. The best case is the combination of all three optimizations (Combo). Overall, I achieved a factor of 1.4x speedup by modifying only 20~30 lines of source code.

I also list the "w/ --fast" column in Table 4.9, which shows the result for the compiler-optimized version. The overall speedup is bigger than that of "w/o --fast". The first and third manual optimization that I made obtain smaller speedups than before, that's probably because the "--fast" flag has already done some similar work for the original code, so my manual modifications gain less speedup.

Table 4.9: Optimization results w/ or w/o "--fast" flags

	w/o --fast		**w/ --fast**	
	Run Time(s)	***Speedup***	***Run Time(s)***	***Speedup***
Combo	9.02	1.38x	3.20	1.47x
VG	9.98	1.25x	3.39	1.39x
P 1	11.65	1.07x	4.54	1.04x
CENN	11.57	1.08x	4.59	1.02x
Original	12.47	1.00x	4.70	1.00x

4.3 Discussion and Summary

New parallel programming models provide newer abstractions for programmers. However, performance tools need to keep pace with these changes to present useful performance information in an intuitive way. A few established profilers, such as HPCToolkit-data-centric, lack the full capability to properly profile PGAS languages, where most variables in Chapel benchmarks are regarded as "unknown data". For example, in experiments, CLOMP has 96.88% performance statics falling in "unknown data" category and LULESH reports 95.1% in "unknown data", which cannot provide useful information to programmers. Compared to some existing work, my tool distinguishes itself in several aspects:

First of all, my tool is the first Chapel-specific performance measurement and analysis tool. Different from [11, 12, 13, 14], my approach chooses Chapel as the target language and provides valuable insights into the performance issue. My approach will support profiling Chapel in both the single-locale and multi-locale environments. For single-locale, my approach appropriately handles the task-based multi-threaded situation by merging the performance data from each thread to a single node while for multi-locale my approach will aggregate the complete performance data from each node. At the end of the profiling, users can directly identify the most time-consuming variables through a Graphic User Interface without any more digging in the performance data. It is a more straightforward data-centric view than locating problematic variables by source code and line number as [11, 17].

Secondly, my novel approach is able to map performance statistics back to variables in a user-level context. Unlike TAU [3], PPW [12], and pprof [16], which attribute performance data to functions, loops, basic blocks, mine attributes the performance loss to variables with real names in the source code. Besides, my tool only blames the variables from the user code for the performance loss because all the other variables are utilized to compute the ultimate values of user variables, and it is better for users to concentrate on optimizing their own code.

Lastly, my approach supports profiling a full user code calling context. A call path is a chain of functions with calling relationships. Associating performance losses with call paths provides unique performance insight into program executions. For example, consider a threaded program that employs task-based parallelism. If threads spend a lot of time spinning in synchronization routines for accesses to shared resources, a flat profiling without the call path information cannot tell which task caused the spinning. Besides, without call path information, programmers cannot distinguish variables with the same name but different scopes. Unlike pprof [16] which lacks the ability to gain the calling context and HPCToolKit [17] which shows the limited capability in analyzing multi-threaded Chapel programs, mine has the full support of call path profiling on the Chapel code, mine maps the performance data to the original user code elements from the bottom frame in the stack and propagates it all the way up to the top main function. This support helps users quickly identify the performance bottlenecks in programs.

In this chapter, I designed and developed a data-centric profiling tool *ChplBlamer* that supports PGAS languages, using Chapel as an exemplar. I introduced the state-of-art HPC/PGAS profiling tools and compared them with my tool. I demonstrated the functionality and usability of ChplBlamer on three well-known benchmarks. With the guidance supplied by ChplBlamer, I significantly improved the performance by factors of 1.4x for LULESH, 2.3x for MiniMD, and up to 2.1x for CLOMP, with minimal changes to the source code. I also concluded that domain remapping and zippered iterations are expensive to use. The overhead of ChplBlamer is discussed in a multi-locale context in Section 5.4 Discussion and Summary.

Chapter 5

Data-centric Profiling for Multi-locale Chapel Programs

In the prior chapter, I proposed a data-centric performance measurement tool *ChplBlamer* for single-locale Chapel programs. In this chapter, I extended the prior work by providing a more functional data-centric and code-centric combined Chapel profiler, *ChplBlamer-ML*, to pinpoint performance losses due to data distribution and remote data accesses in a multi-locale environment. ChplBlamer-ML improves the prior work in several aspects:

1. It supports more generic Chapel code, including multi-locale Chapel and abstractions that support both asynchronous and remote tasks.
2. It provides additional tool capabilities: such as inter-node load imbalance examination to help users investigate performance issues more efficiently.
3. The instrumentation to the Chapel runtime library is optimized and the runtime overhead is significantly reduced from 3.5x to 14% compared to the previous work in Chapter 4.

To demonstrate the utility of ChplBlamer-ML, I studied three multi-locale Chapel benchmarks. For each benchmark, ChplBlamer-ML found the causes of the performance losses. With the optimization guidance provided by ChplBlamer-ML, I significantly improved the performance by up to 4x with little code modification. This chapter is adapted from a paper that has been presented at the International Conference on Supercomputing (ICS'18) [69].

5.1 Challenges and Solutions

Conducting data-centric profiling on multi-locale Chapel is far more challenging than the single-locale. Chapel, as a PGAS language, includes a runtime middleware to handle inter-node communication and data distribution. A single-line distributed vector addition statement "*C* = *A*+*B*" in the Chapel source will be compiled to thousands of instructions that involve calls to the Chapel runtime library to handle data distribution and inter-node communication. In order to accurately attribute blame to source code variables, I need to address several challenges.

```
use CyclicDist;
const myD = {1..N} dmapped Cyclic(startIdx=1);
var myVec: [myD] real;
forall a in myVec
    a = ..;
begin {
    localCompute(myVec);
}
on Locales[1] do
    remoteCompute(myVec);
```

Figure 5.1: Sample multi-locale Chapel code

Figure 5.1 shows a simple example using Chapel multi-locale syntax that the single-locale ChplBlamer cannot handle. The variable myVec is a distributed array defined on a cyclic distributed domain *myD*; forall loop tries to leverage all threads on all locales to initialize myVec; the begin block creates a new task on the current locale and the parent thread continues without waiting for the block to finish; the on clause launches a remote task on Locale 1 in an asynchronous style as well.

5.1.1 1st Challenge and Solution

Challenge

For a variable that is distributed among multiple locales and requires remote access, there are hundreds of aliases and temporary variables representing a block of the data of the variable in the computation. How to identify those data blocks and finally aggregate their individual blame share to the original variable becomes a problem. Moreover, Chapel creates a unique private identifier (PrivID) for each distributed variable (e.g., *myVec* in Figure 5.1) for future references. Therefore, when it's accessed and passed through functions, the original logic to handle variables with a type of array or structure in Chapter 4 will fail since now reference of those distributed variables are simply integers and will not be regarded as "exit variables".

Solution

Since distributed variables are referenced via unique PrivIDs (private IDs), I established a mapping between the vertex that represents the PrivID and the one that represents the original source variable when I build graphs in the static analysis. Every time a distributed variable is accessed, I can locate the corresponding PrivID node following the dataflow path in the IR. And with the link between the PrivID and the original variable, I can identify exactly which variable is being accessed. However, the biggest question is how do I figure out which nodes represent PrivIDs and link them to the corresponding variables (I use “object” to refer to the original source variable and all its compiler-generated aliases and tempory copies) since PrivIDs are no different than constant integers once created at the IR level. I solved that by tracking two critical functions from the Chapel runtime library: *chpl_getPrivatizedCopy,* and *chpl_getPrivatizedClass.* I found that these functions resolve PID from the corresponding object. Therefore, I can identify PrivIDs and their associated objects. Once I determine a PrivID and the object, I find all aliases[3] of the PrivID as well as the aliases of the object backward and forward in this function. Now wherever an object alias is accessed, I can trace it back to the original source variable.

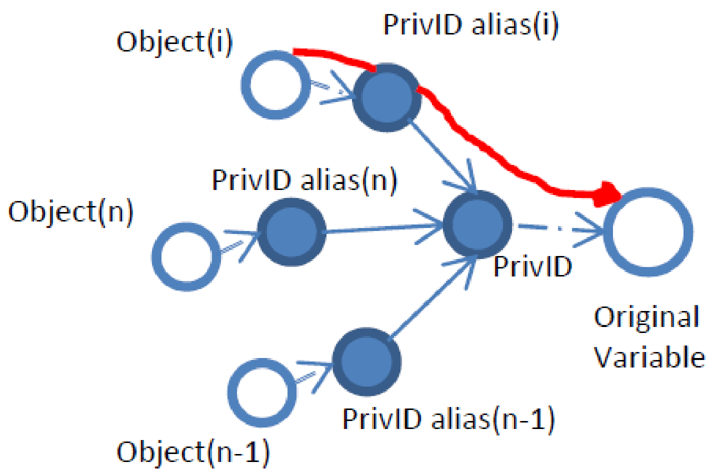


Figure 5.2: The process of locating the original variable

[3] “Alias” is not technically correct for PIDs since they are integers so “aliases” are just variable copies, but I use it for its literal meaning and the way I find them is similar to alias analysis.

Figure 5.2 illustrates the idea of this process. At any access point to Object(i), I can follow the red path (the top arrowed curve) to track down the original distributed variable that this Object(i) was derived from.

Since an integer is not recognized as a PrivID until it is found to be a parameter in one of these two functions (*chpl_getPrivatizedCopy,* and *chpl_getPrivatizedClass*), then how does the upper-level function know whether the integer variable will be used as a PrivID in its callees? The answer is that it is not known in the intra-procedural static analysis step. To solve this problem, ChplBlamer-ML conservatively treats all integer parameters as potential PrivIDs and store their aliases; then in the postmortem analysis, it can get all PrivIDs back frame by frame along the call path if any integer parameters are determined as PrivIDs in a certain frame.

5.1.2 2nd Challenge and Solution

Challenge

At the IR level, multi-locale Chapel programs call functions from the runtime library and standard modules to retrieve the locality information for remote data access, which involves implicit dataflow information. For example, communication calls such as *chpl_gen_comm_get* and *chpl_gen_comm_put*, implicitly generate data dependency between the remote data and the local copy. I need to recover this hidden dataflow information to propagate blame properly. Moreover, explicit operations on distributed variables or within a parallel region at the source level will be wrapped into generated functions and implicitly invoked within Chapel runtime functions using function pointers. The “transfer function” mechanism from Chapter 4 is not able to handle this case, thus it will fail the inter-procedural blame propagation.

Solution

In regard to the second challenge, I observed that non-user functions containing important dataflow information fall into two categories: module functions and runtime functions.

For functions from Chapel standard modules, since their definitions are also in the IR, I implemented a simplified blame analysis to figure out the blamed parameters that are responsible for any call to that function. For a few functions from the Chapel runtime library, I manually figured out the blamed parameter indices since their function bodies are not included in the IR. In this way, I keep propagating blame to the callers via blamed parameters for calls to those functions.

To solve the problem of wrapper functions and function pointers, I conduct additional analysis on the program IR. First, I record the table of function pointers (with symbol names) that point to all generated wrapper functions in this program. Second, I extract parameters of three critical functions from the Chapel runtime library: *chpl_executeOn, chpl_taskListAddBegin,* and *chpl_taskListAddCoStmt.* The most important parameter of these functions is a constant integer that equals to the index of the corresponding function pointer in the table I previously recorded. In this way, I retrieve the exact wrapper function that will be called. Finally, I also need to match the parameters of those runtime functions to the real parameters that will be fed to the wrapper function. This is also tricky since the parameters are decomposed and reconstructed and it’s not a 1-1 correspondence. With all these efforts, I am able to mimic the explicit operations at the source level with the program IR and recover the dataflow information.

5.1.3 3rd Challenge and Solution

Challenge

A multi-locale Chapel program does not launch the same execution from the main function on all locales simultaneously; rather only the master locale launches the execution from the very beginning and all other locales launch their jobs as needed during the entire course of execution (essentially a fork-join model). Therefore, when I walk the stack of a thread on a worker locale, it is very likely that the top stack frame (suppose the stack grows downwards) is not the "main" function but somewhere that particular task starts from. Missing the complete calling context precludes propagating blame along the call path appropriately. Moreover, Chapel's asynchronous tasking feature aggravates this problem, since now a task can be created on Locale 1 at beginning of the execution and later remotely executed by Locale 2, while Locale 0 continues right after it launches that task.

Solution

To get the complete user-level calling context for each sample, I instrumented both the tasking and communication layers of the Chapel runtime using callback functions. In the tasking layer, I insert a callback in the function *add_to_task_pool*, so that every time a new task is added to the local task pool, I unwind the stack of the current thread and keep the unique function ID (referred as "fID") for that task. The stacktrace shows the call path before a local task is executed. In the communication layer, I insert callbacks in function *chpl_comm_execute_on,* *chpl_comm_execute_on_nb,* and *chpl_comm_execute_on_fast*, so that every time a remote task is created and sent to another locale, I unwind the stack and keep the unique fID, as well as the locale IDs of the sender and receiver (referred as “sID” and “rID”) for the task. The stacktrace shows the call path before a remote task is launched.

During the instrumentation and sampling of the program execution, I also track the frame name as I unwind the stack. Once I find the top frame is one of the fork wrapper functions of a remote task defined in the Chapel runtime (e.g. “*fork_wrapper*”), I read the fID, sID, and rID from the function parameters, or simply fID if the top frame indicates a local task.

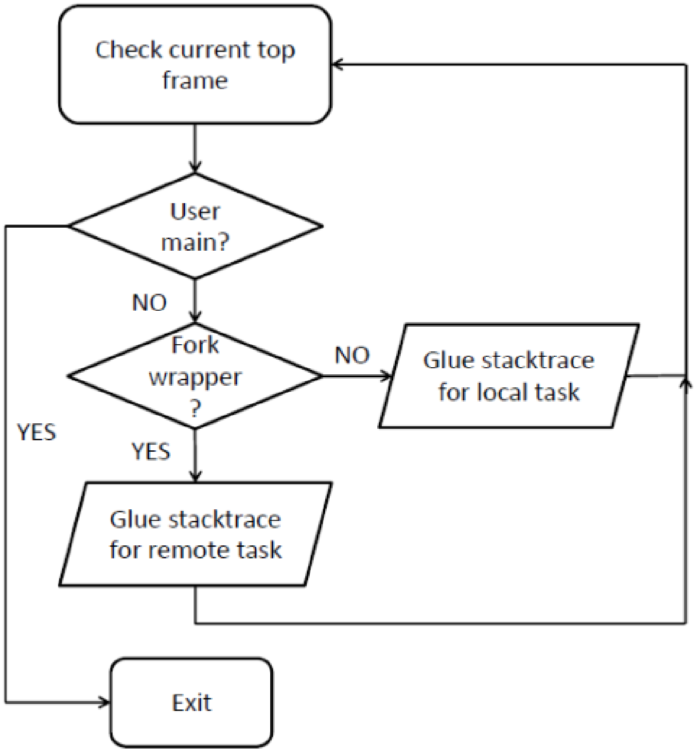

Check current top frame
User main?
NO
Fork wrapper ?
NO
Glue stacktrace for local task
YES
YES
Glue stacktrace for remote task
Exit

Figure 5.3: Stacktrace concatenation flowchart

Finally, during the process of stacktrace concatenation, I use the above keys (fID, sID, rID) to find the call path before a certain task. The parent task is found iteratively until I see the user main function in the stack. In the meantime, I also remove frames that are not resolved to user functions so that the ultimate sample stacktraces are in a full user-level context. Figure 5.3 shows the flowchart of this concatenation process; I use hash maps to retrieve stacktraces efficiently.

The way I reconstruct the calling context for samples brings two benefits:

First, it essentially solves the asynchronous and remote tasking problem. Since every piece of sequential work is a task in Chapel, with the keys (fID, sID, rID), I know what it does and where it was launched. If Locale 1 launched a task on Locale 2 at Time 1 and Locale 2 later executed the task at Time 2 where a sample is triggered, I can reproduce the calling context without interfering with the program execution.

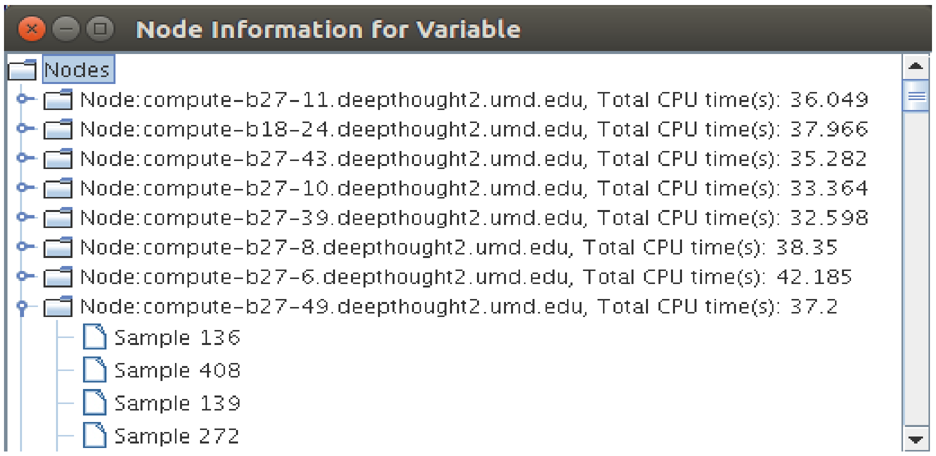
Second, it significantly reduces the runtime overhead. While unwinding the stack in the callbacks, I also keep those keys in a set that is shared by all threads in the same locale. Therefore, the next time a task with the same key comes in, I need not unwind the stack again since the same information has been recorded. The prior work in Chapter 4 unwinds the stack every time a task list is executed (A task list is created for each "*forall*" or "*coforall*" parallel loop). That approach could not handle asynchronous parallelism and would incur unacceptable overhead in certain circumstances. Table 5.1 shows that this approach can reduce the average overhead from 3.5x to 14% for three single-locale Chapel benchmarks. The overhead is measured using the formula $\left(\frac{profiled\ execution}{clean\ execution} - 1\right) \times 100\%$.

Table 5.1: Tool overhead comparison on single-locale

Benchmark	MiniMD	CLOMP	LULESH
Prior overhead	4.2x	1.4x	4.9x
Current overhead	5%	9%	27%

5.2 Inter-node Load imbalance Examination

Load-imbalance is a critical performance problem in High Performance Computing, researchers use static or dynamic load balancing techniques to evenly distribute data and computations across all processors/nodes in order to optimize the run time and system I/O. Most traditional approaches present this information based on the computation cost on each processor/node, instead, I include a feature of data-centric inter-node load imbalance examination in ChplBlamer-ML.



```
Nodes
  Node:compute-b27-11.deepthought2.umd.edu, Total CPU time(s): 36.049
  Node:compute-b18-24.deepthought2.umd.edu, Total CPU time(s): 37.966
  Node:compute-b27-43.deepthought2.umd.edu, Total CPU time(s): 35.282
  Node:compute-b27-10.deepthought2.umd.edu, Total CPU time(s): 33.364
  Node:compute-b27-39.deepthought2.umd.edu, Total CPU time(s): 32.598
  Node:compute-b27-8.deepthought2.umd.edu, Total CPU time(s): 38.35
  Node:compute-b27-6.deepthought2.umd.edu, Total CPU time(s): 42.185
  Node:compute-b27-49.deepthought2.umd.edu, Total CPU time(s): 37.2
    Sample 136
    Sample 408
    Sample 139
    Sample 272
```

Figure 5.4: Node information for *Ab* of HPL on 32 locales

In ChplBlamer, there are three ways to view the data: a flat data-centric view, a traditional code-centric view, and a hybrid view using the concept of "blame point" where you can basically stop blame propagation at a certain point in the call path and reflect the performance statistics at that scope. Besides the above three different views, I also include a view of workload information. Clicking on a particular variable in the data-centric view will pop up a window showing the total CPU seconds for that variable on each locale. Note that the total CPU time is the aggregation of all cores involved in that locale. You can also drill down from each node to display the specific samples that contribute to the time, which can be used with data profiles to verify the result. The different time on each locale shows the load imbalance situation in terms of this variable. For a distributed array, if certain locale consumes significantly more or less time than others, it means significantly greater or fewer array elements are distributed on that locale than others. Thus the user should tune the block size of the distribution based on the array size for that variable. Figure 5.4 shows an example of variable *Ab* in HPL on 32 locales.

5.3 Case Studies

I evaluated ChplBlamer-ML on a local InfiniBand-based cluster Deepthought2. Deepthought2 consists of 484 nodes with dual socket (20 cores per node) Ivy Bridge 2.8 GHz processors. I used from 2 to 32 nodes in each case in this evaluation. I use CPU clock cycles as the sampling event and the sampling period is 1,073,807,359, properly chosen to balance the overhead and precision. Taking HPL as an example, the time overhead of ChplBlamer-ML ranges from 13% to 25% with 2 to 32 nodes, respectively; the space overhead is 90KB for problem size 500 with 32 nodes.

In this section, I studied three well-known multi-locale Chapel benchmarks. All programs were built with Chapel 1.15 and the `--fast` (equivalent to "-O3" in GNU compilers) optimization. The description of each benchmark is as follows:

- HPL [72], the High Performance Linpack benchmark solves a uniformly random system of linear equations and reports time and floating-point execution rate using a standard formula for operation count.
- ISx [73] is the scalable Integer Sorting application. The Chapel version is fully Single-Program-Multiple-Data (SPMD), creating a task per locale and a task per physical core on each locale.
- LULESH [74] approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It has many implementations for most HPC programming models and languages, including Chapel.

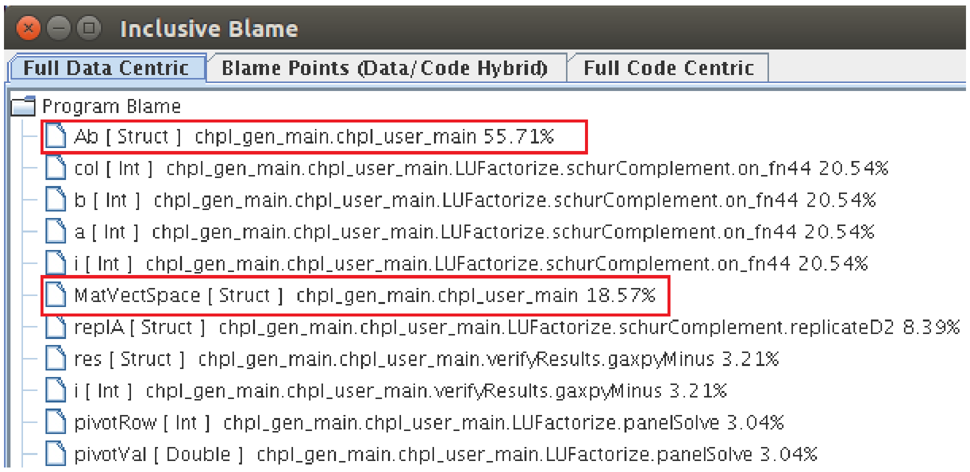
I tried different Chapel configurations to get the fastest run time and used that as my performance baseline. There are two environment variables I tuned for performance: CHPL_TASKS ("fifo" or "qthreads" implementation as the Chapel tasking layer) and CHPL_RT_NUM_THREADS_PER_LOCALE (up to how many threads can be created per node). Based on my experimental results, I've concluded that fifo is better for HPL and LULESH while qthreads is better for ISx. Those values are what I used to measure the performance reported in the rest of this section. As for CHPL_RT_NUM_THREADS_PER_LOCALE[4], it only affects the fifo version and

[4] Ideally, I would not need to set this environment variable since I want fifo to spawn as many threads as it needs and qthreads internally creates fixed number of user-level threads. However, the performance difference makes it a worthwhile effort.

the details will be discussed case by case. I focused on the strong scaling study for HPL and LULESH (fixed problem size), and the weak scaling study for ISx (fixed problem size per task).

To clarify, while profiling, I only used "fifo" as the Chapel tasking layer and compiled all programs with "`--llvm --no-checks`" (using the llvm frontend with no boundary checks). I did not use "`—fast`" in profiling since my intraprocedural analysis heavily depends on the generated LLVM bitcode of the Chapel program and "`—fast`" option loses too much debug information that I need to associate the IR-level objects (temporary variables and registers) with the source-level variables. I also discussed the optimization guided by my tool for each benchmark in detail. Since all execution time was measured for binaries built with "`—fast`", I demonstrated that the optimization found by profiling non-optimized versions still helps in tuning the optimized versions.

5.3.1 HPL



```
Program Blame
  Ab [ Struct ]  chpl_gen_main.chpl_user_main 55.71%
  col [ Int ]  chpl_gen_main.chpl_user_main.LUFactorize.schurComplement.on_fn44 20.54%
  b [ Int ]  chpl_gen_main.chpl_user_main.LUFactorize.schurComplement.on_fn44 20.54%
  a [ Int ]  chpl_gen_main.chpl_user_main.LUFactorize.schurComplement.on_fn44 20.54%
  i [ Int ]  chpl_gen_main.chpl_user_main.LUFactorize.schurComplement.on_fn44 20.54%
  MatVectSpace [ Struct ]  chpl_gen_main.chpl_user_main 18.57%
  replA [ Struct ]  chpl_gen_main.chpl_user_main.LUFactorize.schurComplement.replicateD2 8.39%
  res [ Struct ]  chpl_gen_main.chpl_user_main.verifyResults.gaxpyMinus 3.21%
  i [ Int ]  chpl_gen_main.chpl_user_main.verifyResults.gaxpyMinus 3.21%
  pivotRow [ Int ]  chpl_gen_main.chpl_user_main.LUFactorize.panelSolve 3.04%
  pivotVal [ Double ]  chpl_gen_main.chpl_user_main.LUFactorize.panelSolve 3.04%
```

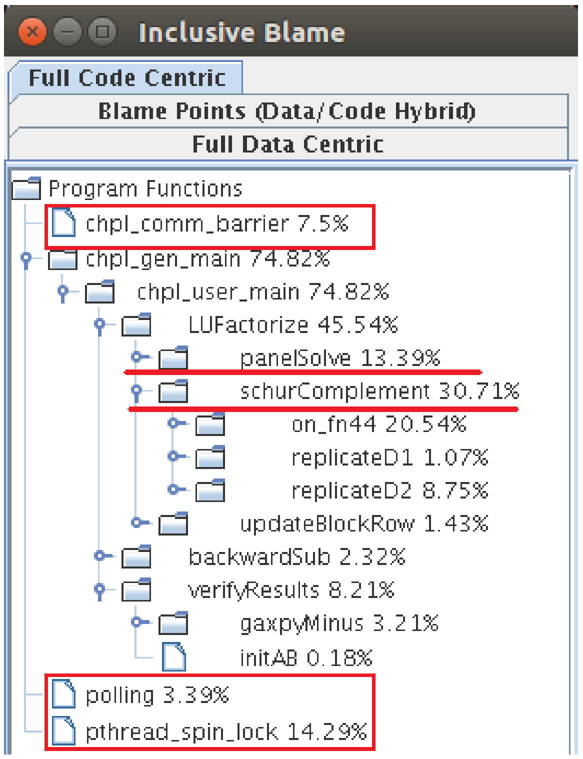
Figure 5.5: Data-centric blame for HPL on 2 locales. The red rectangles enclose interesting variables with their name, type, full calling context, and blame percentage.

Figure 5.5 shows the blame for each variable as well as the context information in the source, including the name, type, and full call path to the point where the associated variable is declared. Note that there are some functions ending with numbers (e.g. "*on_fn44*" and "*coforall_fn14*") in the calling context; they are auto-generated by the compiler to handle the parallel constructs (e.g. "forall") in the program. Users can simply ignore those functions or treat them as code blocks of the nearest caller function when analyzing the result. First, I describe the purpose of the most blamed variables and functions (the corresponding blame is shown in the parentheses). Then I explain how I interpret the blame results in both code-centric and data-centric ways to discover the performance bottlenecks and scalability issues. Finally, I use this information to optimize the program.

Ab (55.7%): A 2D array allocated on distributed memory with the distributed domain *MatVectSpace*. It holds the value of a matrix and a vector and is responsible for the calculation of most linear equations in the program.

MatVectSpace (18.6%): A 2D domain that represents the *n X n* matrix adjacent to the column vector b. It uses the BlockCyclic distribution to distribute *Ab* to all nodes, leveraging the spatial locality in the blocked-computation. The default block size is 8, which can be tuned for performance in executions.

Figure 5.6 shows the inclusive code-centric result in the user-level calling context. Several functions are described below:



```
Program Functions
    chpl_comm_barrier 7.5%
    chpl_gen_main 74.82%
        chpl_user_main 74.82%
            LUFactorize 45.54%
                panelSolve 13.39%
                schurComplement 30.71%
                    on_fn44 20.54%
                    replicateD1 1.07%
                    replicateD2 8.75%
                updateBlockRow 1.43%
            backwardSub 2.32%
            verifyResults 8.21%
                gaxpyMinus 3.21%
                initAB 0.18%
    polling 3.39%
    pthread_spin_lock 14.29%
```

Figure 5.6: Code-centric blame for HPL on 2 locales. The red rectangles enclose runtime functions that indicate special performance issues and the underscored are user functions that are to be optimized

LUFactorize (45.5%): The function that consumes the most CPU cycles. It computes the blocked LU factorization with pivoting for matrix augmented with a vector of right-hand-side values. The computation is on a block by block basis.

schurComplement (30.7%): Computes the distributed matrix-multiplication. Each locale with a block of data updates itself by multiplying the neighboring left block to the upper block.

panelSolve (13.4%): Does unblocked-LU decomposition in the specified panel and updates the pivot vector for *Ab*, difficult to optimize for its unblocked data accesses.

pthread_spin_lock (14.3%): A low-level synchronization function used by the Chapel runtime for concurrent operations on the shared memory. The percentage shows the overhead in the Chapel tasking layer.

chpl_comm_barrier (7.5%): A Chapel runtime library function used to implement implicit barriers in Chapel. The time spent on *chpl_comm_barrier* indicates load imbalance in the program.

polling (3.4%): *A* specific task created on each locale to check for Active Messages (both requests and replies) which are inbound to that locale. Time spent on *polling* shows the communication overhead in the program.

Table 5.2: Major data-centric and code-centric blame percentages for HPL on different number of locales

#Locales	2	4	8	16	32(200)
Variable Name	**Data-centric Blame**				
Ab	55.7%	45.8%	36.4%	32.7%	23.6%
MatVectSpace	18.6%	27.8%	37.1%	38.7%	51.0%
Function Name	**Code-centric Blame**				
schurComplement	30.7%	22.2%	17.3%	14.0%	8.7%
panelSolve	13.4%	15.7%	15.0%	15.8%	12.0%
polling	3.4%	11.5%	7.5%	11.7%	9.6%
chpl_comm_barrier	7.5%	10.5%	11.4%	11.5%	11.0%
pthread_spin_lock	14.3%	3.8%	7.1%	5.0%	4.5%

Discussion and Optimization: Variable *Ab* and *MatVectSpace,* and function *schurComplement* and *panelSolve* are the most telling data about what is going on in the program. Table 5.2 summarizes the profiling data of these program elements after executing on from 2 to 32 locales. CHPL_RT_NUM_THREADS_PER_LOCALE is set to be the number in the parentheses associated with the number of locales entry and unset otherwise, for the best performance on that number of locales. The Same denotation is used in Table 5.3 and Table 5.9. The relative weight change between variable *MatVectSpace* and *Ab* as more locales are used shows that the initialization cost of a distributed domain is very high. The increasing proportion of *polling* and *chpl_comm_barrier* and the decrease of *pthread_spin_lock* show that the inter-locale overhead becomes dominant over the intra-locale as more locales are involved.

I explored several ways to optimize the program. First, in *schurComplement*, I can enable the 'local' clause right inside the 'forall' loop to assert the local matrix multiplication and remove redundant communication calls. Code segments guaranteed to access only local data may be enclosed within a 'local' statement. The keyword restrains the compiler from generating wide pointers[5] to access some distributed variables. This change reduced the total execution time by 3.1%. The speedups are summarized in Table 5.3.

The optimization to function *panelSolve* is trickier as most of its computation needs to access remote data. I leveraged the *ReplicatedDist* module to create a local copy of a row in *Ab* on each locale to avoid frequent remote accesses within the loop. The modified code is shown in Figure 5.7: Loop optimization using replication for *panelSolve*.

[5] Chapel uses "wide pointers" to point to non-local data.

```
const DLow: int = panel.dim(2).low;
const DHigh: int = panel.dim(2).high;
const DRep: domain(2)
  dmapped ReplicatedDist()={1..1,Dlow..Dhigh};
var AbRep: [DRep] elemType;
for k in panel.dim(2) {
  ...
  AbRep = Ab[k..k, DLow..DHigh];
  forall (i,j) in panel[k+1.., k+1..] {
     local {
       Ab[i,j] -= Ab[I,k] * AbRep[1,j];
...
```

Figure 5.7: Loop optimization using replication for *panelSolve*

However, this optimization only got a speedup of 1.1x in the test on 4 locales. It is because the overhead caused by updating the local copy (*AbRep*) in every iteration cannot be compensated by the performance benefit it brings when the block size is not well tuned with different problem sizes and the number of locales, in which case the locality is not fully leveraged.

The HPL benchmark has been studied and highly optimized for years so it is hard to further improve the performance without a major change to the fundamental algorithm. However, I still gained some insights and speedups with the help of ChplBlamer-ML.

Table 5.3: Speedups of the localization optimization for HPL

#Locales	2	4	8	16	32(200)
original (s)	13.67	18.26	17.70	19.30	30.48
localization (s)	13.49	18.01	17.52	17.98	29.08
speedup	1.01x	1.01x	1.01x	1.07x	1.05x

5.3.2 ISx

The Chapel port of ISx is a newly developed benchmark based on the OpenSHMEM implementation [73]. Table 5.4 shows the most blamed objects in both data-centric and code-centric profiling of ISx execution on 2 or 8 locales. The difference between 2-loc and 8-loc tells us which program objects (variable or function) are more affected by the communication and task synchronization cost.

myBucketedKeys (41.1%): It is a local variable in *bucketSort*, an array of configurable number (default 5,592,400) of keys. Every task allocates each one of this variable and populates the value in function *bucketizeLocalKeys*.

barrier (10.3%): An instance of the Barrier class in Chapel, it is used for task synchronization in the program. Since the current implementation of the Barrier standard module is not expected to perform well at scale, this variable becomes a major performance bottleneck when the number of tasks increases.

sendOffsets (27.3%) and *bucketOffsets(26.9%)*: *sendOffsets* is an array of integers allocated on the master locale and *bucketOffsets* is the local copy of *sendOffsets* for each task and is used to compute *myBucketedKeys* in function *bucketizeLocalKeys*.

Function *bucketSort* is the core function to implement the sorting algorithm. It consists of 5 steps (each step is implemented by a sub-function): *makeInput*, *countLocalBucketSizes*, *bucketizeLocalKeys*, *exchangeKeys*, and *countLocalKeys*.

Table 5.4: Data-centric and code-centric results of the most blamed variables and functions in ISx on 2 or 8 locales

Data-centric	**type**	**context**	**2-loc**	**8-loc**
myBucketedKeys	Struct	bucketSort	41.1%	22.9%
myKeys	Struct	bucketSort	36.9%	20.9%
sendOffsets	Struct	bucketSort	27.3%	15.4%
bucketOffsets	Struct	bucketizeLocalKeys	26.9%	15.2%
barrier	Struct	chpl_user_main	10.3%	20.8%
Code-centric		**context**	**2-loc**	**8-loc**
bucketSort		chpl_user_main	80.9%	64.2%
bucketizeLocalKeys		bucketSort	40.2%	22.3%
countLocalKeys		bucketSort	11.4%	6.4%
pthread_spin_lock		chpl_gen_main	16.7%	29.3%
chpl_comm_barrier		Chapel runtime	0	3.46%

Discussion and Optimization: With the help of ChplBlamer-ML, I easily identified the most "valuable" variables (such as *barrier*, *myBucketedKeys*, *and myKeys*) and functions (such as *bucketizeLocalKeys*, *countLocalKeys*) in terms of the performance. Optimizing the Barrier module would be the best thing to do since it affects the scalability largely and that is indeed part of the future work of the Chapel team. Here, I optimized the code using localization (localizing certain computation using the ‘local’ clause), similar to what I did for HPL. By tracking the most blamed variable *myBucketedKeys*, I found an opportunity for localization inside *bucketizeLocalKeys*. I enclosed all computation of that function in a ‘local’ statement. The same modification was done for the scan operation on variable *sendOffsets*. After the optimization, the blame percentage of those variables and functions also decreased correspondingly, as shown in Table 5.5. Table 5.6 lists the speedups on a different number of locales with the simple modification.

Table 5.5: Blame change before and after the optimization for related variables and function (bottom row)

Name	original	localization
myBucketedKeys	41.11%	17.78%
sendOffsets	27.28%	6.02%
bucketOffsets	26.85%	5.46%
bucketizeLocalKeys	40.24%	24.54%

Table 5.6: Speedups of the localization optimization on a different number of locales for ISx

#Locales	2	4	8	16	32
original (s)	0.53	0.66	0.89	1.30	2.21
localization (s)	0.42	0.59	0.85	1.19	1.99
speedup	1.26x	1.12x	1.05x	1.09x	1.11x

5.3.3 LULESH

The input problem size for all tests is 15 elements per edge. I carefully tuned CHPL_RT_NUM_THREADS_PER_LOCALE for tests on a different number of locales. The best values are indicated in Table 5.9. After manually tuning the parameter, I found that the best value is always 4 when you allocate more than 8 locales. This experience shows the poor intra-node scalability of the program because the thread-level parallelism is not fully utilized. Table 5.7 shows the data-centric blame result of LULESH.

Table 5.7: Data-centric blame for LULESH

Variable	Type	Blame	Context
Elems	Struct	74.3%	chpl_gen_main
elemToNode	Struct	60.4%	chpl_gen_main
xd/yd/zd	Struct	48.0%	chpl_gen_main
x/y/z	Struct	37.0%	chpl_gen_main
fx/fy/fz	Struct	35.6%	chpl_gen_main
dvdx/dvdy/dvdz	Struct	33.4%	CalcHourglassControlForElems
x8n/y8n/z8n	Struct	33.3%	CalcHourglassControlForElems
elemMass	Struct	29.5%	chpl_gen_main
hgfx/hgfy/hgfz	Array	26.7%	CalcFBHourglassForceForElems
shx/shy/shz	Double	26.7%	CalcElemFBHourglassForce
hx/hy/hz	Array	26.6%	CalcElemFBHourglassForce
dxx/dyy/dzz	Struct	12.2%	CalcLagrangeElements

Elems (74.3%): The essential domain that the construction of most distributed variables use. It uses block distribution so the block of elements to compute are evenly distributed among all compute nodes.

elemToNode (60.4%): A large distributed array that supports the complement mapping between each element and its surrounding nodes (*Node* and *Element* are the two most important units for computation in the program; each *Element* has 8 neighboring *Nodes* by default). Therefore, *elemToNode* is accessed frequently during the entire course of execution by all nodes to retrieve the index information.

x/y/z (37.0%), xd/yd/zd (48.0%), and *fx/fy/fz (35.6%)*: These are attributes of each *Node*, representing the coordinates, velocities, and forces in each dimension. They are calculated and updated frequently during the simulation process.

Table 5.8: Code-centric blame for LULESH

Function (caller->callee)	**Blame**
chpl_gen_main->chpl_user_main	94.3%
chpl_user_main->CalcForceForNode	47.9%
CalcForceForNode->CalcVolumeForceForElems	46.6%
CalcVolumeForceForElems->CalcHourglassControlForElems	34.6%
CalcHourglassControlForElems->CalcFBHourglassForceForElems	27.1%
chpl_user_main->ApplyMaterialPropertiesForElems	15.5%
chpl_user_main->CalcLagrangeElements	12.9%
chpl_user_main->CalcQForElems	12.6%
CalcLagrangeElements->CalcKinematicsForElems	11.4%
CalcQForElems->CalcMonotonicQGradientsForElems	9.4%
CalcVolumeForceForElems->IntegrateStressForElems	8.7%

Table 5.8 lists the most blamed user functions. I show the callsite of each function (caller->callee). The blame percentage shown above is the inclusive result.

Discussion and Optimization: I found some optimizations that speed up the program by a factor of 1.4x for LULESH on a single locale. First, I tried those optimizations to see if they still benefit the performance in a multi-locale environment.

Two of my earlier optimizations from a single locale still help the multi-locale LULESH execution as you can see the speedups of "O1" over "original" in Table 5.9. The modification to function *CalcElemNodeNormals* improves the performance by 6% by minimizing the construction and destruction of temporary tuples. However, the biggest contribution is by safely hoisting several distributed local variables such as *dvdx/dvdy/dvdz, x8n/y8n/z8n, dxx/dyy/dzz* to the global space so that they won't be dynamically allocated whenever the function that declares them is called. I call this optimization "globalization" in the following description. Globalization is very important to multi-locale execution since creating and initializing distributed variables is expensive. Within a single locale, frequent data allocation and reclaim also cause thread contention, which is also bad for performance.

However, the scalability of LULESH is still not good enough although the execution time does seem to drop a little bit on the 32-locale case. I further examined the program by tracking the accesses of the most blamed variables. *localizeNeighborNodes* is an inline function that is called at multiple places to get the local copies of some attributes, like coordinates (x/y/z) and velocities (xd/yd/zd) of the neighboring *Nodes* of an *Element* for the purpose of optimization. However, it performs 32 (when I use the default value 8 for the parameter *nodesPerElem*) remote data accesses for each *Element* in a sequential order because the neighboring *Nodes* may not be on the same locale with the *Element*. Besides, the function is called inside deeply nested loops, so it still causes significant communication overhead.

Figure 5.8 illustrates an example of such as a case: The attributes of the blue *Element* and *Nodes* are stored on Locale 1 while the red ones are on Locale 2; the blue *Element* on the border of the two locales needs to access the red *Nodes* on Locale 2, which incurs a remote access.

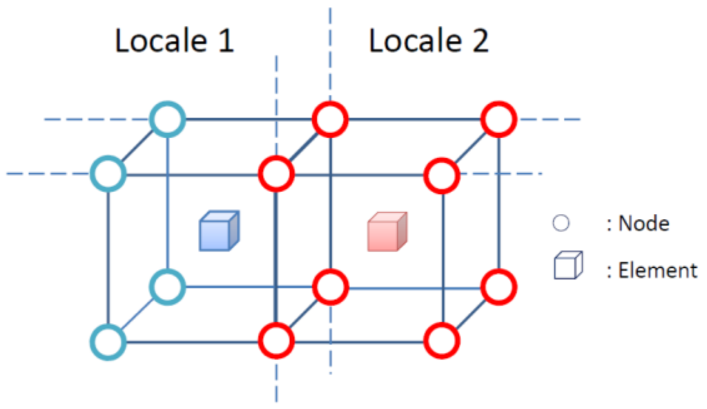


Figure 5.8: An Element-Node topology that would cause remote data accesses

To fix the problem, I allocate 6 new array variables: *x_map, y_map, z_map, xd_map, yd_map,* and *zd_map* to prestore the 8 neighboring *Nodes* for each *Element*. They use the same distributed domain *Elems* so that they can be read or written in a distributed parallel style, just like other *Elems* based distributed variables. Now except for the first call of *localizeNeighborNodes* in *initMasses*, I can remove all other calls to that communication-intensive function *localizeNeighborNodes* and simply do the copy to localize *Node's* attributes.

To update those map variables as the execution continues, I create a function *updateNeighborNodeMaps* using full parallelism (all available threads) to do so once in each *LagrangeNodal* call. In this way, I avoided redundant remote accesses to the data that has been accessed before and I refer to this optimization as “replication”. Replication brings more opportunities for localization, now I can enclose most computation into the “local” statement as long as they are within the loop iteration of same distributed domain *Elems*. I’ve found several functions that can benefit from this replication and localization combined optimization, such as *CalcHourglassControlForElems*. The performance improvement is shown in Table 5.9 (“O2” is the combination of all optimizations: globalization, localization, and replication). Overall, I improved the performance of LULESH by a factor of 4x on 32 locales. Significantly, I move from having slowdown as more locales were added to having speedups.

Table 5.9: Speedups of optimization on a different number of locales for LULESH

#Locales	2(12)	4(12)	8(4)	16(4)	32(4)
original (s)	17.70	17.99	19.84	22.80	28.26
O1 (s)	14.89	13.40	14.73	14.51	11.29
speedup-01	1.19x	1.34x	1.35x	1.57x	2.51x
O2 (s)	11.73	9.74	8.15	8.20	7.10
speedup-02	1.51x	1.85x	2.43x	2.78x	3.98x

5.4 Discussion and Summary

This chapter describes ChplBlamer-ML, a profiler to identify, quantify, and analyze the performance bottlenecks in multi-locale Chapel programs. Compared to the single-locale ChplBlamer, ChplBlamer-ML fully supports multi-locale, asynchronous and remote tasking; provides richer information such as inter-node load imbalance, and incurs much lower runtime overhead, from 3.5x to 14%. Guided by ChplBlamer-ML, I was able to pinpoint performance bottlenecks in three communication-bound multi-locale Chapel benchmarks and identify the causes in the user-level context. I used three optimization techniques: globalization, replication, and localization to improve three benchmark codes. With little modification to the code, I was able to achieve speedups of 1.05x for HPL, 1.11x for ISx, and 4.0x for LULESH on 32 locales over the previously fastest versions.

I also studied the overhead of ChplBlamer-ML. Table 5.10 shows the overhead study of ChplBlamer-ML on three multi-locale benchmarks. It shows the time spent in each step of a profiling, including a pre-run static analysis, an execution with sampling and instrumentation enabled, and a post-run processing. The runtime overhead is calculated with $(Monitored\ execution/Clean\ execution) - 1$, which shows the extra time cost of the sampling and instrumentation I added. The total overhead is calculated with $(Total\ profiling\ time/Clean\ execution) - 1$, while the *Total profiling time* is the sum of static analysis, monitored execution and post processing. The total overhead shows the overhead of one-time profiling of a particular benchmark. As is shown in the table, there is a big difference between the runtime overhead and total overhead due to the pre-run and post-run analyses. Fortunately, the static analysis runs only once for each benchmark. After that, users can experiment with different problem sizes as many times as they need, thus, the overhead of static analysis can be amortized. The post processing is proportional to the total number of samples, which is originally determined by an adjustable sampling rate that's used in ChplBlamer-ML. Therefore, the runtime overhead is the key overhead to users.

Further investigation reveals the major contributor to the runtime overhead; it's the instrumentation I added into the Chapel runtime library. Since it enforces a stack unwinding whenever a unique asynchronous task is created or a unique parallel region is met in context, the more unique parallel and asynchronous regions exist in a program, the higher runtime overhead it may incur.

Table 5.10: ChplBlamer overhead study

Benchmark name	**Clean execution**	**Static analysis**	**Monitored execution**	**Post processing**	**Runtime overhead**	**Total overhead**
HPL	19.38	7.82	22.16	10.24	14.3%	107%
ISx	2.13	5.68	2.18	1.35	2.3%	332%
LULESH	15.27	32.66	20.56	9.75	34.6%	312%

Unit: seconds

Chapter 6

Data-centric Profiling for GPGPU Applications

Historically, GPUs were used for graphics only. However, with the high demand of computing capability and the increased programmability of GPUs, people are seeking to apply GPUs for general purpose applications (GPGPU). Using a CPU-GPU hybrid computing framework is becoming a common configuration for mainstream supercomputers. The wide deployment of GPUs (as well as other hardware accelerators) brings to the HPC community a big question: Are we using them effectively? Unlike CPU programming, GPU programming must take into consideration GPU architecture specifications. Inappropriate use of GPUs will generate incorrect results in certain cases, but more often, will slow down the program instead of speed it up with its massive parallelism. In CUDA, functions that are launched by CPUs and run on GPUs are called "kernels". The performance of kernels directly determines how well we utilize the GPU devices. Unfortunately, besides NVP [57], there are few performance tools for CUDA programmers to detect performance bottlenecks and obtain insight for optimization. Therefore, my work, CUDABlamer, fills a critical need for programmers to analyze the runtime characteristics of kernels and obtain insights for optimizations for better GPU utilization. With CUDABlamer, we improved the kernel performance of two benchmarks by a factor of up to 46.6x.

6.1 Introduction

GPU profiling is a non-trivial but valuable problem. It is difficult because it requires in-depth knowledge of the characteristics of the GPU hardware and the complicated execution model of both the CPU and GPU involved. Combining modern computing systems with multi-core CPUs with multi-threading, plus the massive parallelism GPUs, the complexity of program execution substantially escalates. Therefore, the complexity to reasonably reflect the performance characteristics of these GPU-accelerated applications has significantly increased. There have been continuous efforts made in this field since GPU began to be popular in the HPC community. However, tool development has resulted in only a few profilers that can be used for GPU programmers. It is partially because the hardware is quickly upgrading and new features are being added to the architecture, while software support is still on the way. Moreover, CUDA [54], as the most commonly used GPU programming model, is not open-sourced, which limits effective measurement of the language performance by academic researches.

Currently, NVP [57] dominates the CUDA profiling and optimization needs. While it is an easy-to-use, information-rich performance tool with comprehensive performance metrics measurement and valuable optimization guidance, it has several limitations: 1st, it does not associate performance statistics to fine-grained data or code objects within launched kernels; 2nd, it does not provide complete calling context for each kernel launched. These two features are very desirable for CUDA programmers to understand performance and conduct fine-grained tuning.

Therefore, I have designed and implemented a performance analysis tool "CUDABlamer", for GPU-accelerated programs as an alternative approach for programmers who are interested in GPU kernel performance analysis and optimization. Based on the same "Blame" idea as was used in ChplBlamer, I use a static analysis and dynamic sampling combined approach to generate data-centric performance profiles for CUDA programs. Though the top-level idea and the framework are the same as ChplBlamer, shown in Figure 4.1, CUDABlamer needs to handle a number of language-specific issues in implementing such a data-centric profiler. As to the runtime information collection, I used PAPI to access CPU hardware counters, and its sampling mechanism drives my sampling-based approach in profiling. However, PAPI has no native support for GPU; its current support is essentially a simple wrapper of NVIDIA CUPTI library. In CUDABlamer, I used CUPTI to obtain the runtime information of the target applications.

The NVIDIA CUDA Profiling Tools Interface (CUPTI) provides performance analysis tools with detailed information about how applications are using the GPUs in a system. CUPTI provides two simple mechanisms to enable performance tools to understand the inner workings of an application and deliver valuable insights to developers. The first mechanism is a callback API that allows tools to inject analysis code into the entry and exit point of each CUDA C Runtime (CUDART) and CUDA Driver API function. Using this callback API, tools can monitor an application's interactions with the CUDA runtime and driver. The second mechanism allows performance analysis tools to query and configure hardware event counters in GPU and software event counters in the CUDA driver. These event-counters record activity such as instruction counts, memory transactions, cache hits/misses, divergent branches, and more [77]. How we utilized CUPTI will be detailed in Section 6.2.2.

6.2 Tool Design and Implementation

Profilers that monitor GPU kernel execution are complicated by the limited hardware support of fine-grained kernel measurement and the asynchronous concurrency that exists between the CPU and GPU. Programs that run on GPUs are treated like a black box, where measurements can only be read at the start and stop points of kernel launches. Therefore, pure timestamps based measurement of GPU-accelerated applications is too coarse-grained for complicated kernels. Most current profiling methods provide an overview of the behaviors of the application in a summarized manner without exposing sufficient low-level details. My tool, CUDABlamer, has been designed to provide more low-level details about kernel execution. More importantly, CUDABlamer is another instantiation of my "blame" idea for highly parallel programming models. I re-used the same basic framework of ChplBlamer, using LLVM based static analysis and sampling based dynamic analysis approaches to generate complete performance profiles, which will attribute performance data back to CUDA source data objects. In order to apply the unique data-centric profiling capability of ChplBlamer to CUDA programs, I have dealt with several CUDA-specific technical problems.

6.2.1 Language-specific LLVM Handling in Static Analysis

The two dominant programming models for GPUs are NVDIA's CUDA [54] and the cross-platform OpenCL standard [78]. While NVIDIA has previously open sourced their NVPTX code generator [79], encouraging language and compiler research and development, a completely open-source CUDA compiler is necessary for promoting general compiler and architecture research, especially in addressing the performance issues and understanding the execution characteristics. Fortunately, Jingyue and Artem, et al. [80] have proposed *gpucc*, an LLVM-based, fully open-source CUDA compatible compiler. The work has been integrated into LLVM toolchain [81] since LLVM 3.9 and is still in active development. To support the GPU in the test, GP100 [89], I used LLVM 4.1 and CUDA 8.0 package. When running static analysis on the CUDA IR (intermediate representation), I added some CUDA-specific language features that didn't need to be handled for Chapel.

First, CUDA runtime API and driver API calls. Common calls like *cudaMalloc,* associates a pointer variable with a memory allocation on the GPU; *cudaMemcpy* establishes a data-dependency relationship between host memory and device memory pointers; *cudaBindTexture* binds a memory area to a texture, therefore establishing a data-dependency relationship between the pointer pointing to the memory area on device and the texture reference, etc. Since my data-centric profiling approach depends on the complete data-flow information, I took special steps to analyze those library functions. Specifically, I manually figured out the blamed parameter indices of those CUDA API functions and prestore the corresponding parameter indices in a file; my tool retrieves the information and recovers the data-dependency relationships between parameters during the postmortem processing step. With the information of blamed parameters, CUDABlamer is able to propagate blame to the callers via blamed parameters for calls to those functions.

Second, the Clang frontend for CUDA generates IR in a style that is different from Chapel's LLVM frontend in some ways. For example, to handle the situation in GPU kernels where variables are defined in different GPU memory spaces: *global, shared, local*, it frequently uses the "*addrspacecast .. to*" instruction, which converts the pointer value from one type to another. Pointer conversions within different address spaces should use this instruction while "*bitcast .. to*" must be performed for pointers in the same address space. I basically mimicked the way that ChplBlamer processed "*bitcast .. to*" instruction so that the intra-procedural blame analysis will not be broken and correct source variables can be blamed through appropriate propagation among complex temporary variables.

Third, the Clang frontend for CUDA tends to produce composite instructions in the IR, which means an operand of an instruction can be another instruction. The most frequent instruction that has this kind of composite operands is the "getelementptr" (GEP) instruction. As we introduced in Section 4.1.2, the analysis of GEP instructions is very important in implementing my hierarchical blame attribution idea. The composite operand can be another GEP instruction or a casting instruction, such as *addrspacecast* or *bitcast*. In order to maintain the "Parent-Child" relationships between correct variables, I used a recursive approach to process it whenever a composite operand is met, keeping the necessary dataflow information complete and concise in the static analysis phase.

Last, name de-mangling. The names of functions and variables in CUDA IR are mangled, thus simply presenting the mangled names to users will be confusing. I retrieved native names from debug information and kept them along with mangled link names for each program object and finally presented native names in GUI.

6.2.2 Calling Context Construction for CPU-GPU Hybrid Framework

Besides the changes in the static analysis component, I basically rewrite the whole dynamic sampling and part of the postmortem analysis component, in order to construct the complete calling context for CUDA applications. The complete calling context I refer to here is defined as the combination of a complete call stack on GPUs (device) from the sampled point during the execution to the top-level kernel function, and the complete call stack on CPU (host) from that exact kernel's launch point to the top-level main function. To my knowledge, this work is the first attempt at gaining the complete calling context for a CPU-GPU Hybrid computing framework. It was a surprise to me at first that there was no existing work that provides such information, as its importance in performance profiling and tuning is obvious. Later, I realized the difficulty in doing so due to the limitation in both hardware support and complex asynchronous execution model of CUDA. In this work, I use static analysis combined with the runtime partial stack information to reconstruct the complete calling context. This approach has been evaluated on 16 open-source benchmarks and proved to be effective in most cases.

CUDABlamer utilizes the CUPTI Callback API and Activity API to sample the kernel execution and gain runtime stack information. Just like using the NVIDIA profiler, programmers need to insert two function calls: "*initTrace*" and "*finiTrace*" at the start and the end of the code region that they are interested in profiling. There is another option in CUDABlamer that programmers don't need to do anything to their source code and the profiling will begin when the program starts to execute. Currently, manually inserting those two calls is the default option since it allows programmers to explicitly control the scope of interest and avoid additional runtime overhead. After initializing the profiling, CUDABlamer will start tracking kernel execution as the program runs.

6.2.2.1 CPU Stack for Kernel Launch

To get the CPU stack trace for each kernel launch, CUDABlamer registers a callback in "*initTrace*", which is invoked whenever a kernel is launched. The callback method is a mechanism in the device layer that triggers callbacks on the host for registered actions or events. The callback function in CUDABlamer does a simple stack unwinding on CPU using libunwind [82]. While logging the stacktrace for each kernel launch, it also records the "correlationId" of that specific kernel launch, to be used as a unique ID when concatenating with the GPU stack.

6.2.2.2 GPU Sampling

We use sampling to get the CUDA program runtime characteristics. CUPTI provides an Activity API that allows asynchronous collections of a trace of an application's CPU and GPU activity. Moreover, CUPTI supports device-wide sampling of the program counter (PC). In CUDA, each SM (streaming multiprocessor) splits its own blocks into warps (currently with a maximum size of 32 threads). All the threads in a warp execute concurrently on the resources of the SM. The PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Samples are taken in Round-robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not. The PC

Sampling feature is only available on devices with compute capability 5.2 and higher. Therefore, in “*initTrace*”, CUDABlamer enables the tracking of PC Sampling activity by calling *cuptiActivityEnable(CUPTI_ACTIVITY_KIND_PC_SAMPLING)* and configures the sampling period by choosing one from five CUPTI pre-set options (MIN, LOW, MID, HIGH, MAX). CUPTI Activity API also provides an asynchronous buffering mechanism, with which you can record the activity data and deliver the data to output streams asynchronously. There are three types of activity information that CUDABlamer delivers: 1. FUNCTION: it records device function (including kernels) information, including the unique function ID, module ID and function name; 2. SOURCE_LOCATOR: it records the source line and file information for sampled instructions, with a unique ID for each new source locator; 3. PC_SAMPLING: it records the corresponding source locator ID, function ID, and correlation ID, which maps to the exact kernel launch that this sample is associated with. These activity records are collected as profiles and are written out to disk for further analysis.

6.2.2.3 Reconstruct the Calling Context

With all the runtime information collected in Section 6.2.2.1 and Section 6.2.2.2, CUDABlamer is able to derive the complete calling context for each sample in the postmortem processing step. As I explained in the previous section, each PC sample records 3 numbers: source locator ID, function ID, and correlation ID. The correlation ID can relate that particular sample to the CPU stacktrace of the associated kernel launch. The source locator ID and the function ID show the source line number and filename corresponding to that sample and in which device function that sample is triggered. Now we have the CPU stacktrace, but how do we rebuild the GPU stacktrace for a particular sample? Consider Figure 6.1 as an example: the GPU kernel *kernelFunc* calls two device function *foo* and *bar* at line 8 and line 18, respectively and function *foo* also calls *bar* at line 38. CUDABlamer has already obtained all call sites for each procedure earlier in the static analysis step; therefore, it's not difficult to build a call graph (Note the arrows are reversed to show the call path from the callee to the caller) for the sample code, displayed in Figure 6.2.

```
1     __global__ void kernelFunc(…){
           …
8          foo();
           …
18         bar();
           …
      }

28    __device__ void foo(){
           …
38         bar();
39         x = 1;              //Sample 1
40         y = 2;              //Sample 2
           …
      }

48    __device__ void bar(){
           …
56         A[i] = B[i] * s;    //Sample 3
           …
88    }
```

Figure 6.1: Sample CUDA code

Now suppose we have three samples gathered during execution, denoted as Sample 1, Sample 2 and Sample 3. For Sample 1 and Sample 2, their GPU stacks are easy to determine since the paths from the sample point to the kernel are unique. For Sample

3, however, there are two possible paths from the sample point to the top-level kernel: *bar->kernelFunc* and *bar->foo->kernelFunc*. Until now, limited by the context information from CUPTI library, CUDABlamer is unable to distinguish these two potential call paths. Therefore, it will create two stacktraces for the same sample with a weight of 0.5 for each stacktrace. The weight for each GPU stacktrace is basically calculated by dividing 1 by the number of possible call paths from the sample point to the top-level kernel.

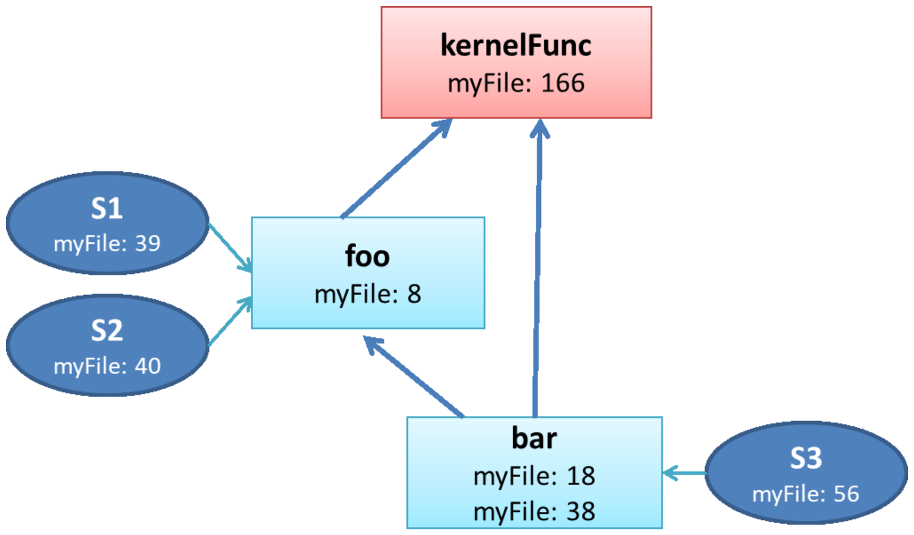


Figure 6.2: Reversed call graph for the sample code Fig. 6.1

Each sample starts with a source function and a destination function (E.g., S3 has a source function "*bar*" and a destination function "*kernelFunc*"). To derive all possible call paths for a particular sample from the source and destination functions, we developed a modified Depth-First-Search [83] algorithm to recursively traverse all possible call nodes in the reversed call graph. The pseudocode is shown in Figure 6.3. For each sample, it starts with a srcName and desName, representing the bottom frame where this sample is triggered and the top kernel frame on GPU, respectively.

The algorithm keeps adding frames to the call stack for this sample until *srcName* equals to *desName*, meaning it finishes finding a path from the sample point to the kernel.

```
void findAllPaths(string srcName, string desName, int
     &idx,   unordered_map<string,   bool>   &visited,
     Instance &inst, int srcLine, string srcFile) {

     create a frame for srcName;
     push it into the instance from this sample;
     set the function with srcName as visited;
     if srcName == desName
          finish the instance created for this sample;
          put the instance to the global instance map;
     else
          for all callers of this callee (srcName)
               if the caller is not visited
                    get the source line and file;
                    recursively  call  findAllPaths  on
the caller function with srcName being caller's name;


     pop out the new frame;
     set the function as unvisited;
}
```

Figure 6.3: Pseudocode of finding all possible call paths

This approach explores all potential GPU stacktraces for each sample; therefore, it is conservative and could incur high processing time. However, since most GPU call stacks will have no more than three levels and the ambiguity only happens when the sample is triggered from a device function that has multiple call sites within the same kernel launch, this method of combining the static calling context information with the runtime sample information works very well in most benchmarks we tested. Evaluation details can be found in Section 6.2.2.4. Moreover, since the call depth on GPU is usually shallow, the time cost of running my modified depth-first-search algorithm is negligible. I also pre-process the GPU sample profile to count the number of occurrences of each unique sample, using the combination of source locator ID, function ID, and correlation ID as the key to the sample. Therefore, CUDABlamer does not need to process every sample, but only those unique ones, which saves the overall postmortem processing time by orders of magnitudes. The blame weight of each sample is now calculated by the following formula:

$$bWeight(s) = cWeight * occurence$$

where *bWeight* represents the final weight of this sample with a particular call stack in blame calculation, *cWeight* is a fraction representing the share of this call stack out of all possible call stacks, *occurrence* represents how many times this unique sample was generated during the entire execution.

6.2.2.4 Evaluation

To evaluate the usability of my approach in constructing the complete calling context, I tried CUDABlamer on 16 benchmarks from two widely-used open-source benchmark suites. The two open-source benchmark suites are:

SHOC: SHOC [84] is a spectrum of programs that test the performance and stability of scalable heterogeneous computing systems. At the lowest level, SHOC uses micro-benchmarks to access architectural features of the system. At higher levels, SHOC uses application kernels to determine system-wide performance including intra and inter node communication among devices. I picked 8 benchmarks from SHOC 1.1.5.

Rodinia: Rodinia [85] includes applications and kernels which target multi-core CPU and GPU platforms. The Rodinia is inspired by Berkeley's dwarf taxonomy [86]. I picked 8 benchmarks from Rodinia 3.1 that can be compiled by gpucc [80] and represent different types of applications.

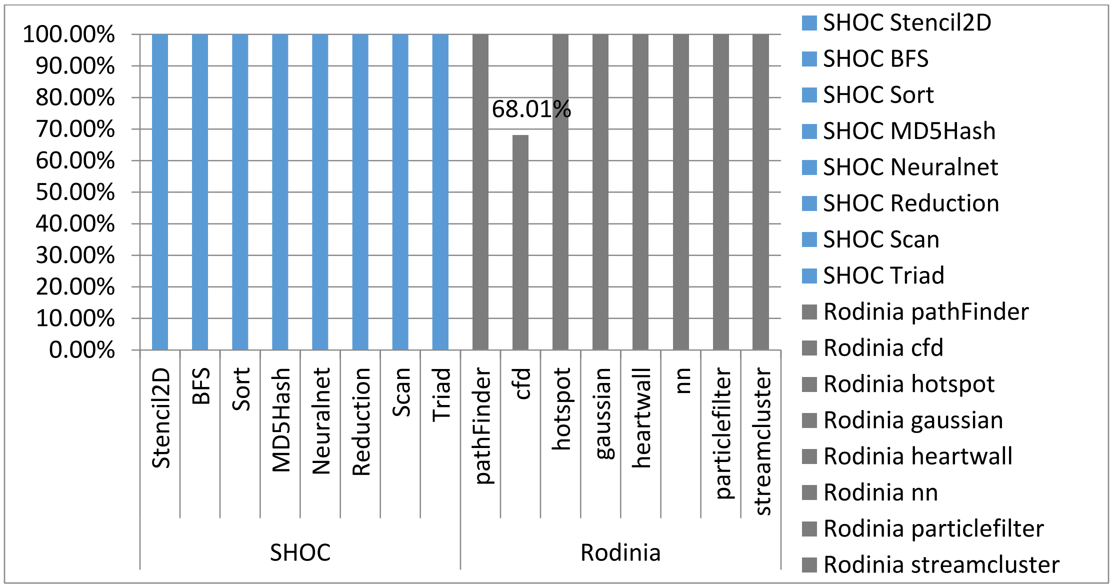


Figure 6.4: Coverage for SHOC and Rodinia benchmarks

The metrics for evaluating the usability of CUDABlamer's approach to a particular benchmark is called "coverage", determined by the following formula:

$$coverage = \frac{totalNumSamples - numAmbiSamples}{totalNumSamples}$$

where *numAmbiSamples r*epresents the number of samples that have more than one possible stacktraces ("ambiguous sample") and *totalNumSamples* is the total number of samples as the name tells. The *coverage* metric basically indicates what percentage of samples that are obtained from one run of a program will have a deterministic stacktrace from the sample point to the top-level kernel function. The higher the coverage is, the more precisely CUDABlamer can attribute time spent in that benchmark to variables and functions.

From Figure 6.4, only one benchmark *cfd* is not 100% covered. It is due to the calls to several inline function *compute_speed_sqd, compute_pressure, compute_speed_of_sound, compute_velocity,* and *compute_flux_contribution* at different points within the same kernel *cuda_compute_flux*.

6.3 Case Studies

To evaluate CUDABlamer, I tested it on 16 open-source benchmarks from two widely-used benchmark suites, as introduced in Section 6.2.2.4. From those, I studied the performance details of two benchmarks: *Particlefilter* and *Triad* and manually optimized the code. The experiments were done on a server with 2 NVIDIA Tesla P100 GPUs and 64 Intel Xeon Gold 6142 CPU processors (2.6 GHz). The NVIDIA P100 accelerator uses the Pascal architecture, featuring at extreme performance, including high speed, high bandwidth interconnect NVLink, and the first high capacity, highly efficient Chip-on-Wafer-on-Substrate stacked memory architecture HBM2. Each P100 GPU contains 16 GB on-chip memory and 56 SM (streaming multiprocessors). Each SM has 64 FP32 cores and 32 FP64 cores, which makes a total 5376 CUDA cores. Each SM also has 64KB of shared memory, which can be accessed as quickly as a register under certain access patterns. The entire GPU device also provides 48KB of constant memory, which is basically a global memory space but cached for frequent reads. Effective use of shared memory and constant memory can be very helpful for CUDA performance optimization. From a software perspective, the compilers that were used are nvcc 8.0, gcc 4.8.5 and clang 4.0.1 and "-O2" was used as the optimization option in the compilation.

6.3.1 Particlefilter

Particle Filter (PF) is a medical imaging application that is used for tracking leukocytes and myocardial cells. However, this algorithm can be used in different domains, including video surveillance, and video compression.

Table 6.1 shows the most blamed variables in Particlefilter, the information that CUDABlamer delivers to programmers include the blame percentage, name, type, calling context which tells whether the data object is allocated on the host or device.

Table 6.1: Profiling result of Particlefilter

Variable	Type	Context	Blame
ye/xe	double	main.particleFilter	100%
arrayX/arrayY	*double	main.particleFilter	100%
xj	*double	main.particleFilter	97.9%
yj	*double	main.particleFilter	97.8%
xj_GPU	*double	main.particleFilter	97.9%
yj_GPU	*double	main.particleFilter	97.8%
index	int	main.particleFilter.kernel	95.7%

xe/ye (100%): Estimated centroid object location, initialized at the beginning of each iteration for a frame and later calculated from particle coordinates. Therefore, in each round, the values depend on the result of the previous round of kernel execution.

arrayX/arrayY (100%): Array of coordinates of particles, reassigned every time after a kernel execution from *xj/yj* arrays.

xj/yj (97.9%/97.8%): CPU copies of *xj_GPU* and *yj_GPU*. Temporary arrays to store particle coordinates after the kernel execution.

xj_GPU/yj_GPU (97.9%/97.8%): Pointers to GPU allocations, storing the computed results of new particle coordinates. Each element is calculated by one CUDA thread.

index (95.7%): The main variable calculated in the kernel, indicating the index of the input arrays to be loaded to the corresponding output *xj_GPU/yj_GPU* element. therefore is attributed to most samples triggered within the kernel.

***Discussion and Optimization*:** The kernel is not complicated and does not call other device functions. Basically, each CUDA thread is responsible for loading one element in the output array based on the global Id of the thread. Therefore, simplifying the algorithm in the kernel is a dead end. However, CUDABlamer told us that *xj_GPU* and *yj_GPU* are the major blame holders. Since they are the only GPU arrays that are written while a few other arrays are read-only within the kernel, we can use GPU constant memory to store those read-only arrays. Constant memory is a global memory with cache, meaning except the cost for the first access, further reading the same memory addresses is almost as fast as on registers. Therefore, I moved 4 arrays (*arrayX_GPU, array_GPU, u_GPU, CDF_GPU*) that were previously allocated on the normal GPU global memory to the constant memory, resulting in an average speedup of 46.6x for the kernel performance, as shown in Table 6.2. The Larger the number-of-frames parameter is, the higher the kernel speedup will be since the cost of first-time access can be amortized. CUDABlamer also provides traditional code-centric profiling result, with a complete calling context. Since Particlefilter has only one kernel, the code-centric view attributes 100% to each function on the single call path: main->particleFilter->kernel.

Table 6.2: Performance comparison for Particlefilter

Original Kernel Execution (ms)	**Modified Kernel Execution (ms)**	**Speedup**
163.1	3.5	46.6x

6.3.2 Triad

From SHOC, this benchmark is a CUDA version of the STREAM Triad benchmark [87], which measures sustainable memory bandwidth for a large vector dot product operation on single precision floating point data. The benchmark uses a block-pipelined implementation to partially overlap the cost of the dot computation with the transfer of data from the host memory to the device memory. Table 6.3 shows the most blamed variables for Triad. First, I briefly introduced these variables.

h_mem (100%): a host-allocated big array that has the initial values for the vectors and stores the output data of kernel calls. It uses offset to properly store the value for three vectors in a continuous memory space, thus saving time on the memory allocation. Therefore the value of *h_mem* holds the ultimate result of running the entire program and is assigned 100% of the blame.

d_memC0/d_memC1 (50.8%/49.2%): The output vector of the vector dot production operation. The benchmark uses 2 copies to switch between computation and data movement, therefore the two output vectors share the total blame.

d_memB0/d_memB1 (14.9%/19.8%): The input vector B of the kernel.

d_memA0/d_memA1 (4.2%/5.3%): The input vector A of the kernel.

gid (7%): The global thread id calculated in each CUDA thread, acting as the index of an element in the vectors.

Table 6.3: Profiling result of Triad

Variable	Type	Context	Blame
h_mem	*float	main.RunBenchmark	100%
d_memC0	*float	main.RunBenchmark	50.8%
d_memC1	*float	main. RunBenchmark	49.2%
d_memB1	*float	main. RunBenchmark	19.8%
d_memB0	*float	main. RunBenchmark	14.9%
gid	int	main. RunBenchmark	7.0%
d_memA1	*float	main. RunBenchmark	5.3%
d_memA0	*float	main. RunBenchmark	4.2%

Discussion and optimization: The Triad benchmark has been widely-studied and highly optimized in multiple ways. The kernel is extremely small and there are no shared data accesses between threads within a block. Therefore, changing the memory types for vector allocations does not give us speedups. However, I still found an opportunity for optimizing the calculation of the output vectors. In the original code, each CUDA thread calculates one element, even though the memory accesses coalesce, the thread creation and destruction become the obvious overhead compared to the simple computation in the kernel. Therefore, I manually tuned the number of blocks allocated to a thread grid and use multiple operations per thread, which reduced the parallelism to some extent but also reduced the overhead by reusing active threads. The comparison between the original and modified code snippet is shown in Figure 6.5 and the performance comparison in Table 6.4 shows that we gained a speedup of 1.2x from this optimization.

```
__global__ void triad (float* A, float* B, float* C, float s)
{
    int gid = threadIdx.x + (blockIdx.x * blockDim.x);
    C[gid] = A[gid] + s*B[gid];
}
```

Original

```
__global__ void triad (float* A, float* B, float* C, float s)
{
    int gid = threadIdx.x + (blockIdx.x * blockDim.x);
    // do multiple calculations per thread
    for (; gid < nElems; gid += gridDim.x * blockDim.x)
        C[gid] = A[gid] + s*B[gid];
}
```

Optimized

Figure 6.5: Code comparison for Triad

Table 6.4: Performance comparison for Triad

Original Kernel Execution (ms)	**Modified Kernel Execution (ms)**	**Speedup**
20.87	17.75	1.2x

6.4 Discussion and Summary

Tuning code for GPUGPU and other emerging many-core platforms is challenging because there are few models or tools that can precisely pinpoint performance bottlenecks. Although several GPGPU profilers exist, most traditional tools, unfortunately, simply provide programmers with a number of different kinds of measurements and metrics obtained by running applications. As a result, it is very hard for users to map these metrics back to their source code to understand the root causes of slowdowns, much less decide what next optimization step to take to alleviate the bottlenecks and improve the overall performance. Some model-based approaches [59, 62] are able to provide such fine-grained information by appropriately mapping performance metrics to the application. However, the intrinsic deficiency of model-based approaches limits the application of those techniques. G-HPCTOOLKIT [64] employed some smart ways to shift the blame of slowdown back and forth between CPU thread execution and GPU tasks, while keeping a very low overhead. Although it provides the call stack on the CPU for each kernel launch, it does provide the call stack on the GPU. Therefore, if a complicated kernel generates a deep call stack on the GPU, G-HPCTOOLKIT [64] is not able to provide performance insights into that kernel. In comparison, my tool has the ability to provide users with the complete user-level calling context, from the CPU side to the GPU side. The call stack information can direct programmers precisely to the root causes of program slowdowns.

Lim [67] focuses on characterizing kernel executions and it uses the same sampling mechanism provided by CUPTI [66] library as I do. Although it provides more detailed kernel information “instruction mix” than previous performance analysis tools, it does not identify the bottlenecks in the source code that cause the performance slowdowns. In comparison, my techniques map the performance metrics back to source code elements so the programmers can have a straightforward idea of what is slowing down the program and what are the bottlenecks of the performance.

Moreover, none of the existing profilers provide data-centric metrics as I use in the performance analysis.

In summary, CUDABlamer distinguishes itself significantly in three aspects:

First, the tool offers fine-grained, in-depth performance analysis into the kernel execution, providing programmers much more insights about the functions and tasks executed on GPUs. The insights are straightforward and mapped to the source; therefore, programmers are able to quickly locate the hotspot data or functions.

Second, the tool uses a data-centric performance analysis technique for GPU-accelerated applications. I utilize the GPU hardware sampling technique to get sampled runtime information and map that back to source code variables.

Third, it is the first tool that offers the complete calling context in the execution profile, from the CPU side to GPU side, including the call stack before a kernel is launched and the call stack within a kernel.

With CUDABlamer, I studied the performance of 16 GPU benchmarks and optimized two of them. For Particlefilter, I gained a speed of 46.6x by simply moving some read-only data to the constant memory space on the GPU. For Triad, I gained a speedup of 1.2x by reusing active threads. Moreover, I again demonstrated the usability and applicability of the data-centric idea "Blame" in performance analysis of high parallel programming models by instantiating another performance tool for a widely-adopted HPC programming model.

I also studied the overhead of CUDABlamer; the result is shown in Table 6.5, using the same categories as Table 5.10. As I explained in Section 5.4, the runtime overhead is the major concern of our profiler. From Table 6.5, we see a significant difference in the runtime overhead between benchmarks. With further investigation, I found that the high runtime overhead is largely due to the poor performance of PC_SAMPLING mechanism from the CUPTI library. Therefore, when the kernel is large and complex, the runtime overhead can surge, such as Streamcluster. Unfortunately, PC_SAMPLING is relatively new and currently the only available tool that supports sampling the GPU execution. I will keep looking for optimizations and hopefully, its performance will be improved in the future CUDA releases.

Table 6.5: CUDABlamer overhead study

Benchmark name	**Clean execution**	**Static analysis**	**Monitored execution**	**Post processing**	**Runtime overhead**	**Total overhead**
Hotspot	10.43	1.61	10.82	0.83	3.7%	27.0%
Streamcluster	16.96	2.54	115.35	55.46	580%	922%
Particlefilter	10.21	1.34	11.1	1.74	8.7%	38.9%

Unit: seconds

Chapter 7

Conclusions

This chapter summarizes the conclusions of this dissertation. Section 7.1 summarizes the contributions of this dissertation; Section 7.2 describes some open problems that are opportunities for future work.

7.1 Summary of Contributions

Using static analysis, plus the sampling-based measurements triggered by hardware performance counters in conjunction with call path profiling, I was able to develop data-centric profilers with reasonable overhead to analyze program executions on a parallel architecture with many hardware threads, deep memory hierarchies, and GPU accelerators. These methods can provide valuable insights to guide code optimization.

New Performance Attribution for Emerging Programming Models

As supercomputing evolves, the hardware tends to be more distributed and heterogeneous to provide massive parallelism. Meanwhile, emerging parallel programming models that support software programming on these powerful machines are in active development. Newer parallel programming models provide newer abstractions for programmers. However, performance tools need to keep pace with these changes to present useful performance information in an instructive way. Some traditional performance attribution methods may not be sufficient to profile these newer programming models.

In this dissertation, I proposed a new performance data attribution method for two highly parallel programming models: PGAS and CUDA. The new attribution approach is referred to as data-centric profiling and is based on the performance metric *Blame* explained in Chapter 3. This data-centric profiling technique allows users to attribute performance data to program variables and data structures instead of functions and code regions. Today, it is the data instead of the computation that frequently becomes the bottleneck of the overall performance. Therefore, memory allocation, data storage, and inter-node communication are critical to the performance of an HPC system and thus data-centric performance measurement and mapping provide valuable insights into performance optimization.

To validate the applicability of my data-centric profiling idea, I designed and implemented two profilers for PGAS and CUDA, extending the Blame tool by Rutar [18]. For PGAS, I developed ChplBlamer, for both single-node and multi-node Chapel programs. It supports most Chapel language features and provides hierarchical profiling over program abstractions and call path profiling in the user context. I also augmented ChplBlamer with some new features such as data-centric inter-node load imbalance identification. The combination of the tool's data-centric and code-centric profiling provides insights into inter and intra node communication bottlenecks as well as optimization opportunities that could not be discovered by previous Chapel profilers. For CUDA, I developed CUDABlamer for GPGPU programs with features not available in previous CUDA profilers. More importantly, I used the same tool framework, the same *Blame* metric, and the same Graphical User Interface (GUI) as ChplBlamer uses to implement CUDABlamer. Specifically, the use of pre-run static analysis of a language-independent intermediate representation (LLVM) combined with minimum necessary runtime data and thorough post-run processing is proven to be a generic approach to build performance tools for different parallel programming models. My tool framework is extensible to support other languages.

Complete User-level Calling Context

This dissertation also shows the importance of constructing a complete user-level calling context for runtime samples in effectively delivering performance issues to programmers. I used different strategies to construct a complete calling context for PGAS and CUDA based on their execution models. The functionality to get the complete user-level calling context for Chapel and CUDA was not achievable from existing performance analysis tools as far as I know.

For the PGAS language Chapel, I added lightweight instrumentation in the Chapel runtime library, inside tasking and communication layers and used CPU sampling to gain runtime data from one run of a program. I reconstructed the complete user-level calling context in the post-run analysis step based on the logged runtime data and the pre-run static information to minimize the intrusion to the program's execution.

For CUDA, I used the CUDA Profiling Tools Interface (CUPTI) to do GPU sampling and added lightweight instrumentation to each kernel launched. Besides, I constructed a static calling context with the help of the pre-run static analysis. With the source information recorded along with runtime samples, I used a modified Depth-First-Search algorithm to determine the actual calling context for each collected sample. More details can be found in Section 6.2.2.

Valuable Performance Insights

To evaluate the effectiveness of ChplBlamer and CUDABlamer, I tested both ChplBlamer and CUDABlamer with several widely-studied open-source benchmarks. From the profiling results, I derived valuable insights into each program and found optimization for each benchmark that I have studied. For single-locale ChplBlamer, I gained a speedup of up to 2.3x and concluded that users should restrain using domain remapping and zippered iterations because they are expensive features to use in Chapel's current implementation. For ChplBlamer-ML, I gained a speedup of up to 4.0x and concluded that using some techniques like localization, globalization, and replication can significantly improve the performance and scalability of a multi-locale benchmark. For CUDABlamer, I gained a speedup of up to 46.6x for a GPU kernel execution and concluded that appropriate use of special GPU memories, such as constant memory and shared memory, can be of great benefit to the kernel performance. Also, creating too many parallel threads with little work on each thread sometimes hurts the overall performance. These programming experience and performance insights are valuable to the development of PGAS and CUDA as well as to application developers.

7.2 Open Problems

In this section, I present some high-level ideas for future work. I plan to extend approaches described in this dissertation in four ways: finer blame attribution, blame combined with auto-tuning, GPU read-only memory identification, and blame used in taint analysis.

7.2.1 Finer Blame Attribution

Currently, my data-centric profilers only present the blamed variables in descending order to reveal the possible performance bottlenecks. However, programmers cannot determine why certain variables stand out in the final result and such information can be very useful for precise optimization and better understanding the execution characteristics. From my experience, in a multi-thread and multi-node computing system, the synchronization among threads within a node and among multiple nodes is commonly one of the major performance bottlenecks in HPC. Also within a CPU-GPU hybrid architecture, the synchronization between the host (CPU) and the device (GPU) also consumes a big portion of execution time in many GPU-accelerated applications I studied. It would be great if ChplBlamer and CUDABlamer can associate those specific performance bottlenecks to the corresponding variables, such as shared memory variables within a GPU kernel block that needs synchronization between the threads within that block, or a distributed variable that is allocated across nodes and needs synchronization between nodes to continue execution. With this kind of strengthened performance mapping technique, a data-centric profiler can be more effective in guiding user-level optimizations.

7.2.2 Blame Combined with Auto-tuning

Our profiling system can figure out the time spent in populating the value of each program abstraction. Many of the variables shown in our results have tunable parameters that affect how much computation goes into calculating the data for that variable. These tunable parameters range from the communication patterns used for distributed data structures to the underlying data structures that are used to represent the variable such as whether to use a sparse or dense matrix. Using Chapel as an example, the data parallelism provides the domain control over array-like variable allocation. Domains are first-class index sets, which specify the size and shape of arrays. For distributed memory systems, different domain maps can make a big difference in managing load balancing and minimizing communication cost between nodes. Domain maps are “recipes” that instruct the compiler how to map the global view of computation to a locale’s memory and processors. Chapel provides a library of standard domain maps to support common array implementations, and the user can switch between domain maps effortlessly without changing other code. Auto-tuning has established itself as an important tool in HPC. It has been used in place of complex analysis to optimize everything from linear algebra libraries to parallel multicore stencil computations [27]. My future work will also investigate the integration of data-centric profiling and auto-tuning; specifically, I use my profiling framework to get the most blamed variables to reduce the state space and use certain existing auto-tuning framework, such as Active Harmony [28] to tune parameters (e.g., domain map).

7.2.3 GPU Read-only memory Identification

In my evaluation of CUDABlamer, I found an opportunity for optimizing the kernel in the benchmark Particlefilter (Details can be found in Section 6.3.1). Simply placing some read-only data in the constant memory instead of the general global memory in

GPU could bring dramatic performance improvement. The limitations in this optimization are: 1st the size of available constant memory on any current GPUs is small (48KB on P100); 2nd the memory allocation on the constant memory must be done at compilation time. The only effort in this optimization is to manually identify the fitting read-only variables on the GPU and move them to the constant memory space. Therefore, it would be nice if my tool can automatically identify read-only variables on the GPU along with their size information. This would help a user change their allocation without manually searching for optimizable objects. However, one thing to note is that the cost of the first access to the constant memory is very high, which would probably undermine the performance gain from subsequent accesses if reading the same memory space is not frequent enough. Therefore, profiling how often something is accessed is necessary before applying this optimization.

7.2.4 Blame Used in Taint Analysis

Taint analysis is a prevalent approach to detect malicious behavior in programs. Based on the concept that some data (such as the input from the user or any data from the website) is not trustworthy, taint analysis is proposed to keep track of the data which can be used to harm the software, and monitor suspicious actions. There are many previous uses of taint analysis [29, 30, 33, 35]. There are two categories of taint analysis: Static Taint Analysis (STA) and Dynamic Taint Analysis (DTA). DTA is more attractive because it allows us to reason about actual executions [31]. There are two limitations of DTA: 1st under-taint due to the tested inputs missing control-flow information; 2nd prohibitive runtime overhead due to the fact that it needs to examine every executed instruction. There exists some work composing static and dynamic methods to resolve the issues [32, 34, 35, 36, 37]. However, the analysis's runtime cost is still way too high.

My blame analysis is more than a profiler; it can also be used in other fields that require precise dataflow analysis yet need to limit performance impact. Blame analysis can tell programmers what statements in the program will contribute to the value of a particular variable. Since the data dependency information for each variable is mutually inclusive, we can obtain the reverse information: with any variable in the program, you can know what statements or variables that variable will touch. It closely resembles the taint analysis. Meanwhile, the blame system has a very low runtime overhead when using a relatively low sampling rate. To optimize the procedure of taint analysis, I would investigate a novel blame-assisted approach for DTA using sampling. Firstly, we can leverage the dual attributes of our blame analysis to get most intraprocedural dataflow analysis results. During the runtime, I will use sampling to focus on a small portion of variables in taint analysis. However, what type of sampling mechanism to use is still to be determined. This blame-assisted approach provides a tradeoff between monitoring coverage and overhead.

Bibliography

[1] B. Chamberlain, D. Callahan, and H. Zima, “Parallel programmability and the Chapel language,” Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online] Available: http://dx.doi.org/10.1177/1094342007078442

[2] Lydia Duncan, “Chapel: A Productive Parallel Programming Language”, Women Techmakers Community Talks, January 19, 2016. [Online] Available: http://chapel.cray.com/presentations/Duncan-WomenTechmakers.pdf

[3] Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)

[4] Geimer, M., Wolf, F., Wylie, B.J.N., ´Abraham, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)

[5] Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with vampir, vampirserver, and vampirtrace. In: Parallel Computing: Architectures, Algorithms, and Applications, vol. 15, pp.637–644. IOS Press (2008)

[6] Rogue Wave Software, "ThreadSpotter manual, version 2012.1," http://www.roguewave.com/documents.aspx?Command=Core Download&Entry Id=1492, August 2012.

[7] Lachaize, Renaud, Baptiste Lepers, and Vivien Quéma. "MemProf: A Memory Profiler for NUMA Multicore Systems." USENIX Annual Technical Conference. 2012.

[8] Buck, Bryan R., and Jeffrey K. Hollingsworth. "Data centric cache measurement on the Intel Itanium 2 processor." Proceedings of the 2004 ACM/IEEE conference on Supercomputing. IEEE Computer Society, 2004.

[9] Adhianto, Laksono, et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs." Concurrency and Computation: Practice and Experience 22.6 (2010): 685-701.

[10] Loveman DB. High performance fortran. IEEE Parallel & Distributed Technology: Systems & Applications. 1993 Feb;1(1):25-42.

[11] Tallent, Nathan R., and Darren Kerbyson. "Data-centric performance analysis of PGAS applications." Proc. of the Second Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST), San Servolo Island, Venice, Italy. 2012.

[12] Su, Hung-Hsun, Max Billingsley III, and Alan D. George. "Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming." Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 2008.

[13] Seisei Itahashi, Yoshiki Sato and Shigeru Chiba. “Toward a profiling tool for visualizing implicit behavior in X10” The 2014 X10 Workshop (X10'14) co-located with PLDI'14, Edinburgh, UK, 2014

[14] Oeste, Sebastian, Andreas Knüpfer, and Thomas Ilsche. "Towards Parallel Performance Analysis Tools for the OpenSHMEM Standard." Workshop on OpenSHMEM and Related Technologies. Springer International Publishing, 2014.

[15] TAU for Chapel, Available: http://www.nic.uoregon.edu/tau-wiki/Guide:TAUChapel

[16] Vöcking, Heye. "Performance analysis using Great Performance Tools and Linux Trace Toolkit next generation." p. 17. (2012).

[17] Liu, Xu, and John Mellor-Crummey. "A data-centric profiler for parallel programs." 2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE, 2013.

[18] Rutar, Nickolas Jon. Foo's To Blame: Techniques For Mapping Performance Data To Program Variables. Dissertation. 2011.

[19] Buck, Bryan, and Jeffrey K. Hollingsworth. "An API for runtime code patching." The International Journal of High Performance Computing Applications 14.4 (2000): 317-329.

[20] Mucci, Philip J., et al. "PAPI: A portable interface to hardware performance counters." Proceedings of the department of defense HPCMP users group conference. 1999.

[21] R. B. Johnson, J. K. Hollingsworth, "Optimizing Chapel for Single-Node Environments", In CHIUW workshop of the 30th IEEE International Parallel & Distributed Processing Symposium, Chicago, IL, 2016

[22] Sidelnik, Albert, et al. "Performance portability with the chapel language." Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE, 2012.

[23] Almasi, George. "PGAS (Partitioned Global Address Space) Languages." Encyclopedia of Parallel Computing. Springer US, 2011. 1539-1545.

[24] Intel Itanium Processor Reference Manual for Software Development http://people.freebsd.org/~marcel/refs/ia64/itanium/24532003.pdf

[25] Hayashi, Akihiro, et al. "LLVM-based communication optimizations for PGAS programs." Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. ACM, 2015.

[26] Nelson, Philip A., and Greg Titus. "Chplvis: A Communication and Task Visualization Tool for Chapel." Parallel and Distributed Processing Symposium Workshops, IEEE, 2016.

[27] Chen, Ray S., and Jeffrey K. Hollingsworth. "Angel: A hierarchical approach to multi-objective online auto-tuning." Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers. ACM, 2015.

[28] Ţăpuş, Cristian, I-Hsin Chung, and Jeffrey K. Hollingsworth. "Active harmony: Towards automated performance tuning." Proceedings of the 2002 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, 2002.

[29] Newsome, James, and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." (2005).

[30] Clause, James, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework." Proceedings of the 2007 international symposium on Software testing and analysis. ACM, 2007.

[31] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." Security and privacy (SP), 2010 IEEE symposium on. IEEE, 2010.

[32] Aggarwal, Ashish, and Pankaj Jalote. "Integrating static and dynamic analysis for detecting vulnerabilities." Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International. Vol. 1. IEEE, 2006.

[33] Kang, Min Gyung, et al. "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation." NDSS. 2011.

[34] Zhang, Ruoyu, et al. "Static program analysis assisted dynamic taint tracking for software vulnerability discovery." Computers & Mathematics with Applications 63.2 (2012): 469-480.

[35] Chang, Walter, Brandon Streiff, and Calvin Lin. "Efficient and extensible security enforcement using dynamic data flow analysis." Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008.

[36] Greathouse, Joseph L., et al. "Highly scalable distributed dataflow analysis." Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society, 2011.

[37] Greathouse, Joseph L., et al. "Testudo: Heavyweight security analysis via statistical sampling." Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2008.

[38] Weiser M. Program slicing. InProceedings of the 5th international conference on Software engineering 1981 Mar 9 (pp. 439-449). IEEE Press.

[39] Hayashi, Akihiro, et al. "LLVM-based communication optimizations for PGAS programs." https://chapel-lang.org/CHIUW/2014/akihiro.hayashi.CHIUW2014.pdf.

[40] Coarfa, Cristian, et al. "An evaluation of global address space languages: co-array fortran and unified parallel C." Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2005.

[41] Carlson, William W., et al. Introduction to UPC and language specification. Vol. 576. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[42] Yelick, Kathy, et al. "Titanium: A high-performance Java dialect." Concurrency Practice and Experience 10.11-13 (1998): 825-836.

[43] Charles, Philippe, et al. "X10: an object-oriented approach to non-uniform cluster computing." Acm Sigplan Notices. Vol. 40. No. 10. ACM, 2005.

[44] Nieplocha, Jaroslaw, Robert J. Harrison, and Richard J. Littlefield. "Global arrays: A nonuniform memory access programming model for high-performance computers." The Journal of Supercomputing 10.2 (1996): 169-189.

[45] Tardieu, Olivier, et al. "X10 and APGAS at petascale." ACM SIGPLAN Notices. Vol. 49. No. 8. ACM, 2014.

[46] Dun, Nan, and Kenjiro Taura. "An empirical performance study of Chapel programming language." Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012.

[47] Kayraklioglu, Engin, et al. "PGAS Access Overhead Characterization in Chapel." Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016.

[48] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." Proceedings of the international

symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004.

[49] Dean J, Hicks JE, Waldspurger CA, Weihl WE, Chrysos G. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. InProceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture 1997 Dec 1 (pp. 292-302). IEEE Computer Society.

[50] GPGPU, https://devblogs.nvidia.com/author/mharris/

[51] Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU computing. Proceedings of the IEEE. 2008 May;96(5):879-99.

[52] Nickolls J, Dally WJ. The GPU computing era. IEEE micro. 2010 Mar;30(2).

[53] AI computing, NVIDIA Corp. https://www.nvidia.com/en-us/about-nvidia/ai-computing/

[54] Nvidia CU. Nvidia cuda c programming guide. Nvidia Corporation. 2011;120(18):8.

[55] Haque, Riyaz, and David Richards. "Optimizing PGAS overhead in a multi-locale Chapel implementation of CoMD." PGAS Applications Workshop (PAW). IEEE, 2016.

[56] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[57] NVIDIA Corp. Nvidia Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler, Jan 2013

[58] Baghsorkhi SS, Gelado I, Delahaye M, Hwu WM. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In ACM SIGPLAN Notices 2012 Feb 25 (Vol. 47, No. 8, pp. 23-34). ACM.

[59] Pennycook SJ, Hammond SD, Jarvis SA, Mudalige GR. Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. ACM SIGMETRICS Performance Evaluation Review. 2011 Mar 29;38(4):23-9.

[60] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994

[61] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. *In Proceedings of the IEEE International Parallel and Distributed Processing Symposium,* April 2008.

[62] Sim J, Dasgupta A, Kim H, Vuduc R. A performance analysis framework for identifying potential benefits in GPGPU applications. In ACM SIGPLAN Notices 2012 Feb 25 (Vol. 47, No. 8, pp. 11-22). ACM.

[63] Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In ACM SIGARCH Computer Architecture News 2009 Jun 20 (Vol. 37, No. 3, pp. 152-163). ACM.

[64] Chabbi M, Murthy K, Fagan M, Mellor-Crummey J. Effective sampling-driven performance tools for GPU-accelerated supercomputers. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis 2013 Nov 17 (p. 43). ACM.

[65] GPU energy efficiency, NVIDIA Corp. http://www.nvidia.com/object/gcr-energy-efficiency.html

[66] CUPTI, NVIDIA Corp. https://docs.nvidia.com/cuda/cupti/index.html

[67] Lim R, Malony A, Norris B, Chaimov N. Identifying optimization opportunities within kernel execution in GPU codes. In European Conference on Parallel Processing 2015 Aug 24 (pp. 185-196). Springer, Cham.

[68] Zhang H, Hollingsworth JK. Data Centric Performance Measurement Techniques for Chapel Programs. InParallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International 2017 May 29 (pp. 377-386). IEEE.

[69] Hui Zhang, Jeffrey K. Hollingsworth. 2018. ChplBlamer: A Data-centric and Code-centric Combined Profiler for Multi-locale Chapel Programs. In *Proceedings of ACM International Conference on Supercomputing, Beijing, China, June 2018 (ICS'18)*, 11 pages. https://doi.org/10.1145/3205289.3205314

[70] LLVM. The Often Misunderstood GEP Instruction . http://llvm.org/docs/GetElementPtr.html.

[71] "CORAL Collaboration Benchmark Codes," Oak Ridge, Argonne, Livermore. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[72] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. HPC Challenge Benchmarks in Chapel. Technical report, Cray, Inc., 2009.

[73] Hanebutte, Ulf, and Jacob Hemstad. "ISx: a scalable integer sort for co-design in the exascale era." Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on. IEEE, 2015.

[74] Hornung, R. D., J. A. Keasler, and M. B. Gokhale. Hydrodynamics challenge problem. No. LLNL-TR-490254. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[75] Malony AD, Biersdorff S, Spear W, Mayanglambam S. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. InProceedings of the 24th ACM International Conference on Supercomputing 2010 Jun 2 (pp. 127-136). ACM.

[76] Drongowski PJ. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Advanced Micro Devices. 2007 Nov 16.

[77] CUPTI developer, NVIDIA Corp. https://developer.nvidia.com/cuda-profiling-tools-interface

[78] Stone JE, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering. 2010 May;12(3):66-73.

[79] NVIDIA Corp. User guide for NVPTX back-end. http://llvm.org/docs/NVPXUsage.html, Sept. 2015

[80] Wu J, Belevich A, Bendersky E, Heffernan M, Leary C, Pienaar J, Roune B, Springer R, Weng X, Hundt R. gpucc: an open-source GPGPU compiler. In Proceedings of the 2016 International Symposium on Code Generation and Optimization 2016 Feb 29 (pp. 105-116). ACM.

[81] Compiling CUDA with clang https://llvm.org/docs/CompileCudaWithLLVM.html

[82] Mosberger, David. "The libunwind project." (2011). https://www.nongnu.org/libunwind/

[83] Depth first search, https://en.wikipedia.org/wiki/Depth-first_search

[84] Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS. The scalable heterogeneous computing (SHOC) benchmark suite. InProceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units 2010 Mar 14 (pp. 63-74). ACM.

[85] Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on 2009 Oct 4 (pp. 44-54). Ieee.

[86] Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley; 2006 Dec 18.

[87] Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas RF, Rabenseifner R, Takahashi D. The HPC Challenge (HPCC) benchmark suite. InProceedings of the 2006 ACM/IEEE conference on Supercomputing 2006 Nov 11 (Vol. 213).

[88] Intel Corporation, Intel R 64 and IA-32 Architectures Software Developer’s Manual Volume 3: System Programming Guide, June 2015.

[89] GP100, NVIDIA Corp. https://www.nvidia.com/en-us/data-center/tesla-p100/