RICE UNIVERSITY

Performance Analysis of Program Executions on Modern Parallel Architectures

by

Xu Liu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

John Mellor-Crummey, Chair
Professor of Computer Science and Electrical & Computer Engineering, Rice University

Vivek Sarkar
E.D. Butcher Professor of Computer Science and Electrical & Computer Engineering, Rice University

Peter Varman
Professor of Electrical & Computer Engineering and Computer Science, Rice University

James Browne
Professor of Computer Science, Physics, and Electrical & Computer Engineering, University of Texas at Austin

Houston, Texas
June, 2014
ABSTRACT

Performance Analysis of Program Executions on Modern Parallel Architectures

by

Xu Liu

Parallel architectures have become common in supercomputers, data centers, and mobile chips. Usually, parallel architectures have complex features: many hardware threads, deep memory hierarchies, and non-uniform memory access (NUMA). Program designs without careful consideration of these features may lead to poor performance on such architectures. First, multi-threaded programs can suffer from performance degradation caused by imbalanced workload, overuse of synchronization, and parallel overhead. Second, parallel programs may suffer from the long latency to the main memory. Third, in a NUMA system, memory accesses can be remote rather than local. Without a NUMA-aware design, a threaded program may have many costly remote accesses and imbalanced memory requests to NUMA domains.

Performance tools can help us take full advantage of the power of parallel architectures by providing insight into where and why a program fails to obtain top performance. This dissertation addresses the difficulty of obtaining insights about performance bottlenecks in parallel programs using lightweight measurement techniques. This dissertation makes four contributions. First, it describes a novel performance analysis method for OpenMP programs, which can identify root causes of performance losses. Second, it presents a data-centric analysis method that associates performance metrics with data objects. This data-centric analysis can both
identify both a program’s problematic memory accesses and associated variables; this information can help an application developer optimize programs for better locality. Third, this dissertation discusses the development of a lightweight method that collects memory reuse distance to guide cache locality optimization. Finally, it describes implemented a lightweight profiling method that can help pinpoint performance losses in programs on NUMA architectures and provide guidance about how to transform the program to improve performance.

To validate the utility of these methods, I implemented them in HPCToolkit, a state-of-the-art profiler developed at Rice University. I used the extended HPCToolkit to study several parallel programs. Guided by the performance insights provided by the new techniques introduced in this dissertation, I optimized all of these programs and was able to obtain non-trivial improvements to their performance. The measurement overhead incurred by these new analysis methods is very small in both runtime and memory.
Acknowledgments

I would first thank my advisor John Mellor-Crummey at Rice University. His guidance, inspiration, and support infuse this work and foster my interests in research. More specifically, this thesis consists of texts in the published research papers [88, 85, 87, 89, 86] written by John and I. I would also like to acknowledge current and former members on Rice HPCToolkit team: Laksono Adhianto, Michael Fagan, Mark Krentel, and Nathan Tallent for their development and maintenance of HPCToolkit. Without their effort, this dissertation would not have been possible.

I want to express my gratitudes to my friends in the Department of Computer Science at Rice University, such as Zhuhua Cai, Milind Chabbi, Karthik Murphy, Chaoran Yang, and friends outside of the the department, such as Jiebo Li, Chao Sun, and Jiakui Wang. They have spent time in discussing research issues with me and inspired my thoughts about innovations.

I would like to thank my wife, Shuyin Jiao, who has continued showing her love to me since 2004. Though we were in different countries for three years, she never stopped her support for me. Shuyin has given me concrete advise and help when I came across any difficulty. She is a good wife and also a good advisor in my life.

Finally, I would show my gratitudes and respects to my father, Yuyuan Liu and my mother, Sumei Yang. Your love and support drive me to pursue the Ph.D. degree. This dissertation is dedicated to both of you.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations ix
List of Tables xiii

1 Introduction 1

2 Background 7
  2.1 Hardware Sampling ................................. 7
  2.2 Call Path Profiling ............................... 10
  2.3 Overview of HPCToolkit ......................... 13
  2.4 Benchmarks for Evaluation ..................... 15

3 Related Work 18
  3.1 Performance Tools for OpenMP Programs ........ 18
  3.2 Measurement-based Tools for Analyzing Memory Bottlenecks 21
  3.3 Simulation Methods for Memory Bottlenecks .......... 23
    3.3.1 Performance Tools Supporting Reuse Distance Measurement. . 23
    3.3.2 Algorithms for Accelerating Reuse Distance Measurement . 25
  3.4 Tools for Identifying NUMA Inefficiencies ........... 27
  3.5 Comparison with Our Approaches .................. 29

4 Identifying Bottlenecks in OpenMP Programs 32
  4.1 Approach .......................................... 34
6.2 Lightweight Reuse Distance Measurement .................. 98
  6.2.1 Reuse Metric Computation .......................... 100
  6.2.2 Call Path Collection ................................ 101
  6.2.3 Data-centric Analysis ................................. 102
  6.2.4 Overhead and Accuracy Analysis ...................... 104
6.3 Tool Organization ........................................ 106
  6.3.1 Call Path Profiler .................................. 106
  6.3.2 Pin Plug-in .......................................... 107
  6.3.3 Post-mortem Analysis Tool ........................... 108
6.4 Case Studies .............................................. 108
  6.4.1 LULESH .............................................. 109
  6.4.2 Sweep3D ............................................. 112
  6.4.3 NAS LU ................................................ 113
  6.4.4 Sphot .................................................. 115
  6.4.5 S3D ..................................................... 116
6.5 Discussion ................................................ 117

7 Analyzing NUMA Inefficiencies in Threaded Programs 119
  7.1 NUMA-aware Program Design ............................... 123
  7.2 Address Sampling ........................................ 125
  7.3 NUMA Metrics ............................................. 127
    7.3.1 Identifying Remote Accesses and Imbalanced Requests .. 127
    7.3.2 NUMA Latency per Instruction ........................ 129
  7.4 Metric Attribution ....................................... 130
    7.4.1 Code- and Data-centric Attribution .................. 131
    7.4.2 Address-centric Attribution .......................... 131
  7.5 Pinpointing First Touches ................................ 133
  7.6 Tool Implementation ..................................... 135
7.6.1 Online Profiler ........................................ 135
7.6.2 Offline Analyzer and Viewer .............................. 136

7.7 Experiments ......................................................... 136
  7.7.1 LULESH ....................................................... 139
  7.7.2 AMG2006 ...................................................... 142
  7.7.3 Blacksholes .................................................... 145
  7.7.4 UMT2013 ....................................................... 147

7.8 Discussion .......................................................... 148

8 Conclusions ......................................................... 150
  8.1 Summary of Contributions .................................... 150
  8.2 Open Problems .................................................. 152

Bibliography .......................................................... 154
Illustrations

2.1 An example profile presented by hpcviewer for a parallel program running on a 48-core machine. .................................................. 14
2.2 An example trace view of hpctraceviewer for a parallel program running on a 48-core machine. ................................. 15

4.1 The call stack for master, sub-master and worker threads in an OpenMP program with three levels of nested parallelism. As shown, stacks grow from bottom to top. The calling contexts for the parallel regions shown in grey are scattered across two or more threads. . . . 44
4.2 HPCToolkit’s time-centric view rendering of a complete execution of AMG2006 on 64 cores – 8 MPI ranks with 8 OpenMP threads/rank. . 53
4.3 HPCToolkit’s time-centric view rendering of a single iteration of AMG2006’s solver on 64 cores – 8 MPI ranks with 8 OpenMP threads/rank. .................................................. 54
4.4 A calling context view of AMG2006’s solver phase. More than 12% of the execution time of all threads is spent idling in the highlighted parallel region. .................................................. 55
4.5 A bottom-up view of call path profiles for LULESH. madvise, which is called while freeing memory, is identified as the routine with the most idleness. .................................................. 57
4.6 Pseudo code showing how memory is allocated, initialized, and freed in LULESH. .................................................. 58
4.7 A calling context view of call path profiles for BT-MZ shows that the amount of idleness reported for **exch_qbc** is over seven times larger than the work it performs. 60

4.8 Bottom up view of HEALTH benchmark showing lock contention in the GOMP library. 61

5.1 Code-centric profiling aggregates metrics for memory accesses in the same source line; data-centric profiling decomposes metrics by variable. 69

5.2 A loop allocating data objects in the heap. 71

5.3 A simplified workflow that illustrates data-centric measurement and analysis in HPCToolkit. Rectangles are components of HPCToolkit; ellipses are inputs or outputs of different components. 72

5.4 The top-down data-centric view of AMG2006 shows a problematic heap allocated variable and its two accesses that suffer most remote events to this variable. 82

5.5 The bottom-up data-centric view of AMG2006 shows problematic allocation call sites which may reside in different call paths. 84

5.6 The data-centric view of Sweep3D shows heap allocated variables with high latency. Three highlighted arrays have high memory access latency. 85

5.7 The data-centric view of Sweep3D shows a memory access with long latency to array Flux. This access is in a deep call chain. 86

5.8 The data-centric view shows problematic heap allocated arrays with high latency in LULESH. The red rectangle encloses the call paths of all heap variables that incur significant memory latency. 88

5.9 The data-centric view shows a problematic static variable and its accesses with high latency in LULESH. The ellipse shows that variable's name. 89
5.10 The data-centric view associates a large number of NUMA related
events to a problematic heap allocated variable and its inefficient
accesses in Streamcluster benchmark. ........................................... 90

5.11 The data-centric view highlights two problematic heap allocated
variables and their inefficient accesses in Needleman-Wunsch
benchmark. ................................................................. 92

6.1 Shadow profiling used in reuse distance measurement ................. 99
6.2 The work flow of our prototype. ................................................... 107
6.3 Pinpoint problematic code sections and instructions in LULESH with
their full calling contexts. ....................................................... 109
6.4 The text output from the reuse distance measurement for LULESH.
It is an example for a reuse pair. The reuse instruction at line 1316
shows high memory access latency in Figure 6.3. ......................... 110
6.5 LULESH source code for the use and reuse described in Figure 6.4. It
shows an optimization opportunity for a pair of OpenMP parallel
regions. Arrow points to the use and reuse of \texttt{z8n}. The parallel loops
bounded with blue rectangles should be fused to improve data locality. 111
6.6 Identify problematic memory accesses and reuse points within nested
loops in Sweep3D. ................................................................. 113
6.7 Identify data locality optimization opportunity in NAS LU benchmark. 114
6.8 Identify problematic memory accesses with reuse annotations. ........ 115
6.9 Pinpoint one of problematic memory accesses in the S3D program
with it full calling context. .................................................... 117
7.1 Three common data distributions in a program on a NUMA architecture. The first distribution allocates all data in NUMA domain 1. It suffers from both locality and bandwidth problems. The second distribution maps data to each of the NUMA domains and avoids centralized contention. The third distribution co-locates data with computation, which both maximizes low-latency local accesses and reduces bandwidth contention.

7.2 Identifying the first touch context for each heap variable. Our NUMA profiler applies both code-centric and data-centric attribution for first touches.

7.3 Identifying NUMA bottlenecks in LULESH using code-centric, data-centric and address-centric attributions.

7.4 Address-centric view showing the overall access patterns of \texttt{RAP\_diag\_data} in AMG2006 across all threads.

7.5 Address-centric view for the accesses to \texttt{RAP\_diag\_data} in the most significant loop in AMG2006.

7.6 Address-centric view showing the overall access patterns of \texttt{RAP\_diag\_j} in AMG2006 across all threads.

7.7 Address-centric view for the accesses to \texttt{RAP\_diag\_j} in the most significant loop in AMG2006.

7.8 Address-centric view showing the access patterns of \texttt{buffer} across all threads.

7.9 Memory access patterns across threads in BlackScholes.

7.10 A loop kernel in UMT2013 that has many remote accesses to \texttt{STime}.
Tables

4.1 Running times for our case study applications with (a) the unmodified GOMP runtime library (no sampling), (b) the modified GOMP library with lightweight instrumentation to support blame shifting (no sampling), and (c) the modified GOMP library while profiling at 200 samples per second per thread. Execution times represent the average over three executions. .......................... 52

4.2 Summary of performance bottlenecks identified using our tool and our code optimizations in response. .......................... 63

5.1 Measurement configuration and overhead of benchmarks. ............... 81

5.2 Improvement for different phases of AMG2006 using coarse-grained numactl and fine-grained libnuma. .......................... 83

6.1 Assessing locality optimization opportunities. .......................... 97

6.2 Measurement overhead for our tool. .......................... 108

7.1 Configurations of different sampling mechanism on different architectures. .......................... 137
7.2 Runtime overhead measurement of HPCToolkit-NUMA. The number outside the parenthesis is the execution time without monitoring and the percentage in the parenthesis is the monitoring overhead. The absolute execution time on different architectures is incomparable because we adjust the benchmark inputs according to the number of cores in the system.
Chapter 1

Introduction

Over the past decades, computational modeling and simulation have become an integral component of science research. They not only complement and verify the experimental results, but also provide a solution for scientific problems that cannot be solved with traditional experimental methods. “Scientific computing has become an important contributor to all scientific research programs. [110]” To provide greater insight into the nature of physical phenomena, scientific programs have become increasingly sophisticated. With the exponential growth of computation and data used for modeling and simulation, scientific programs can no longer solve problems in reasonable time when executed sequentially. Today, parallel scientific programs routinely employ clusters and supercomputers to meet their computational needs. However, it is always difficult to obtain high performance. The difficulties come from two aspects: software and hardware.

On the one hand, parallel programs are expected to scale on large supercomputers. For example, ACM Gordon Bell Prize at SC13 [8] announced the winners for their fluid dynamics simulation, running with 6.4 million threads on Lawrence Livermore National Laboratory’s “Sequoia” cluster [80] and achieving a sustained performance of 14.4 PetaFlops. Usually, such large-scale programs employ sophisticated programming models for large-scale parallelism. To parallelize programs inside a node, one can use multithreaded models, such as Pthreads [26], OpenMP [111], Cilk [50], or Intel Threading Building Blocks [114]. To parallelize programs across
nodes, one can use a message passing model, such as MPI [97], or partitioned global address space (PGAS) programming models, such as Coarray Fortran [34], UPC [48], or Chapel [37]. How to choose appropriate programming models and efficiently use them for best performance is still an open question for program developers.

On the other hand, modern supercomputers are complex. First, modern supercomputers have become large scale. For example, Lawrence Livermore National Laboratory’s Sequoia BlueGene/Q cluster [80] has more than 1.5 million cores and supports more than six million threads with a peak performance of 20 PetaFlops. Recent supercomputers have begun to employ a large number of heterogeneous accelerators to increase their computational power. For example, Oak Ridge National Laboratory’s Titan cluster [109] has nearly 300 thousand CPU cores with 18 thousand GPUs with a peak performance of more than 20 PetaFlops. Another example of a heterogeneous cluster is Stampede [136] deployed at the Texas Advanced Computing Center. It integrates thousands of Intel Xeon processors with Intel Xeon Phi co-processors with a peak performance of nearly 10 PetaFlops. Second, modern supercomputers employ deep memory hierarchies to bridge the speed gap between CPU and memory. A typical processor has two or three levels of cache in the memory hierarchy. Recently, IBM announced their POWER8 processors that have four levels of cache [64]. Frequent data movement in such deep memory hierarchies not only degrades performance but also consumes energy. Third, modern multi-socket architectures integrate multiple microprocessors. Each microprocessor has its own memory directly attached. Such architectures have scalable memory bandwidth because different microprocessors can access memory in parallel. On a multi-socket node, a microprocessor can access remote memory because all memories are in a shared address space. However, accessing memory directly attached to a microprocessor is faster than accessing
memory attached to other microprocessors. This architecture is known as a non-uniform memory access (NUMA) architecture, which has become the standard for multi-socket nodes of scalable parallel systems.

Though powerful, mapping applications to architectures with these new features is difficult. First, massive parallelism in system can lead to poor performance, if threads or processes have imbalanced workloads. Second, a deep memory hierarchy cannot avoid long-latency accesses to main memory, if a program has cache-unfriendly memory access patterns, which often evict frequently used data from caches. Third, NUMA architectures may degrade performance of multi-threaded programs that lack a NUMA-aware design. Such programs suffer from excessive remote accesses and memory bandwidth contention.

To solve this inefficient mapping problem, one needs performance insights to guide code design and optimization. To obtain performance insights, using a performance tool to analyze program executions is the most straightforward method. Current performance tools mainly use two methods to understand program performance: simulation and measurement. A simulation-based tool feeds program instructions to a hardware simulator and monitors the program’s detailed execution. Simulation-based tools, such as SLO [16], MemSpy [95], ThreadSpotter [115], and MACPO [113] mainly focus on memory hierarchy bottlenecks. Based on detailed simulations, these tools can give insights into data access patterns, which provide useful guidance for code optimization. However, modern parallel architectures are complex for simulating every feature. Moreover, an exhaustive simulation used by SLO and MemSpy may cause 100x-1000x slowdown to the program execution. ThreadSpotter and MACPO adopt sampling techniques to selectively simulate the behavior of only some memory accesses to reduce measurement overhead.
Unlike simulation, measurement-based tools monitor program executions on real hardware. They can either instrument code to query performance metrics or sample performance events during program executions. Instrumentation-based tools, such as gprof [58], TAU [121], and SCALASCA [140], incur large measurement overhead if the instrumentation is fine grained. For example, TAU reports 30× slowdown [103] for monitoring and proposes overhead reduction by discarding data and losing accuracy. Unlike instrumentation, sampling-based tools, such as Intel Vtune [67], HPC-Toolkit [2], and Oprofile [82], can collect performance data with very low overhead.

In this dissertation, I describe new execution-analysis techniques that employ sampling in conjunction with both simulation- and measurement-based methods to study performance of parallel programs running on modern architectures with many hardware threads, deep memory hierarchies, and NUMA systems. In the rest of this chapter, I briefly describe the thesis statement, the contributions of this work, and the dissertation organization.

**Thesis statement** Using sampling-based measurements triggered by hardware performance monitoring units in conjunction with call path profiling, one can develop lightweight profiling methods to analyze program executions on a parallel architecture with many hardware threads, deep memory hierarchies, and NUMA systems. These methods can provide rich information to guide code optimization.

**Contributions** This dissertation describes the design and implementation of several new performance measurement and analysis methods for programs running on

---

*While gprof uses sampling to attribute execution time to functions, it uses compiler-inserted prologues and epilogues to attribute costs to call graph edges between functions. This can lead to high measurement overhead with small functions [51].*
parallel architectures. First, I developed a lightweight profiler for OpenMP programs. The profiler can collect performance data with low overhead and attribute them to the root causes of bottlenecks, i.e., code that causes load imbalance, oversynchronization, and serialization. Moreover, this method constructs user-level calling contexts for monitored parallel regions across OpenMP threads. Second, I developed a data-centric profiler that identifies memory bottlenecks. It leverages hardware sampling mechanisms to monitor memory accesses and associates performance metrics with both memory access instructions and accessed data objects. This data-centric profiler can quantify memory bottlenecks with very low overhead. Third, I developed a tool that combines measurement and simulation of memory hierarchies to guide memory bottleneck optimization. The tool can benefit from detailed simulation as well as low overhead sampling-based measurement. Finally, I developed a profiler that can identify and quantify performance bottlenecks in NUMA systems. Moreover, it can provide guidance for designing NUMA-aware code.

I implemented and integrated these novel measurement and analysis methods in Rice University’s HPCToolkit [2], which is the state-of-the-art performance tool for parallel programs. It employs hardware sampling and call path profiling techniques to collect performance data.

**Dissertation organization** Chapter 2 describes sampling based on hardware performance monitoring unit events, call path profiling, and HPCToolkit infrastructure which provide the foundation for the work described in this dissertation. Chapter 2 also describes the benchmarks that are used in this dissertation to validate our analysis methods. Chapter 3 reviews previous work in the area of program performance measurement and analysis. Chapter 4 describes the design and implementation of a
new profiler for OpenMP programs. Chapter 5 shows the design and implementation of a data-centric profiler for analyzing memory performance bottlenecks. Chapter 6 integrates sampling-based data-centric measurement with detailed cache simulation to provide additional insights for memory optimization. Chapter 7 describes the design and implementation of a lightweight profiler for guiding NUMA optimization. Finally, Chapter 8 presents some conclusions from our work and identifies a few directions for future research.
Chapter 2

Background

An effective performance tool for long-running parallel programs should have two basic features: low measurement overhead and insightful analysis. On the one hand, collecting performance data with low runtime and space overhead is essential for a performance tool to be useful for measuring production runs of HPC applications. On the other hand, a performance tool is valuable if and only if it provides insightful analysis that can guide performance optimization. This dissertation describes a set of tools that have both features. To achieve this, I built these tools based on two existing techniques: hardware sampling and call path profiling. Moreover, our approaches are based on the infrastructure of HPCToolkit [2], a performance tool developed at Rice University. The rest of this chapter reviews theses two techniques, the HPCToolkit performance tool, and benchmarks used to validate our approaches.

2.1 Hardware Sampling

For over a decade, microprocessors have included performance monitoring hardware to provide insight into an application’s behavior. Using a hardware performance monitor to periodically collect samples during an application’s execution is an effective way to associate work (e.g., instructions executed), resource consumption (e.g., cycles), and inefficiency (e.g., latency) with program regions. Such asynchronous sampling is the mechanism underlying our data-centric analysis of program executions. Most com-
monly, microprocessors contain hardware support for event-based sampling. Some processors support instruction-based sampling as well. We briefly review these two approaches.

**Event-based Sampling** Event-based sampling (EBS) uses hardware performance counters to count the occurrence of various kinds of events during a program execution. One configures an event counter to trigger an interrupt when a particular count threshold is reached. Typically, microprocessors support several performance counters, which enables multiple events to be monitored in a single execution.

On out-of-order microprocessors, EBS using hardware counters often does not accurately attribute events to the instructions that caused them. Several factors make precise attribution difficult, including long pipelines, speculative execution, and the delay between when an event counter reaches a threshold and when it triggers an interrupt. As a result, it is problematic to use EBS to support of a program execution at the level of individual instructions. In response, modern Intel processors support precise event-based sampling (PEBS), which can accurately attribute samples [125]; however, PEBS only supports a small subset of events [68].

As part of EBS and PEBS support, Itanium 2 processors provide *Event Address Registers* (EARs) [69] and Nehalem provides a *load latency facility* [68] to support data-centric analysis. These features enable one to sample loads and record information including effective address, cache and TLB misses, and access latency.

**Instruction-based Sampling** For more accurate attribution of performance events on out-of-order processors, Dean et al. [42] developed an approach known as instruction-based sampling (IBS). Periodically, IBS selects an instruction for monitoring. As a monitored instruction moves through the execution pipeline, hardware
records the occurrence of key events (e.g., cache and TLB misses), latencies of pipeline stages, the effective address of a branch target or memory operand, and whether the instruction completed or aborted. As a sampled instruction completes, IBS triggers an interrupt, signaling that the details of a monitored instruction’s execution history available for inspection. The first tool to use IBS was DEC’s Dynamic Continuous Profiling Infrastructure—a low overhead, system-wide, code-centric, flat profiler known as DCPI [6].

Modern AMD Opteron processors support two flavors of IBS: fetch sampling, and op sampling [45]. Fetch sampling is used to diagnose performance problems associated with instruction fetch. Op sampling provides information about instructions that complete. Information recorded about an instruction in flight includes the effective address (virtual and physical) for a load/store, details about the memory hierarchy response (e.g., cache and TLB miss or misalignment), and various latency measures. AMD’s CodeAnalyst uses IBS to precisely attribute processor events to code regions using flat profiling [4].

IBM POWER5 and subsequent generations of POWER architectures use a mechanism similar to IBS to count marked events [126]. When a marked event is counted, POWER processors set two special purpose registers. The sampled instruction address register (SIAR) records the precise instruction address of the sampled instruction. The sampled data address register (SDAR) records the effective address touched by a sampled instruction if this instruction is memory related. A PMU can be configured to trigger an interrupt when a marked event count reaches some threshold. When a sample is triggered, a tool can use SIAR and SDAR to attribute marked events to both code and data.
2.2 Call Path Profiling

A call path is a chain of functions with calling relationships. Associating performance losses with call paths provides unique performance insight into program executions [61, 62]. For example, consider a threaded program that employs task-based parallelism. If threads spend a large amount of time spin waiting in synchronization routines for access to shared resources, flat profiling without call path information cannot tell which task causes the waiting. Moreover, to understand data allocation in C++ programs, one can monitor allocation routines. However, without call path information, one cannot know the variable in the source code associated with any particular allocation performed by the C++ language runtime. Because of its potential to provide deep insight, call path profiling is widely used in analyzing program performance [118, 133, 66, 112], correctness [18], and security [127, 20]. Therefore, determining the full call path of any execution point of interest is important. Currently, there are three principal techniques to achieve this: call stack recording, precise call path encoding, and probabilistic calling contexts.

Call stack recording There are two principal techniques used to determine call paths during an execution: unwinding the call stack and maintaining a shadow call stack. These call paths are organized using compact data structures, such as call graphs [58, 59] or calling context trees [5, 7].

Call stack unwinding walks procedure frames on the call stack using either compiler-recorded information (e.g., libunwind [104], stackwalkerAPI [23], and Open-SpeedShop [118]) or results from binary analysis (e.g., HPCToolkit [133], VTune [66], and Oracle Solaris Studio [112]). Stack unwinding does not require instrumentation or maintain state information at each call and return. As a result, it adds no execu-
tion overhead, except when a stack trace is requested. This technique is well suited for coarse-grained execution monitoring, e.g., sampling-based performance tools and debuggers. However, applying call stack unwinding to gather calling context information for each machine instruction executed would frequently gather slowly changing calling context at unacceptably high overhead.

Stack shadowing \([58, 23, 107, 22, 29]\) involves maintaining calling context as the execution unfolds; this can be accomplished by instrumenting every function entry and exit either at compile time or using binary rewriting. The advantage of stack shadowing is that at every instant the stack trace is ready; and hence it can be collected in a constant time. Stack shadowing is well suited for tracking call paths of fine-grained instructions during program execution. A disadvantage of stack shadowing is that instrumentation of every call and return adds significant overhead, especially for functions called frequently. Moreover, it requires extra space to maintain the shadow stack. Such high runtime and space overhead makes the stack shadowing method difficult to apply efficiently in performance tools.

Instead of instrumenting all functions, one can choose to instrument a subset of functions on hot paths. Bernat and Miller \([14]\) developed a method that chooses a few functions to instrument. This selective instrumentation can balance the overhead and accuracy. Zhuang et al. \([144]\) designed an adaptive method to sample hot paths, called adaptive bursting. They developed a variety of approaches to reduce instrumentation overhead and obtain high accuracy.

**Precise call path encoding** Recording all call paths in program executions is useful for debugging \([18]\) and security analysis \([20]\). However, both call stack unwinding and stack shadowing incur high runtime and space overhead to do this. Precise call
path encoding is an alternate with low overhead.

Ball and Larus [12] first proposed an accurate and lightweight path encoding algorithm — BL algorithm to determine dynamic execution frequencies of control-flow paths in a routine. The BL algorithm places instrumentation in the edges of control flow graphs to encode control flow paths. Precise calling context encoding (PCCE) [129], based on the BL algorithm, uniquely represents the call path of any execution point using a series of numbers. These numbers are automatically generated by instrumenting the source code of a program at specific points in its call graph. With the lightweight instrumentation, call path encoding can achieve very low overhead. However, the use of PCCE is limited in practice. There are two following-on efforts that extend it. First, Li et al. [83] propose a dynamic encoding method that instruments binary on the fly for both executables and dynamically loaded libraries. This method can automatically analyze function pointers and avoid the profiling burden of PCCE. Second, Zeng et al. [142] develop an encoding method that works for both procedural and object oriented programs. It addresses the encoding space pressure to make the method applicable to large-scale programs.

**Probabilistic calling contexts** Precise call path encoding adds massive instrumentation to program execution. It can still cause high runtime and space overhead. To address this problem, one can encode call paths probabilistically with approximation. Bond and McKinley [19] developed a method to hash contexts to numeric identifiers. Because two different contexts may map to the same identifier, they proposed probabilistic calling contexts to efficiently determine calling contexts with high probability by using instrumentation of function call sites. Mapping identifiers to their call paths is not well supported in their approach. To increase the probability
of hashing contexts to unique identifier, Mytkowicz et al. [106] used program counters and sizes of call stack to encode contexts. To reduce the probability of multiple contexts mapping to one identifier, they modified program and activation records. Moreover, they trained program executions to build an offline map between contexts and identifiers, reducing the online encoding and decoding overhead. Bond et al. [18] further improved this approach by saving the offline training efforts and using online information to encode contexts.

2.3 Overview of HPCToolkit

HPCToolkit [2] is an open-source, multi-platform, sampling-based performance tool for measurement and analysis of application performance on parallel systems. It takes fully optimized binary executables as input, samples program executions, records call paths using call stack unwinding, and collects performance profiles and traces. As a code-centric profiler, HPCToolkit associates performance metrics with program code sections, such as statements, loops, and routines. HPCToolkit has two graphical user interfaces — hpcviewer [3, 2] and hpctraceviewer [134] to present profiles and traces respectively. Figure 2.1 is an example output of hpcviewer. There are three panes in the figure. The top one is the source code pane. The bottom left one shows program contexts with full call paths. These contexts can be configured to show as a calling context view, caller’s view, or flat view. The bottom right one shows the metrics associated with each program contexts. Both inclusive and exclusive values are computed for all metrics and can be configured to show selectively in the metric pane.

Figure 2.2 is an example output of hpctraceviewer. The main pane in the figure shows the trace view of program execution. Trace lines for processes and threads are
Figure 2.1: An example profile presented by hpcviewer for a parallel program running on a 48-core machine.

stacked along the vertical axis and time flows from left to right. The color at each point in a thread’s time line represents the procedure executing at that time. The top right pane shows the call path of the sample pointed by the white crossbar. By choosing different call path depths in this pane, the trace pane can show multi-level views. The bottom left is the depth view showing the depth of call stack along the timeline. Finally, bottom right pane is a mini map that indicates the fraction of the whole execution traces examined in the trace view.

Both hpcviewer and hpctraceviewer provide foundational support for intuitive analysis in our approaches described in this dissertation.
Figure 2.2: An example trace view of hpctraceviewer for a parallel program running on a 48-core machine.

2.4 Benchmarks for Evaluation

In this dissertation, we study several benchmarks to validate our new performance methods. We briefly describe these benchmarks below.

- AMG2006 [79], one of the Sequoia benchmarks developed by Lawrence Livermore National Laboratory, is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. The driver for this benchmark builds linear systems for various 3D problems. It consists of sequential and parallel regions that both affect its performance. AMG2006 is a MPI-OpenMP hybrid benchmark written in C.

- Sphot, one of LLNL’s ASC Sequoia benchmark codes [79], is a 2D photon transport code. Photons are born in hot matter and tracked through a spherical
domain that is cylindrically symmetric on a logically rectilinear, 2D mesh. It is a MPI+OpenMP benchmark, written in Fortran 77. We study a single-process, single-thread execution of the benchmark in this dissertation.

- UMT2013 is a benchmark from LLNL Coral benchmark suite [78]. It performs three-dimensional, non-linear, radiation transport calculations using deterministic methods. UMT2013 is coded in hybrid C, C++, Fortran and parallelized with MPI and OpenMP. In this dissertation, we only used OpenMP but not MPI.

- LULESH [77], a UHPC application benchmark developed by Lawrence Livermore National Laboratory, is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in 3D. In this dissertation, we study a highly-tuned LULESH implementation written in C++ with OpenMP.

- NAS BT-MZ [73] is a multi-zone parallel benchmark written in Fortran that employs two levels of nested OpenMP parallelism. It divides the data into several zones which are computed in the first level parallel regions. In each zone, computation is also parallelized in second level parallel regions. This benchmark manages load balance in the outer-level parallel regions.

- NAS LU [10] is a lower-upper Gauss-Seidel solver in the NAS parallel benchmark suite. This chapter examines its OpenMP version configured with class A workload running on one thread.

- HEALTH [40] is a benchmark that is part of the Barcelona OpenMP Tasks Suite. With the medium input, it creates more than 17 million untied tasks during execution. The benchmark is used to evaluate task creation and scheduling
policies inside an OpenMP runtime system.

- Sweep3D [1] is an ASCI benchmark that solves a time independent discrete ordinates 3D Cartesian geometry neutron transport problem. It is written in Fortran and parallelized with MPI without threading.

- Streamcluster [31] is one of Rodinia benchmarks. For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances metric. It is written in C++ with OpenMP.

- Needleman-Wunsch (NW) [31], another Rodinia benchmark, is a nonlinear global optimization method for DNA sequence alignments. It is also implemented in C++ with OpenMP.

- S3D, developed at Sandia National Laboratories, is a direct numerical simulation application that solves the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry [32]. S3D is written in Fortran 90 and is parallelized with MPI. We study a single process execution of the benchmark.

- Blacksholes is a benchmark from PARSEC benchmark suite [17]. It performs option pricing with Black-Scholes Partial Differential Equation (PDE). It is coded in C and parallelized using OpenMP.
Chapter 3

Related Work

This chapter reviews previous work on methods and tools that measure and analyze program executions on parallel architectures. Section 3.1 discusses performance tools that analyze bottlenecks in OpenMP programs. Section 3.2 and 3.3 review measurement-based and simulation-based tools and methods that identify performance losses in memory hierarchies respectively. Section 3.4 describes previous work on performance problems in NUMA architectures along with tools and optimization techniques to help address them. Finally, Section 3.5 distinguishes our approaches with existing work.

3.1 Performance Tools for OpenMP Programs

Tools for profiling and tracing OpenMP programs use either instrumentation or sampling as their principal measurement approach.

Examples of instrumentation-based tools are TAU [121], ompP [52], Scalasca [140], Vampir [60], and Periscope [55]. Recently, Score-P [99] was developed to provide an integrated instrumentation and measurement infrastructure for TAU, Scalasca, Vampir, and Periscope. These tools use the OPARI [102] source-to-source instrumentation tool to insert monitoring code around OpenMP constructs and the POMP [102] monitoring API to support measurement. POMP was proposed as a potential standard tool API for OpenMP [101]; however, it was not embraced by the OpenMP stan-
stards committee. Concerns about POMP include its reliance on instrumentation, the complexity of the API, and its runtime overhead. IBM provides a nearly complete implementation of the POMP API as part of their High Performance Computing Toolkit (HPCT) [33] and uses binary rewriting to add POMP calls to application executables. Due to concerns about overhead, IBM does not support the POMP API in optimized versions of their OpenMP runtime.

In contrast, tools such as Intel Vtune Amplifier XE 2013 [66], PGI’s Group PGPROF [137], and OpenSpeedShop [118] use sampling to monitor OpenMP programs with low measurement overhead. While Cray’s CrayPat [38] principally uses sampling, it relies on Cray and PGI compilers to add tracepoints to record entry/exit of parallel regions and worksharing constructs. However, none of these tools pinpoint code regions that cause idleness or lock waiting in an OpenMP program. With the exception of PGPROF, which relies on special characteristics of PGI’s OpenMP runtime, these tools cannot attribute performance information to user-level calling contexts and instead present an implementation-level view that separates a main thread from OpenMP worker threads.

Oracle Solaris Studio (OSS) [112] is a full-featured, sampling-based performance tool that supports analysis of OpenMP programs built using Oracle’s OpenMP compiler and runtime [138]. A key component of Oracle’s OpenMP runtime is an API that provides information to support sampling-based performance tools [71]. Using the API requires no compiler support. Oracle’s API confers two important advantages to OSS. First, it enables OSS to collect intricate performance metrics like overhead, idleness and work. Second, it enables OSS to attribute these metrics to full user-level calling contexts. A drawback of OSS, however, is that to collect profiles, OSS must record sample traces and then resolve full calling contexts from these traces.
during post-mortem analysis. Such traces quickly grow large as a program executes. A significant difference between OSS and our work is the approach for measurement and attribution of waiting and idleness. OSS attributes idle or spinning threads to contexts in which waiting occurs rather than trying to relate the idleness back to its causes. For example, OSS does not blame sequential regions for thread idleness. Consequently, it does not highlight sequential regions as opportunities for improving program performance.

To address the generic problem of attributing performance losses to their causes, Tallent and Mellor-Crummey [132] proposed a method for performance analysis now known as blame shifting. Their approach is based on the insight that an idle thread is a symptom of a performance problem rather than the cause. They describe a strategy for sampling-based performance analysis of Cilk programs executed by a work stealing runtime. Rather than a thread attributing samples to itself when it is idle, it announces its idleness to working threads using two shared variables. These two variables count the number of idle and working threads separately. At any asynchronous sample event, a working thread reads both variables and computes its share of the blame, which is proportional to the number of idle threads divided by the number of working threads, for not keeping idle threads busy. We refer to this strategy as undirected blame shifting as a thread has no knowledge of what other thread or threads will share the blame for its idleness.

Later, Tallent at el. [135] proposed a different blame shifting technique for lock spin waiting. In the extended technique, A thread waiting for a lock shifts blame for its waiting to the lock holder. We refer to this strategy as directed blame shifting, since an idle thread arranges to transfer blame to a specific thread. When OpenMP programs execute, various circumstances require either a directed or undirected blame
shifting strategy to attribute idleness or lock waiting to its underlying causes.

3.2 Measurement-based Tools for Analyzing Memory Bottlenecks

Prior work on measurement-based data-centric profilers only focuses on part of the problem. The remaining section describes tools that pinpoint poor temporal and spatial cache locality bottlenecks.

Irvin and Miller were perhaps the first to recognize the importance of data for performance [70]. In response, they extended Paradyn [100] to support a data view. They used Paradyn to dynamically instrument an executable to measure performance of semantic operations, such as collective communication primitives, and they used static analysis to associate measurements with data structures. When a user requested performance data for a particular array, Paradyn dynamically added instrumentation to collect it. Shortcomings of their approach included (1) their tool must know what semantic operations require instrumentation, (2) users must request performance data about particular arrays, (3) costs associated with data are measured using instrumentation rather than hardware counters, which means that details of hardware behavior (e.g. cache misses or latency) are unobservable, and (4) their tool presents performance data using time-line views rather than relating it to source code.

In the first work to employ asynchronous, event-based sampling (EBS) for data-centric analysis, Itzkowitz et al. introduced memory profiling to Sun ONE Studio [72]. They apply this tool to collect and analyze memory accesses in a sequential program and report measurement data in flat profiles. As described in Section 2.1, EBS has a shortcoming of “skid”, which leads to inaccurate attribution of samples to instruc-
tions. Sun ONE Studio attempted to compensate for EBS attribution skid using a backtracking search. However, their backtracking algorithm only succeeds if the instruction responsible for an event and the interrupt PC received in the signal handler are in the same basic block. Shortcomings of their tool are that (1) it requires that applications be specially compiled to pad the generated code with `nop` instructions to improve their ability to map events correctly and (2) it does not collect calling context information to attribute costs in context. Without information about full calling context, one can’t understand the relationship between data objects and program references when using complex, modular libraries such as PETSc [11].

Buck and Hollingsworth developed Cache Scope [24] to perform data-centric analysis using Itanium2 event address registers (EAR). Cache Scope associates latency with data objects and functions that accessed them. Unlike HPCToolkit, Cache Scope does not collect information about calling context and only associates latency metrics with code at the procedure level rather than at the level of loops or individual source lines.

Rutar and Hollingsworth have developed variable blame analysis [116] to map profiling information provided by hardware performance counters to high level data structures. They use data flow analysis to associate blame with program variables for costs that they observe at runtime using sampling. Shortcomings of their tool are similar to those of Sun ONE Studio: (1) it is susceptible to skid inherent in event-based sampling, described in Section 2.1 and (2) their tool requires that executables be specially compiled with LLVM [76], which provides information necessary for their data-flow-based blame attribution. It is not clear how their tool reports blame; the authors don’t describe a user interface and present the association between costs and data object names (without any association with code locations) in tabular form.
Tallent and Kerbyson extended HPCToolkit to support data-centric tracing [131]. However, their work focused on global arrays used in the Partitioned Global Address Space (PGAS) programming models, rather than heap-allocated or static variables used in a common program.

PerfExpert [25, 124, 49] tool developed at University of Texas analyzes data collected by HPCToolkit. To identify data locality bottlenecks, PerfExpert measures program executions to compute local CPI and associate it with code sections such as statements, loops, and functions. PerfExpert requires multiple runs of a program to collect a large set of hardware performance metrics. It automatically analyzes these data and presents results in a form that is easy to understand for users. However, PerfExpert does not directly provide data-centric analysis; it relies on a simulation tool — MACPO for detailed data-centric analysis; we discuss MACPO in the next section.

### 3.3 Simulation Methods for Memory Bottlenecks

We briefly describe related work from two areas, including tools leveraging reuse distance measurement to direct data locality optimizations, and algorithms for accelerating reuse distance measurement.

#### 3.3.1 Performance Tools Supporting Reuse Distance Measurement

Several tools measure and attribute reuse distance to assess data locality problems. Several tools associate reuse distance measurements with source codes to help guide data locality optimization. Beyls and H. D’Hollander developed Suggestions for Locality Optimizations (SLO) [15, 16] to diagnose temporal locality problems in a program’s execution. SLO uses a modified GCC compiler [53] to instrument every mem-
ory access instruction to collect data reuse information. Another compiler-based tool is MACPO [113]. MACPO uses profile feedback from UT’s PerfExpert [25, 124, 49] tool to instrument memory accesses in problematic code regions using the LLVM compiler infrastructure [90]. MACPO reduces instrumentation overhead by only analyzing a fraction of memory accesses. With compiler support, both SLO and MACPO can compute rich reuse information, including reuse distance, call paths and variables involved. However, as compiler-based tools, SLO and MACPO only collect the reuse distance data from the binary generated by their customized GCC compiler and LLVM compiler respectively. For code generated by Intel, IBM or PGI compilers that may optimize memory access patterns at compile time, SLO and MACPO cannot directly analyze its memory access patterns.

The work most closely related to our approach described in this dissertation using reuse distance simulation is ThreadSpotter, a commercial tool based on StatCache [13] and StatStack [47] simulators. Both StatCache and StatStack use the same statistical reuse pair measurement sampler to model caches with random and LRU replacement policies. Their sampler randomly selects a load or store instruction to monitor and sets up a watch point on the cache line touched by the memory access. These tools use watch points (implemented using page protection and single stepping) to trigger an interrupt when the same cache line is accessed later by a subsequent instruction. A monitored instruction and an instruction that subsequently accesses the same cache line form a reuse pair for performance optimization suggestions.

ThreadSpotter attributes reuse distance information for a reuse-pair to the source code line associated with the second access in the pair. To help pinpoint which variable access of many by a source line might be the problem, ThreadSpotter also indicates the source code line where the problematic memory location was last written;
since a source line usually writes only a single variable, this often will be helpful [115]. Moreover, ThreadSpotter does not aggregate information from many places in a program where a variable is accessed to associate performance problems with variables themselves (static variables, or allocation points for heap or stack data). ThreadSpotter’s sampling-based approach has been reported to have roughly 20% overhead [119]. A strength of ThreadSpotter is that it offers some suggestions about how one might improve the memory hierarchy utilization by a program.

Marin and Mellor-Crummey developed a tool [94] to collect reuse distance information and attribute it to calling contexts. Their tool does not quantify the impact of data locality bottlenecks in an execution, which means that data locality optimization efforts may yield little improvement. Moreover, their tool uses a cache simulator to assess the memory hierarchy response for every memory access, which has high overhead.

Marin et al. developed MIAMI [93], which is a simulation tool for diagnosing execution performance problems. MIAMI is based on static binary analysis and online binary instrumentation to collect reuse distance information. It uses a machine independent model to derive metrics, quantify potential optimization gains, and analyze causes of performance losses. MIAMI can identify a variety of program performance bottlenecks, such as insufficient instruction level parallelism, poor data locality, inefficient memory prefetching, and resource contentions.

3.3.2 Algorithms for Accelerating Reuse Distance Measurement

Reuse distance [54] of a memory location is a useful method to quantify a program’s data locality. It is defined as the number of unique cache lines accessed between the use and reuse of a memory location. For example, given a sequence of cache
line accesses—\( \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{b}, \mathbf{a} \)—the reuse distance for cache line \( \mathbf{a} \) is 2. For a fully associative cache, if the reuse distance for a cache line is larger than the cache size, a capacity cache miss occurs. Used as a performance metric, long reuse distance always means poor data locality. Unlike cache miss metrics, a reuse distance simulation can provide both use and reuse instructions, which are helpful for diagnosing data locality bottlenecks. However, collecting full reuse distance histograms can slow a program execution by 1000\( \times \) [16]. Several techniques have been developed to reduce this overhead. We classify these techniques into three categories: sampling, approximation and parallelization.

**Sampling techniques:** Besides the shadow profiling based sampling used in this chapter, there are other sampling methods. An optimized version of SLO [15] uses reservoir sampling to monitor only select accesses, which reduces measurement overhead from 1000\( \times \) to 5\( \times \). MACPO periodically enables and disables instrumentation to collect reuse information during sampling windows, yielding overheads that range from 1.25\( \times \)–5.52\( \times \). Zhong and Chang [143] describe a sampling-based approach that alternates between intervals of full monitoring and lightweight monitoring. They maintain and collect full reuse distance information only during full monitoring intervals. During lightweight monitoring intervals, they only maintain summary information that enables them to properly measure long-distance reuse that stretches beyond the end of an interval of full monitoring. This approach reduces reuse-distance measurement overhead to 2\( \times \)–4\( \times \). Schuff et al. [117] describe a similar approach that supports analysis of multithreaded program executions. Our tool executes monitored intervals concurrently with unmonitored execution, which reduces the impact of monitoring on overall execution time. However, since our tool simulates multiple monitored intervals concurrently in separate processes, it is unable to correct for
long-distance reuse beyond the end of a monitored interval as Zhong and Chang do.

**Reuse distance approximation:** Instead of collecting accurate reuse information with high overhead, some tools collect approximate information with low overhead. Shen et al. [120] use time to approximate the reuse distance instead of tracking memory accesses between instructions accessing the same cache line. This approximation yields a measurement overhead of $2 \times - 5 \times$. Ding and Zhong [44] use a tree compression strategy to approximate reuse distance rather than manipulating the complete balanced search tree used traditionally. Their tree compression strategy reduces measurement overhead from $O(N \log M)$ to $O(N \log \log M)$, where $N$ is the length of execution and $M$ is the size of program data.

**Parallelization:** Besides our work, several other researchers exploit parallel execution to accelerate reuse distance measurement using a divide-and-conquer approach. They divide the memory data reference traces into small pieces and compute reuse distance information for each piece in parallel. Niu et al. [108] use a cluster to compute reuse distance for serial programs, leading to a $13 \times - 50 \times$ speedup on 64 processors. Cui et al. [39] calculate reuse-distance information for serial programs on a GPU and obtain a $20 \times$ speedup.

### 3.4 Tools for Identifying NUMA Inefficiencies

Several performance tools provide support for analyzing NUMA performance issues with multi-threaded programs. These tools mainly use two kinds of methods: simulation and measurement. The simulation tools such as MACPO [113] and NUMA-grind [141] instrument programs to collect memory traces. NUMAgrind feeds memory traces into a cache simulator. The simulator simulates an architecture with NUMA memory hierarchies to analyze the memory traces. However, such simulation-based
tools add overhead that increases a program’s execution time as much as 100×. To reduce overhead, MACPO traces a subset of memory accesses and analyzes them post-mortem without using a simulator. MACPO’s instrumentation only increases a program’s execution time by 3-6×.

On the other hand, measurement-based tools can provide insights with low overhead. For example, Memphis [96] uses AMD instruction-based sampling (IBS) to capture remote accesses and associates them with static variables. It compares IBS with event-based sampling using traditional hardware event counters, showing that IBS gives deeper insights into a program’s NUMA bottlenecks. The author shows that IBS accurately attributes samples to instructions, unlike EBS. Moreover, IBS supports data-centric analysis, which can highlight problematic data objects that are accessed from different contexts. Compared to memory trace collection, IBS has much lower overhead.

MemProf [75], another measurement-based tool also uses AMD IBS to measure a program’s NUMA bottlenecks. MemProf associates NUMA metrics with heap-allocated variables and partially supports attribution to static variables by coarsely aggregating metrics incurred by static variables in the same load module (executable or shared libraries).

Besides NUMA profilers, some previous work improves NUMA locality and performance using support from libraries [21], compilers [92] and operating systems [36, 41]. These approaches aim to ameliorate NUMA problems to the greatest extent possible without source code changes. While these approaches require developers to use specific libraries, compilers, or OS kernels, our tool guides offline optimization of the source code which yields better code that can be run anywhere without any restrictions.
3.5 Comparison with Our Approaches

This section compares our tools with previous work described in this chapter and distinguishes our approaches. First, Section 3.1 describes existing tools that measure and analyze OpenMP programs. These tools use instrumentation or sampling to collect performance data. Existing instrumentation-based tools have relatively high overhead. For example, Morris et al. showed that instrumenting every procedure entry and exit in the MADNESS [63] code slows down execution by $30\times$ [103]. On the other hand, while existing sampling-based tools have low measurement overhead, they do not provide enough insights to guide code optimization. For example, Oracle Solaris Studio [112] supports lightweight measurement of OpenMP programs and constructs user-level calling contexts for sampled OpenMP parallel regions. However, it does not identify root causes of performance losses due to thread waiting at barriers or for locks. Tallent et al. [132, 135] developed two analysis methods to identify root causes of thread idleness in work-stealing programs and thread waiting for locks. However, these two methods are separate and are not designed for OpenMP programs. Moreover, Oracle Solaris Studio does not support construction of user-level contexts for statically linked binaries. In contrast, we developed a lightweight performance tool that addresses both of the aforementioned issues. The requirements for tool support that we identified in our research became the basis for a new tools application program interface for OpenMP. Our tool can identify root causes of performance losses in OpenMP programs using blame shifting and can attribute these losses to user-level calling contexts. We discuss these two approaches in Section 4.1.1 and 4.1.2, respectively.

Second, Section 3.2 presents the shortcomings of existing measurement-based tools for analyzing memory performance bottlenecks: insufficient support for parallel pro-
grams and lack of analysis of memory bottlenecks in both deep cache hierarchies and multi-socket nodes. For example, some tools [72, 24] only analyze cache locality for programs running on a single-socket machine; and some tools [96, 75] only analyze NUMA problems for threaded programs. No existing tool supports comprehensive analysis of all locality problems for large-scale programs coded with both distributed- and shared-memory models. Our data-centric tool, described in Section 5.2, addresses all of these difficulties and has efficient support for such hybrid parallel programs.

Third, Section 3.3 describes existing simulation-based tools for identifying memory bottlenecks. Their common drawback is high runtime overhead. To address this, our tool combines lightweight measurement and detailed simulation to reduce overhead but still provide rich information for analysis. The work most related to ours is ThreadSpotter [115]. It has modest measurement overhead to collect reuse distance during program executions. Unlike ThreadSpotter, our tool collects full calling contexts for reuse memory access pairs and supports data-centric attribution. We discuss these capabilities integrated with our novel lightweight reuse distance tool in Section 6.2.

Finally, Section 3.4 shows that existing tools for analyzing NUMA bottlenecks fall short of what we desire in one or more ways. Instrumentation-based tools suffer from relatively high overhead. For example, MACPO [113] samples a subset of memory accesses to reduce measurement overhead for NUMA problems. However, it still slows down program executions by 3-6×. On the other hand, tools that employ sampling-based measurement have lower overhead, yet they do not provide the full range of insights we desire. Specifically, Memphis [96] and MemProf [75] use hardware performance monitoring units to collect samples related to NUMA events. Both tools can attribute performance metrics to both program code and data objects.
However, they do not provide enough insights for optimization guidance. For example, they cannot directly tell programmers how to transform the code to optimize NUMA bottlenecks. To address this issue, our tool defines and computes derived metrics to quantify NUMA bottlenecks, described in Section 7.3. Section 7.4 describes how our tool attributes these metrics to code, data, and address ranges, which provides unique deep insights about NUMA bottlenecks by identifying important code sections, data objects, and access patterns. To guide code optimization for NUMA bottlenecks, our tool identifies data initialization points that determine data placement; this is described in Section 7.5.
Chapter 4

Identifying Bottlenecks in OpenMP Programs

Developing multithreaded programs for shared memory systems can be tedious, especially if one directly uses a primitive threading library. To simplify development of multithreaded programs, the high performance computing community has developed the OpenMP Application Program Interface (OpenMP API) [111]. The compiler directives, library routines, and environment variables collectively known as the OpenMP API define a portable model for shared-memory parallel programming. OpenMP is the preferred model for adding threading to MPI applications for recent supercomputers based on highly-threaded processors.

Given the importance of OpenMP, it is necessary to develop effective tools that pinpoint and quantify causes of performance losses in OpenMP applications. To address that goal, our research aims to develop effective performance analysis strategies and techniques for OpenMP, prototype these strategies and techniques in a tool to evaluate their utility, and ensure that these strategies and techniques are embodied in a general OpenMP performance tools API that is suitable for inclusion in the OpenMP standard. This chapter describes the results of our research.

There are several challenges for building effective performance tools for OpenMP. First, a tool must have low runtime overhead. Otherwise, any measurements it records will be suspect. Second, one must attribute performance measurements back to the program. In previous work [2], we have shown that attributing performance metrics to full calling contexts in parallel programs is essential to understand context-dependent
performance bottlenecks, such as synchronization delays. Third, in parallel programs, identifying symptoms of performance losses is easy; gaining insight into their causes requires a more sophisticated approach.

To address these challenges, we developed a novel framework for measurement and analysis of OpenMP programs. Our approach supports all OpenMP features including nested parallel regions, work-sharing constructs, synchronization constructs, and OpenMP tasks. We have developed two novel methods to measure and analyze the performance of OpenMP programs. First, we developed a measurement methodology that attributes blame for work and inefficiency back to program contexts. We show how to adapt prior work on measurement methodologies (blame shifting, see section 3.1) to OpenMP programs. We extended these methods to support dynamic thread-level parallelism in both time-shared and dedicated environments. Furthermore, we describe a more space-efficient mechanism for handling locks. Second, we developed a novel deferred context resolution method that supports on-the-fly attribution of performance metrics to full calling contexts within an OpenMP program execution. This approach enables us to collect compact call path profiles for OpenMP program executions without the need for traces.

We demonstrate the utility of our tool in case studies of four well-known application benchmarks using a version of the GNU OpenMP (GOMP) implementation, augmented with lightweight instrumentation and an API to support our measurement approach. Using insights provided by our tool, we were able to significantly improve the performance of these codes.

The next section of the chapter surveys major OpenMP profiling tools and distinguishes their various approaches from our approach. Section 4.1 describes how we attribute blame for performance losses in OpenMP programs. Section 4.1 also
explains our online deferred context resolution method for attributing performance metrics to full calling contexts in OpenMP programs. Section 4.2 discusses implementation issues for our tool. Section 4.3 demonstrates the effectiveness of our solution by studying four OpenMP benchmarks and identifying nontrivial improvement opportunities. Section 4.4 summarizes our conclusions of this chapter and describes our ongoing work.

4.1 Approach

Providing insight into the performance of OpenMP programs requires careful design and implementation of mechanisms for measuring and attributing execution costs. To achieve low overhead, we use a measurement approach principally based on asynchronous sampling. The efficacy of sampling-based profiling for analyzing program performance is well known, e.g., [133]. A sampling-based performance tool can collect a flat profile that identifies source lines and routines where an OpenMP program spends its time without any special support from an OpenMP runtime system. Such an approach can identify symptoms of inefficiencies, such as spin waiting at barriers or for locks. However, providing insight into causes of inefficiencies in OpenMP programs requires designing tool measurement capabilities for this purpose and assistance from OpenMP runtime systems. We designed and prototyped lightweight instrumentation for OpenMP runtime systems that enables our tool to attribute idleness to causes. To make our approach attractive as a potential standard OpenMP tools API, important goals of our work were to minimize the runtime overhead of our instrumentation and developer effort needed to add it to any OpenMP runtime.

The following two sections elaborate our approach. Section 4.1.1 discusses how to attribute performance metrics to their causes for OpenMP programs. Section 4.1.2
describes how to attribute metrics to full calling contexts on-the-fly for OpenMP programs.

4.1.1 Attributing Idleness and Lock Waiting

Our tool measures and attributes an OpenMP thread’s execution time into four categories:

- *Idleness*: The time a thread is either spin waiting or sleep waiting at a barrier for work. For example, OpenMP threads waiting at a barrier inside or outside a parallel region are idle. If a thread is not idle, the thread is busy.

- *Work*: The time a busy thread spends executing application code, including both serial and parallel regions.

- *Overhead*: The time a busy thread spends executing code in the OpenMP runtime system.

- *Lock waiting*: The time a thread spends spin waiting for locks.

These four metrics reflect different aspects of an OpenMP program execution. Depending on which metrics are prominent, different tuning strategies apply. If idleness dominates work attributed to a code region, that code region is insufficiently parallelized. If lock waiting associated with a code region is high, that code is associated with significant lock contention and the use of mutual exclusion in that code region needs review. If overhead for a code region is high but idleness is low, increasing the granularity of parallelism may reduce the overhead. If overhead is low and idleness is high, then decreasing the granularity of parallelism may reduce the idleness. Finally, if the overhead and idleness for a region are both high, then changing the granularity
of parallelism will not help performance and the parallelization of the code region merits review.

To collect these four metrics, we modified a version of the GNU OpenMP runtime to explicitly maintain a state variable for each thread. When a thread takes a sample, it can query the runtime to determine which metric should be adjusted. It is worth of noting that the sum of these four metrics represents the aggregate execution time across all threads.

Finally, we use different techniques to attribute these metrics to their causes. Since work and overhead belong to the threads performing them, we attribute both metrics to a thread receiving a sample. For idleness, we apply an undirected blaming technique to attribute it to busy threads, highlighting the fact that threads suffering from excessive work are the causes of idleness. For lock waiting, we apply a directed blaming technique to attribute it to lock holders to identify problematic locks.

The rest of this section discusses the challenge and our solution for the undirected and directed blaming technique in OpenMP programs respectively, and describes how we integrate undirected and directed blaming to attribute idleness and lock waiting in OpenMP programs.

**Undirected Blaming for Idleness**

As discussed in Section 3.1, previous work by Tallent and Mellor-Crummey [132] considers how to blame idleness among threads in a static thread pool. Their approach is insufficient for OpenMP, because OpenMP allows the number of worker threads to change dynamically during execution. In contrast, our undirected blaming technique supports attribution of idleness for OpenMP programs in the presence of dynamic thread-level parallelism in both time-shared and dedicated environments.
Equation 4.1 shows how we compute idleness for a program context in the presence of dynamic parallelism.

\[ I_c = \sum_{k=1}^{s_c} I_{c,k} = \sum_{k=1}^{s_c} \frac{i_{c,k}}{b_{c,k}} = p \sum_{k=1}^{s_c} \left( \frac{i_{c,k}}{b_{c,k}} + 1 \right) = p \sum_{k=1}^{s_c} \left( \frac{t_{c,k}}{b_{c,k}} - 1 \right) \]  

(4.1)

\( I_c \) is the total idleness attributed to a specific context \( c \) in the program over time. \( I_c \) is measured in processor cycles or microseconds and represents the aggregate idleness across all threads while one or more threads executed code in context \( c \). \( I_{c,k} \) is the idleness attributed to context \( c \) for the \( k^{th} \) sample. \( p \) is the sampling period, which is constant during program execution. \( s_c \) is the total number of samples taken in the context \( c \). \( i_{c,k} \) is the number of idle threads in the system when sample \( k \) occurs in context \( c \). \( b_{c,k} \) is the number of busy threads in the system when sample \( k \) occurs in context \( c \). Like Tallent and Mellor-Crummey [132], we apportion blame for instantaneous idleness in the system among the busy threads at the time. \( t_{c,k} \), shown as the sum of \( i_{c,k} \) and \( b_{c,k} \), is the total number of threads considered by our undirected blaming technique in the context \( c \) when sample \( k \) is taken. Equation 4.1 shows how we transform our idleness method to compute total idleness for a context by replacing the need for \( i_{c,k} \), a measure of instantaneous idleness, with \( t_{c,k} \), a measure of instantaneous thread-level parallelism. After the transformation, there is no need for our tool to maintain a shared variable reflecting the number of idle threads in the system.

Note, however, that choosing an appropriate value for \( t_{c,k} \) depends on the environment. To see why, consider the appropriate action concerning sequential code occurring before the first OpenMP parallel region. In a time-shared environment, an OpenMP program should not be charged for idleness outside a parallel region.
contrast, in a dedicated environment, sequential code should be charged for the idleness of allocated cores even if threads have not yet been allocated to run on them. In the text that follows, we elaborate the details of how our tool automatically chooses the appropriate value of $t_{c,k}$, depending on the environment.

For a time-shared environment, we should compute the idleness incurred by OpenMP threads required by each code region. If a code region in an OpenMP program does not use all hardware threads, these threads are available to other co-running programs and the OpenMP program should not be charged for unused threads. Thus, a sequential code region of an OpenMP program is not charged for idleness because it requires only one thread and keeps it busy. To handle this case, in a time-shared environment, we let $t_{c,k}$ to be the number of threads in use when sample $k$ is received in context $c$. In this case, $t_{c,k}$ is a variable of $c$ and $k$. Fortunately, OpenMP provides an API to query the number of threads in use at any time. Thus, we can compute $I_{c,k}$ eagerly at each sample $k$ and accumulate it to $I_c$.

For a dedicated environment, we propose two ways to choose $t_{c,k}$ values to compute two different idleness metrics to reflect inefficiency with respect to different viewpoints. To compute core idleness, we set $t_{c,k}$ to be the total number of cores* in a dedicated system. This setting blames working threads in an OpenMP program if a core in the system is unused. For this case, the value of $t_{c,k}$ is constant during execution and can be read from system configuration information. Alternatively, to compute thread idleness, we want $t_{c,k}$ to be the maximum number of threads ever in use during a program’s execution. This idleness metric is useful when assessing the scalability of an OpenMP program on different numbers of threads. In this case, the maximum number of threads used by a program will be unknown until the pro-

*Alternatively, one could do this for SMT hardware thread contexts.
ogram terminates because an OpenMP program can dynamically adjust its thread count using an API. To handle this case, we add two callbacks to the OpenMP runtime system so our tool is notified when a thread is created or destroyed. In these callbacks, our tool maintains an instantaneous global thread count using atomic increment/decrement. During execution, our tool records the maximum instantaneous thread count observed. When computing thread idleness, $t_{c,k}$ represents the maximum instantaneous thread count during execution. Since our tool won’t know its value until the end of an execution, we separately accumulate two terms in Equation 4.1, $p \sum_k \frac{1}{b_{c,k}}$ and $-p s_c$, for each context. At the end of an execution, we compute both core and thread idleness metrics for each context $c$ according to Equation 4.1 by simply multiplying $\sum_k \frac{1}{b_{c,k}}$ through by the appropriate value of $t_{c,k}$, which is constant for all $k$ samples in both cases, and adding it to the second term, which is proportional to the number of samples in $c$. Both of the aforementioned methods attribute idleness to sequential code regions in OpenMP programs, but relative to different baselines. In our case studies, we use the thread idleness metric to identify performance bottlenecks for benchmarks running in a dedicated environment.

**Directed Blaming for Lock Waiting**

Tallent at el. [135] developed a version of directed blaming for pthread programs, described in Section 3.1. Their method, which uses function wrapping to override pthread locking primitives, is not applicable to OpenMP runtime systems for three reasons. First, OpenMP runtimes have several different mutual exclusion implementations: locks, critical, ordered, and atomic sections. There is no defined interface that a tool can wrap. Second, efficiently wrapping a locking primitive requires intimate knowledge about the representation it uses. However, OpenMP runtime vendors
are unwilling to commit to a fixed lock representation—they want the freedom to use whatever representation they deem best for a particular architecture because the choice of lock representation may affect a program’s speed and scalability. Third, the approach in [135] associates extra state with each live lock to store blame information. Here, we describe a new approach for directed blaming for OpenMP programs that efficiently supports directed blaming for all mutual exclusion constructs in an OpenMP runtime system. Furthermore, it requires only modest additional space for managing blame; namely, proportional to the maximum number of concurrently contended locks rather than the maximum number of live locks.

Our directed blaming technique works in three parts: lock acquisition, asynchronous samples, and lock release. To minimize the overhead of lock acquisition, we do not require that any code be added to the critical path of an uncontended lock acquisition in an OpenMP runtime. Only if a lock acquisition fails and a thread is forced to wait does our tool record the lock we are trying to acquire in a thread-local variable. In this case, recording the lock will have little or no effect on execution time since this operation is done only when a thread begins to wait.

When a thread receives an asynchronous sample, our performance tool’s signal handler queries the thread state maintained in the OpenMP runtime. Besides returning the thread state, if the thread is currently in the lockwait state, the state query API we defined for the OpenMP runtime will also return the address of the lock that the thread is awaiting. If the thread is in the lockwait state, the signal handler uses the lock address as the key for a hash table. Our tool uses a hash table of fixed, modest size, e.g. 1024 entries, to maintain a map of blame that must be charged to the thread holding any particular lock.† The hash table is initialized when our tool is

†The idea of only maintaining blame for actively contended locks in a hash table is due to Alexan-
first attached to the program. When the signal handler tries to accumulate blame for a lock in the hash table, it allocates an entry for the lock in the table on demand if one doesn’t already exist. Hash table entries are tagged with the lock address. If two held locks hash to the same entry, blame for one will be discarded. When monitoring is enabled and a lock release callback is registered with the OpenMP runtime, then every lock release will invoke the callback with the lock address as an argument. At this point, we look up the lock in the hash table, accept any blame that has accumulated in the table for the lock, and reset the table entry’s state to free so that it is available to be reallocated on demand. If the blame accumulated in the table is greater than 0, then our tool unwinds the call stack and attributes the accumulated blame for lock waiting to the calling context of the lock release.

When designing the protocol for concurrently maintaining the hash table to accumulate blame, we considered two types of implementations: a heavyweight algorithm that carefully handles any data race that might arise when two locks hash to the same table entry, and a lighter weight implementation that doesn’t use atomic operations to manage data races. The heavyweight approach uses atomic operations and has higher runtime overhead. The lighter weight method might cause a lock to receive additional blame or some blame to be lost. The odds of this happening in practice are extremely small. As a result, we opted for the lighter weight protocol. It works exceptionally well in practice because (a) the number of locks actively contended at the same time is much smaller than the size of hash table so collisions are rare, and (b) only locks with very large sample counts charged to them are worthy of attention by an analyst. While data races can change sample counts slightly, infrequent errors of this sort are extremely unlikely to change an analyst’s assessment as to whether a

dre Eichenberger (IBM T.J. Watson Research Center).
particular lock is a problem or not.

**Integrating Directed and Undirected Blaming**

When an OpenMP program has threads in both idle and lockwait states at the same time, it suffers performance losses from both thread idleness and lock contention. To handle this case properly, directed and undirected blaming techniques need to coordinate. The *lockwait* metric needs no special care because we can always blame it directly to the context of lock holders. However, the attribution of *idleness* in such circumstances deserves careful attention.

Consider an OpenMP program execution at a point in time when there are \( n_i \) threads idle, \( n_w \) threads working, and \( n_l \) thread waiting for locks. There are three reasonable ways to consider attributing blame for *idleness* among these threads. First, we consider using undirected blaming to only charge idleness of the \( n_i \) idle threads to the \( n_w \) working threads. Any working thread that receives an asynchronous sample in this execution state charges itself for \( \frac{n_i}{n_w} \) units of idleness. This blaming scheme highlights code regions being executed by working threads as contributing to the idleness of other threads, but it ignores threads holding locks as deserving extra attention. The second method uses undirected blaming to charge an equal fraction of idleness to each thread that is working or waiting for a lock. This method attributes \( \frac{n_i}{n_w+n_l} \) idleness to the context that receives an asynchronous sample in a working or waiting thread. This scheme charges idleness not only to the context of each working thread that receives an asynchronous sample, but also to the context of any thread waiting for a lock. The rationale of this scheme is that an idle thread is waiting for all working and waiting threads. Note, however, that each thread waiting for a lock is waiting for a lock holder. Our third method uses undirected blaming technique
to charge idleness to threads both in working and lock waiting state, and then uses directed blaming to charge idleness inherited by a thread waiting for a lock to its lock holder. For example, if all $n_l$ waiting threads are spinning at the same lock, this method attributes $\frac{n_l}{n_w+n_l}$ idleness to each context that receives an asynchronous sample in a working thread, $\frac{n_i n_l}{n_w+n_l}$ idleness to the lock release, and no idleness to the context in each waiting thread. We choose the third of these methods because it transfers any blame for idleness received while waiting for a lock directly to the lock holder.

4.1.2 Deferred Context Resolution

Attributing metrics to full calling context is essential for understanding the performance of parallel programs and the underlying causes of inefficiencies. Consequently, interpreting the runtime stack for both master and worker threads in an OpenMP program is paramount. Most OpenMP runtimes manage master and worker threads separately. In fact, to our knowledge, only PGI’s OpenMP runtime uses a cactus stack to readily provide both master and worker threads with a unified call stack view.

To illustrate the problem tools face interpreting call stacks of OpenMP threads, consider Figure 4.1, which shows the call stacks of several threads in the presence of nested parallelism. When a worker thread in an outer parallel region encounters nested parallelism, it becomes a master thread for the inner region. Threads other than the master thread lack full calling context for computations they perform. Assembling the complete user-level calling context for work in a parallel region requires information about the calling context for the region itself, as well as that of any enclosing regions. Thus, to assemble the full calling context for region 3 being executed
by the level 3 worker, the level 3 worker requires the location in region 3 from its own stack, the context in which region 3 was invoked by the level 2 sub-master thread, the context in which region 2 was invoked by the level 1 sub-master thread, and finally the context in which region 1 was invoked by the master thread.

A straightforward way to support assembly of distributed context information involves the following steps when encountering a parallel region:

1. Create a unique ID for the parallel region instance.

2. If the master thread encounters a parallel region, unwind its call stack to capture
the region’s creation context and store the resulting context in a region table where it can be looked up by region ID.

3. If a sub-master thread or a worker encounters a new parallel region, unwind its call stack to the parallel region at the base of its stack, look up the calling context for that parallel region in the region table, assemble the full calling context of the new region instance, and register it in the region table.

When a thread receives an asynchronous sample as the program runs, the thread merely needs to unwind its own call stack, determine the parallel region at the base of its stack, look up the prefix of its calling context in the region table, and assemble the final result. To support this approach, the OpenMP runtime system needs to supply a callback at the entry of each parallel region instance and an API that the tool can use to query a parallel region instance ID.

Capturing the context using the outlined approach, however, is more expensive than it needs to be. The overhead of this approach can be reduced in some cases. If a parallel region is small and worker threads in the region run quickly enough so that no asynchronous samples are taken while they are executing work in the region, then there is no need to recover the full calling context for the region. To avoid the cost of context capture when no worker thread has been sampled in the region, Itzkowitz [71] proposed a deferred context resolution technique that is used in Oracle Solaris Studio [112].

Rather than capturing the calling context context for a new a parallel region instance as it is entered, Oracle Solaris Studio defers capturing context until the end of the region instance. They maintain a global map that indicates whether a thread received an asynchronous sample within a parallel region instance. If not, then one
can skip capturing the region’s context. While the deferred context resolution can dramatically reduce measurement overhead for small, but frequently called parallel regions, it introduces a new problem: a thread can no longer immediately assemble the full context for its work when it receives an asynchronous sample. The context for any enclosing parallel regions will only become available after the regions exit. In Oracle Solaris Studio, Itzkowitz et al. cope with this by recording available partial context information for each sample in a trace. Then, in a post mortem phase, full context is reconstructed using context for enclosing parallel regions that later becomes available. Supporting deferred context resolution places an additional constraint on an OpenMP runtime system—a callback from region exit is required.

To avoid the need for large traces and post-mortem context assembly when using deferred context resolution, we devised an approach for online deferred context resolution. Whenever the master thread receives an asynchronous sample, it unwinds its call stack and enters its call path into the reference calling context tree (CCT). Whenever an OpenMP worker thread receives an asynchronous sample, the thread queries its parallel region ID. Its full calling context won’t be known at this point because the context of the enclosing parallel region won’t be known until after the region exits. Let us denote the call stack suffix the worker thread recovers at this point as $s$. Since the calling context for the enclosing parallel region is not yet known, $s$ will have a placeholder at its base associated with its enclosing parallel region instance ID $r_1$. The thread next records $s$ in a thread-private CCT rooted at $r_1$. This CCT is stored in a thread CCT map indexed by $r_1$. When the thread receives a subsequent sample, it again unwinds its call stack. If it is still in the parallel region instance with ID $r_1$, it adds its new call stack suffix to its calling context tree rooted at $r_1$ in the thread CCT map.
When a master or sub master exits a parallel region $r$, it checks the *region instance sample map* to see if a sample was processed by any worker or sub-master thread in that parallel region instance. If so, the thread unwinds its call stack to acquire the calling context $s_2$ of the parallel region instance. $s_2$ is entered in the *resolved region map* indexed by the parallel region ID $r$.

When a worker or sub-master thread receives an asynchronous sample, if it finds itself in a parallel region different than the one for the previous sample, it inspects the *thread CCT map* to see if any context information for its CCTs is available from the *resolved region map*. If so, it prepends context information from the *resolved region map* to its relevant CCTs. Any CCT whose context is now fully resolved is taken out of the *thread CCT map* and placed into the reference CCT. When a thread exits or a process terminates, all remaining deferred contexts are resolved and any remaining information in the *thread CCT map* is folded into the reference CCT.

We compare our online method for deferred context resolution with the offline method used by Oracle Solaris Studio using the LULESH application benchmark, which is one of our case studies described in Section 4.3.2. For a 48-thread execution of LULESH using the same program input and sampling rate, Oracle Solaris Studio records a 170MB performance data file to perform deferred context resolution post-mortem. Using our online deferred context resolution strategy, our tool records a compact profile for each thread, 8MB total—a reduction in data volume of more than 20×.

Like Oracle Solaris Studio, our prototype tool distinguishes OpenMP runtime frames on the call stack from others by simply identifying that they belong to the shared library for the OpenMP runtime. For the OpenMP standard, a solution suitable for statically-linked programs is needed as well. We are part of the OpenMP
standards committee working group designing a new tools API that provides the necessary support. The new API has the OpenMP runtime record stack addresses for procedure frames that enter and exit the OpenMP runtime. When one uses call stack unwinding, this interface will enable one to identify sequences of procedure frames on the stack that belong to the OpenMP runtime rather than user code, even for statically linked programs. We are in the process of evaluating this design to provide feedback to the OpenMP tools API subcommittee.

A note about context resolution for OpenMP tasks

OpenMP 3.0 tasks [9] support work stealing execution. With respect to context resolution, this introduces an additional consideration. Since task creation may be separated from task execution, tasks have an additional component to their context—the creation context.

To record the creation context of a task, we add an 8-byte field in the OpenMP task data structure maintained by the runtime system. Once a non-immediate-run task is created, a callback function is invoked to pass the 8-byte field to the performance tool. The performance tool unwinds the call stack to collect the task creation context in the callback function invoked at each task creation point. It then updates the 8-byte field in the task structure with the task creation context. When a task is executed, the performance tool uses two query APIs to get the task creation context and the root frame of the executing task. We combine the two to get the full context of the task. One complicated but practical case is that one task can be nested in another task. We can concatenate the outer task creation context with the inner task creation context to resolve the full context of the inner task. However, if the program recursively creates nested tasks, the call site of inner-most tasks may be deep in the call stack. Long recursive call paths aren’t typically of much interest to a performance
analyst. For that reason, we collapse the call stack for recursive tasks. If the root frame of one task is the same as that of its parent, we fold the performance data of the task into its parent and skip its frames on the call stack.

Incorporating the creation context into the complete context for every OpenMP task is expensive. The overhead of eagerly gathering this information cannot be tied to the sampling rate, and therefore it is uncontrolled. Consequently, we do not employ creation-extended task context construction by default. Rather than providing a full creation context for a task by default, we simply associate the task with its enclosing parallel region. If a performance analyst needs the full creation context for tasks, our tool will collect it upon request, though this extra information increases monitoring overhead. Our case studies in Section 4.3 did not require full creation contexts for tasks to provide performance insights.

4.2 Tool Implementation

To evaluate our ideas for measurement and analysis of OpenMP program executions, we implemented them in the HPCToolkit [2] performance tools, which are introduced in Section 2.3. We extended HPCToolkit’s existing call path profiler to support the techniques described in the previous section. Using these techniques, our extended HPCToolkit is able to analyze the performance of OpenMP and OpenMP+MPI programs by both collecting measures of work, idleness, overhead, and lock waiting; and then attributing these measures to full calling contexts in a unified user-level view. To analyze performance data, we use HPCToolkit’s hpcviewer graphical user interface. hpcviewer associates both measured and derived metrics to full calling contexts, thereby representing a unified view reconstructed using our online deferred context resolution strategy.
The implementation of our techniques, however, requires support from the OpenMP runtime system. Consequently, we took special care to minimize the required support. Our initial prototype was based on an open-source OpenMP implementation — GNU OpenMP (GOMP) [56]. To support both our blaming strategies and online deferred context resolution, we modified GOMP source code to insert necessary callbacks, query functions, and data structures associated with parallel regions and tasks. Our modification encompassed 5 files and less than 50 lines of code. The software development cost to add the necessary OpenMP runtime support for our methods is quite low. Also, as we show in Section 4.3, the runtime impact of our proposed support is similarly low.

To quantify the improvement one can obtain after optimization, we compute two derived metrics from each raw metric we defined in Section 4.1.1. For example, we derive absolute idleness and relative idleness metrics from the raw idleness metric. Absolute idleness quantifies the idleness in a given context relative to the entire effort of the program. It shows the maximum possible (percentage) improvement if the idleness is eliminated for that context. Relative idleness reflects the parallel efficiency for the context. A high relative idleness value means the context is making poor use of parallel resources. A good rule of thumb is to focus optimization efforts on contexts with both high absolute idleness and high relative idleness.

Note that both derived metrics are computed after the raw data has been collected. The defining equations are shown in equation 4.2. The notation key in equation 4.2 is as follows: $I_{c,\text{abs}}$ is the absolute idleness computed for context $c$; $I_{c,\text{rel}}$ is the relative idleness computed for context $c$; $I_c$, $W_c$, $O_c$, and $L_c$ are idleness, work, overhead, and lock waiting respectively attributed to the context $c$; $I_r$, $W_r$, $O_r$, and $L_r$ are idleness,
work, overhead, and lock waiting respectively aggregated for the whole program.

\[
I_{c,\text{abs}} = \frac{I_c}{I_r + W_r + O_r + L_r} \times 100\%
\]

\[
I_{c,\text{rel}} = \frac{I_c}{I_c + W_c + O_c + L_c} \times 100\% \tag{4.2}
\]

It is worth noting that analogous derived metrics exist for work, overhead and lock waiting as well.

4.3 Case Studies

In this section, we evaluate the utility and overhead of our measurement approach using a set of application benchmarks that employ OpenMP parallel loops with and without nested parallel regions, OpenMP tasking, and OpenMP in conjunction with MPI. We studied four benchmarks: AMG2006 [79], LULESH [77], NAS BT-MZ [73], and HEALTH [40]. These benchmarks are described in Section 2.4.

We compiled each of these benchmarks using the GNU 4.6.2 compiler with -O3 optimization and linked with our modified version of GOMP library from GCC 4.6.2. We measured the performance of these codes on a system with four 12-core AMD Magny-Cours processors and 128 GB memory. For AMG2006, we studied executions consisting of 4 MPI processes and 8 OpenMP threads per process. For LULESH, we studied executions with 48 OpenMP threads. For NAS BT-MZ, we studied executions that used 4 threads for outer parallel region and 8 threads for inner parallel regions. For HEALTH, we studied runs on 8 OpenMP threads. We measured the performance of these codes using asynchronous sampling at a rate of 200 samples/s per thread.

We first evaluate the measurement overhead associated with our modified GOMP library and profiler. Table 4.1 compares the execution time of each benchmark code
using the native GOMP library, our modified GOMP library with performance measurement hooks, and the modified GOMP library with our profiler attached. The capabilities employed by our profiler include using undirected, directed as well as integrated blaming for *idleness, work, overhead* and *lock waiting*, and using online deferred context resolution for parallel regions, resolving task contexts to their enclosing parallel region. By comparing the times in the first and second columns of the table, we see that our GOMP modifications to support performance tools add almost no overhead. Comparing the first and third columns shows that the measurement overhead using our profiler is less than 5% for each of these codes.

Note that if we resolve the *full* creation context for tasks to their creation contexts in HEALTH, the run time of a profiled execution jumps from 74.27s to 431.48s. This is more than a 6× slowdown compared to its unmonitored execution. Resolving the full creation context for tasks in HEALTH is so expensive because HEALTH creates 17 million tiny tasks as it executes, with each task requiring a call stack unwind to recover its full creation context. To keep measurement overhead low, our profiler’s default measurement approach is to only resolve task contexts to their enclosing parallel region.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>native GOMP</th>
<th>modified GOMP</th>
<th>profiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2006</td>
<td>54.02s</td>
<td>54.10s</td>
<td>56.76s</td>
</tr>
<tr>
<td>LULESH</td>
<td>402.34s</td>
<td>402.56s</td>
<td>416.78s</td>
</tr>
<tr>
<td>NAS BT-MZ</td>
<td>32.10s</td>
<td>32.15s</td>
<td>34.23s</td>
</tr>
<tr>
<td>HEALTH</td>
<td>71.74s</td>
<td>72.20s</td>
<td>74.27s</td>
</tr>
</tbody>
</table>

Table 4.1: Running times for our case study applications with (a) the unmodified GOMP runtime library (no sampling), (b) the modified GOMP library with lightweight instrumentation to support blame shifting (no sampling), and (c) the modified GOMP library while profiling at 200 samples per second per thread. Execution times represent the average over three executions.
In the following four sections, we illustrate the effectiveness of our profiling approach by describing insights it delivered for each of our benchmark applications and how they helped us identify optimizations that yielded non-trivial improvements to these codes.

4.3.1 AMG2006

Figure 4.2 shows a time-centric view of traces of AMG2006 executing on 8 MPI ranks with 8 OpenMP threads per rank. An execution of AMG2006 consists of three phases: MPI initialization, problem setup, and solve.

During initialization, only the master thread in each MPI rank is active. The initial white space in the traces of OpenMP worker threads shows that they are idle before they are created. During the setup phase shown in Figure 4.2, the light gray color
Figure 4.3: HPCToolkit’s time-centric view rendering of a single iteration of AMG2006’s solver on 64 cores – 8 MPI ranks with 8 OpenMP threads/rank.

for OpenMP worker threads represents worker threads blocked in `pthread_cond_wait`. This view illustrates that OpenMP threads are idle most of the time; the largest component of that idleness occurs while the master thread in each MPI rank executes serial code for `hypre_BoomerAMGCoarsen`. Our code-centric view for the full execution (not shown) reports that 34.9% of the total effort in the execution corresponds to idleness while `hypre_BoomerAMGCoarsen` executes; the relative idleness our tool attributes to this routine is 87.5%—exactly what we would expect with one of 8 cores working within an MPI rank. Our tool’s quantitative measure for idleness attributed to this serial code is a result of our blame shifting approach. Neither VTune nor Oracle Solaris Studio directly associates idleness with serial code. During the setup phase, short bursts of activity by OpenMP worker threads separate long gray intervals of idleness. However, the intervals of activity by the worker threads are of
unequal lengths, which indicates that even when threads are working, their work is imbalanced. While the setup phase in this benchmark execution accounts for a large fraction of the total time, the solve phase dominates in production computations. For that reason, we focus the rest of our analysis on the performance of the solve phase. Figure 4.3 shows an expanded view of one iteration of the solve phase. The imbalanced intervals of gray indicate idleness caused by a severe load imbalance in OpenMP loops employed by `hyper_BoomerAMGRelax`.

To analyze the solver phase in more detail, we use HPCToolkit’s code-centric `hpcviewer` to analyze measurement data collected for just the solve phase of the benchmark in an execution by 4 MPI ranks with 8 OpenMP threads per rank. Figure 4.4 at-
tributes idleness, work and overhead to the complete calling context of parallel regions. Figure 4.4 shows only inclusive metric values. The 40.75% aggregate absolute idleness metric for the solve phase means that threads are working only about 60% of the time. The OpenMP runtime overhead for the solve phase is negligible—only 0.02%. Drilling down the call path from main, we find that a large part of the idleness in the solve phase is attributed to a parallel region—hyper_BoomerAMGRelax_omp_fn.23, which is highlighted in Figure 4.4. Work by that routine (and routines it calls) accounts for 12.42% of the idleness in the solve phase. The relative idleness measure of 34.15% for this program context indicates that this parallel region is active, roughly one third of the threads are idle. The discussion in Section 4.1.1 indicates that if overhead is low and idleness is high, one should reduce the granularity of parallel work to increase parallelism and improve load balance.

To apply this optimization, we examined the source code corresponding to hyper_BoomerAMGRelax_omp_fn.23 in the source pane of Figure 4.4. We found that this parallel region decomposes the computation into one piece of work for each thread and each thread is statically assigned a piece of work. Since each piece of work computes an equal number of iterations in the parallel loop, it would appear that each thread has equal amount of work. However, the execution time of iterations differs significantly due to characteristics of the input dataset, resulting in load imbalance. To reduce the load imbalance that causes the high idleness in this parallel region, we decomposed the work into a number of smaller pieces and used dynamic scheduling to balance the work among threads. Decomposing the work into five chunks per thread provided enough flexibility for dynamic scheduling to reduce load imbalance without adding too much overhead for the scheduling itself. The optimization reduced the idleness of hyper_BoomerAMGRelax_omp_fn.23 from 3.05e+11 to 4.53e+09 CPU
Figure 4.5: A bottom-up view of call path profiles for LULESH. `madvise`, which is called while freeing memory, is identified as the routine with the most idleness.

cycles, reducing the running time of the solve phase by 11%.

4.3.2 LULESH

In Figure 4.5, we analyze measurements from a 48-thread execution of LULESH using a bottom-up view which attributes metric consumption to routines and their calling contexts. We sorted the results to identify routines with the highest exclusive absolute idleness. The top routine is the `madvise` system call, which is called by `free` from both the application and the OpenMP runtime. The idleness associated with `free` amounts to 4.6% of the total thread execution time. The percentage of relative idleness indicates that only the master thread calls `free` and the other 47
for( ... ) {
    malloc a[n], b[n], c[n];
    #pragma parallel for
    for(i=0; i<n; i++) {
        a[i] = ...
        b[i] = ...
        c[i] = ...
    }
    free a, b, c;
}

Figure 4.6 : Pseudo code showing how memory is allocated, initialized, and freed in LULESH.

threads are idle. It is worth noting that the performance tool reports that free accounts for only 0.1% of total execution time. Without idleness blame shifting, it would not appear that memory management is a potential performance problem. To see if we could improve performance by using a better memory allocator, we used Google’s TCMalloc [57] (a high-performance threaded allocator) instead of glibc’s implementations of malloc and free. Surprisingly, the execution time drops from 402s to 102s—a 75% improvement!

To understand the improvement, which was well beyond our expectations, we compared profiles of the original and optimized versions of LULESH. As we expected, the absolute idleness associated with free drops from 4.6% to less than 0.1%. However, we also found that the work associated with some parallel regions shrank by a factor of 20. Further examination showed that data is allocated immediately before entering these regions on each iteration of an enclosing loop. Pseudo code in Figure 4.6 shows the structure of these problematic parallel regions. Because glibc’s free releases memory pages to the operating system, a subsequent malloc causes pages to be added back to the application’s address space. As threads attempt to write these
newly-allocated pages in the parallel region, they fault and stall as the operating system lazily zero-fills the pages. Repeatedly freeing, reallocating, and zero-filling pages is surprisingly costly. Unlike glibc, TCMalloc does not return deallocated pages to the operating system so it avoids the cost of repeated zero-filling, leading to a 75% speedup.

While our idleness measure didn’t help us predict the full benefit of changing the memory allocator, it did uncover the “tip of the iceberg,” which helped us improve the performance of LULESH. The high overhead of zero-filling pages that we missed with our measurements was masquerading as work since it was incurred in parallel regions. An analysis based on sampling measurements of real time and graduated instructions might have provided some additional insight into operating system overhead that was unobserved.

### 4.3.3 NAS BT-MZ

To evaluate HPCToolkit’s support for measurement and analysis of OpenMP programs with nested parallel regions, we used it to study the performance of the NAS BT-MZ benchmark. Figure 4.7 shows the unified calling contexts that HPCToolkit recovers for nested parallel regions in BT-MZ. In the figure, the parallel region `MAIN_omp_fn.3` is nested inside `MAIN_omp_fn.0`, showing that our method properly reconstructs calling contexts for nested parallel regions. The aggregate *absolute idleness* is larger than *absolute work*, which means that threads are idle more than half of the time during execution. Examination of the inclusive idleness measurements for the calling context of the nested parallel regions shows that most of the idleness comes from the innermost region. To optimize BT-MZ, we refactored the `exch_qbc` routine highlighted in Figure 4.7 because it has 1.55% *absolute work* but
11.1% absolute idleness. The relative idleness for this routine is more than 87%.

Looking deeper along the call path inside the innermost parallel region, we see that `pthread_create` is responsible for idleness that amounts to roughly 3% of the overall execution cost. This cost is incurred because GOMP does not use a persistent thread pool for threads in nested parallel regions. Threads used in nested parallel regions are created at region entry and destroyed at region exit. For a nested parallel region in a loop, frequent thread creation and destruction can degrade performance.

Changing the inner parallel region inside `exch_qbc` to a sequential region eliminated the idleness caused by thread creation and destruction, improving performance by 8%.
4.3.4 HEALTH

To evaluate HPCToolkit’s support for measurement and analysis of programs based on OpenMP tasking, we studied the HEALTH benchmark. In this execution, we used HPCToolkit’s (low-overhead) default context resolution for tasks, which attributes tasks back only to their enclosing parallel region rather than their full creation context. In experiments with HEALTH on eight threads, we found that its absolute idleness was much less than 1%, indicating that each thread was working on a parallel task most of the time. However, severe lock contention in the benchmark caused threads to spend much of their time spin waiting. Figure 4.8 shows a bottom-up view of the HEALTH benchmark showing lock contention in the GOMP library.
the lock waiting metrics for an execution of HEALTH. The absolute lock wait metric shows threads in HEALTH spent roughly 75% of their total execution time waiting for locks. As described in Section 4.1.1, blame for lock waiting gets attributed to the context where the lock holder releases the lock. Figure 4.8 shows blame being attributed up call paths from lock release to contexts where the lock was released.

The highly contended locks are released on line 201 of GOMP_task and line 348 of GOMP_taskwait. Locks released at these points account for significant waiting—roughly 41% and 33% of the thread execution time respectively. The OpenMP task sim_village_par._omp_fn_2 is the common caller of GOMP_task and GOMP_taskwait, where lock contention leads to significant waiting. The lock release highlighted in the source code pane shows the problematic lock—task_lock. Note that task_lock is used in the GOMP library, not the user’s code. task_lock is the lock used to manipulate the task queue created in a parallel region. It causes excessive contention in sim_village_par._omp_fn_2, as this routine recursively spawns 17 million tiny tasks and all threads contend for the lock to access the task queue. The task_lock used by sim_village_main.par._omp_fn_1 causes little contention, so there is no need to optimize this parallel region. The best way to reduce contention for the task queue inside sim_village_par._omp_fn_2 is to reduce the number of tasks by increasing task granularity. Another version of the program achieves this objective by using a cutoff to stop spawning tasks once granularity falls below a threshold. This improvement reduced program execution time by 82%, which is consistent with what our lock waiting measurements predicted.
Table 4.2: Summary of performance bottlenecks identified using our tool and our code optimizations in response.

<table>
<thead>
<tr>
<th>app</th>
<th>problem</th>
<th>optimization</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2006</td>
<td>high idleness in parallel regions</td>
<td>use dynamic scheduling to eliminate load imbalance</td>
<td>11%</td>
</tr>
<tr>
<td>LULESH</td>
<td>high idleness caused by the OS frequently zero filling allocated pages</td>
<td>use Google’s tcmalloc threaded allocator to reduce page zero filling</td>
<td>75%</td>
</tr>
<tr>
<td>BT-MZ</td>
<td>high idleness caused by frequent thread creation and exit in nested parallel regions</td>
<td>eliminate unnecessary inner parallel regions</td>
<td>8%</td>
</tr>
<tr>
<td>HEALTH</td>
<td>high lock contention for the task queue in OpenMP runtime</td>
<td>coarsen the task granularity</td>
<td>82%</td>
</tr>
</tbody>
</table>

4.4 Discussion

This chapter proposes mechanisms to support low-overhead, sampling-based performance analysis of OpenMP programs. We demonstrate that an implementation of these techniques in HPCToolkit provides deep insight into the performance of threaded program executions by measuring and attributing informative metrics including idleness, work, overhead, and lock waiting. Our OpenMP profiler employs online deferred context resolution to efficiently and accurately attribute these metrics to full calling contexts in compact profiles, avoiding the space overhead of traces required by prior tools. Reducing the space overhead is an important aspect of our strategy that will enable it to scale to large parallel systems. The case studies reported in this chapter validate the effectiveness and low overhead of our approach. Table 4.2 indicates the problems we identified using our tool in each of the application benchmarks, summarizes the optimizations we applied to improve each program, and shows the percent improvement that we achieved.
We are presently working with the OpenMP standards committee to define a standard tools API for OpenMP — OMPT. OMPT provides support for our blame shifting approach. This includes mandatory interfaces that enable us to attribute performance metrics to full calling contexts as well as optional interfaces to pinpoint and quantify root causes of thread idleness and lock waiting. Moreover, OMPT provides support sufficient for analysis of both statically and dynamically linked applications. Currently, OMPT is accepted by OpenMP language committee as an official technical report [46] and will be sent to OpenMP language committee for review. IBM has added support for a draft of OMPT to a new lightweight OpenMP runtime system for their XL compilers for Blue Gene/Q. Mellor-Crummey added initial support of OMPT to Intel’s OpenMP runtime. An implementation of HPCToolkit employs this draft interface to measure OpenMP programs executed by these runtimes.

In the future, we plan to generalize our approach for deferred context resolution to support assembly of global view call path profiles on a heterogeneous system that offloads computation from a host node populated with conventional multicore chips to attached manycore accelerators, such as Intel Xeon Phi chips.
Chapter 5

Analyzing Memory Bottlenecks with Lightweight Measurement

In modern processors, accesses to deep layers of the memory hierarchy incur high latency. Memory accesses with high latency not only degrade performance but also increase energy consumption. For many programs, one can reduce average memory latency by staging data into caches and accessing it thoroughly before it is evicted. Access patterns that do so are said to have excellent data locality. There are two types of data locality common to both sequential and multithreaded programs: spatial and temporal. An access pattern exploits spatial locality when it accesses a memory location and then accesses nearby locations soon afterward. Typically, spatial locality goes unexploited when accessing data with a large stride or indirection. An access pattern exhibits temporal locality when it accesses individual memory locations multiple times.

Multi-socket systems have an extra layer in the memory hierarchy that poses an additional obstacle to performance. Processors in such systems are connected with a high bandwidth link, e.g., HyperTransport for AMD processors and QuickPath for Intel processors. To provide high aggregate memory bandwidth on multi-socket systems, memory is partitioned and some is directly attached to each processor. Each processor has its own memory controller to access directly-attached memory. Such an architecture has Non-Uniform Memory Access (NUMA) latency. Accesses to directly-attached memory are known as local accesses. A processor can also access memory
attached to other processors; such accesses are known as remote accesses. Remote accesses have higher latency than local accesses, so each thread in a multi-threaded program must access local data for high performance. A thread that primarily performs local accesses is said to have NUMA locality.

Tools for identifying data locality bottlenecks use either simulation or measurement. Simulation-based data-centric tools, such as CPROF [81], MemSpy [95], ThreadSpotter [115], SLO [16, 15], and MACPO [113], instrument some or all memory accesses and compute the approximate memory hierarchy response with a cache simulator. There are two drawbacks to simulation. First, gathering information about memory accesses with pervasive instrumentation is expensive. Although sampling techniques for monitoring accesses, e.g., [117, 143], can reduce measurement overhead for instrumentation, they do so with some loss of accuracy. Second, the accuracy of simulation-based methods depends on the cache simulators they use. Since it is prohibitively expensive for cache simulators to model all details of memory hierarchies in modern multi-socket systems, the aforementioned tools make simplifying assumptions for simulators that reduce the tools’ accuracy.

Unlike simulation-based tools, measurement-based tools collect performance metrics using hardware performance counters. Measurement-based tools can directly assess the performance of a program execution with low overhead. Measurement-based tools can be classified as code-centric and/or data-centric. In the best case, code-centric tools such as VTune [66], Oprofile [82], CodeAnalyst [4], and gprof [58] map performance metrics back to statements accessing data. Such code-centric information is useful for identifying problematic code sections but fails to highlight problematic variables. Data-centric tools, on the other hand, attribute performance metrics to variables or dynamic memory allocations. A tool that combines both data-
centric and code-centric analysis is more powerful for pinpointing and diagnosing data locality problems. Section 5.1 elaborates on the motivation for data-centric tools.

A variety of data-centric tools currently exist; we have discussed them in detail in Section 3.2. Some tools focus on data locality in sequential codes [85, 87, 24]; the others focus on NUMA problems in threaded codes [96, 75]; none of them supports comprehensive analysis of all kinds of data locality problems. Moreover, existing tools work on modest numbers of cores on a single node system; none of them tackles the challenge of scaling and is applicable across a cluster with many hardware threads on each node. Obviously, data-centric measurement tools must be scalable if they are to be used to study codes on modern supercomputers with non-trivial input data.

To address this challenge, we extended the HPCToolkit performance tools [2] with data-centric capabilities to measure and analyze program executions on scalable parallel systems. Our resulting tools have three unique capabilities for data-centric measurement and analysis.

- They report all data locality issues. HPCToolkit can measure and analyze memory latency in threaded programs running on multi-socket systems. Failing to exploit temporal, spatial, and NUMA data locality exacerbates memory latency.

- They work for large-scale hybrid programs that employ both MPI [98] and OpenMP [111]. HPCToolkit collects data-centric measurements for scalable parallel programs with low runtime and space overhead. Moreover, HPCToolkit aggregates measurement data across threads and processes in a scalable way.

- They provide intuitive analysis of results for optimization. HPCToolkit provides an intuitive graphical user interface that enables users to analyze data-centric metrics attributed to variables, memory accesses, and full calling contexts. It
provides multiple data-centric views to highlight variables and attribute costs to instructions that access them.

HPCToolkit exploits hardware support for data-centric measurement on both AMD Opteron and IBM POWER processors. To evaluate the effectiveness of our methods, we employ our tools to study five parallel benchmarks, covering MPI, OpenMP and hybrid programming models. With the help of HPCToolkit, we identified data locality problems in each of these benchmarks and improved their performance using insights gleaned with our tools.

The rest of this chapter is organized as follows. Section 5.1 elaborates on the motivation for this work. Section 5.2 presents the design and implementation of our new capabilities in HPCToolkit. Section 5.3 studies five well-known benchmarks and shows how HPCToolkit’s data-centric analysis capabilities provide insights about data access patterns that are useful for tuning. Finally, Section 5.4 summarizes our conclusions of this chapter and outlines our future directions.

5.1 Motivation

There are two principal motivations for extending HPCToolkit to support data-centric measurement and analysis of program executions on scalable parallel systems. In Section 5.1.1, we illustrate the importance of data-centric profiling for analyzing data locality problems. In Section 5.1.2, we describe why scalability has become an important concern for data-centric profilers.

5.1.1 Data-centric Profiling

Two capabilities distinguish data-centric profiling from code-centric profiling. First, while code-centric profiling can pinpoint a source code line that suffers from high ac-
Figure 5.1: Code-centric profiling aggregates metrics for memory accesses in the same source line; data-centric profiling decomposes metrics by variable.

cess latency, without a mapping from machine instructions to character positions on a source line, code-centric profilers can’t distinguish the contribution to latency associated with different variables accessed by the line. In contrast, data-centric profiling can decompose the latency and attribute it to individual variables accessed by the same source line. Figure 5.1 illustrates the latency decomposition that data-centric profiling can provide. Data-centric methods can decompose the latency associated with line 4 and attribute it to different variables. From the percentage of latency associated, one can see that array $C$ is of principal interest for data locality optimization. One can then continue to investigate the program to determine how to improve data locality for $C$. Second, data-centric profiling aggregates metrics from all memory accesses that are associated with the same variable. Aggregate metrics can highlight

```c
1: for (i = 0; i < n; i++) {
2:   for (j = 0; j < n; j++) {
3:     for (k = 0; k < n; k++) {
5:     }
6:   }
7: }
```
problematic variables in a program. This can help one identify when a data layout is poorly matched to access patterns that manipulate the data, or pinpoint inefficient memory allocations on a NUMA architecture.

5.1.2 Scalability

Scalability is a significant concern for data-centric profilers because (a) application performance on scalable parallel systems is of significant interest to the HPC community; (b) memory latency is a significant factor that limits performance and increases energy consumption; and (c) it is desirable to study executions on large-scale data sets of interest rather than forcing application developers to construct representative test cases that can be studied on small systems. Unfortunately, existing data-centric profilers do not scale to highly-parallel MPI+OpenMP programs.

Data-centric profilers for scalable parallel systems should have low time and space overhead. Runtime overhead mainly comes from two parts: collecting performance metrics and tracking variable allocations. One can reduce the overhead of collecting metrics by sampling hardware performance monitoring units (described in Section 2.1) with a reasonable sampling period. However, tracking variable allocations is difficult to control. If a program frequently allocates data, the overhead of tracking allocations can be unaffordable. Unfortunately, existing tools lack the capability to reduce such overhead; instead, they assume that programs don’t frequently allocate heap data. We show a benchmark in our case study that contradicts this assumption.

Space overhead is also critical when profiling parallel executions. Systems like LLNL’s Sequoia support millions of threads. If each thread in a large-scale execution generates 1MB of performance data, a million threads would produce a terabyte of performance data. Thus, for analysis at scale a compact profile is necessary to
for (i = 0; i < 100; i++) {
    var[i] = malloc(size);
}

Figure 5.2 : A loop allocating data objects in the heap.

keep the size of measurement data manageable. However, many existing tools trace variable allocations and memory accesses [70, 75]. The size of memory access traces is proportional to execution time and the number of active threads. A trace of variable allocations can easily grow unaffordably large. Consider the code in Figure 5.2. A memory allocation is called 100 times in a loop, so 100 allocations would be recorded in a thread trace. However, if this loop is in an OpenMP parallel region executed by each MPI process, millions of allocations would be recorded in a large-scale execution on a system like LLNL’s Sequoia.

Besides overhead, existing tools do not display data-centric results in a scalable way for effective analysis. Again, consider the code in Figure 5.2. Metrics may be dispersed among 100 allocations, without showing any hot spot. However, one might be interested in aggregating metrics for these 100 allocations per thread. If this loop is called by multiple threads and MPI processes, metrics associated with these data objects should be coalesced to highlight var as a problematic array.

To address these issues, we added scalable data-centric profiling support to the HPCToolkit performance tools. HPCToolkit employs novel techniques to scale the data collection and presentation with low time and space overhead.
Figure 5.3: A simplified workflow that illustrates data-centric measurement and analysis in HPCToolkit. Rectangles are components of HPCToolkit; ellipses are inputs or outputs of different components.

5.2 Data-centric Capabilities for HPCToolkit

Figure 5.3 shows a simplified workflow that illustrates data-centric measurement and analysis in HPCToolkit. HPCToolkit consists of three principal components: an online call path profiler, a post-mortem analyzer, and a graphical user interface (GUI) for presentation. The profiler takes a fully optimized binary executable as its input. For each thread in a program execution, the profiler collects samples and associates costs with memory accesses and data objects. Section 5.2.1 describes the profiler. The post-mortem analyzer, described in Section 5.2.2, gathers all profiles collected by the profiler for each process and thread. It also analyzes the binaries, extracts information about static code structure, and maps profiles to source lines, loops, procedures, dynamic calling contexts, and variable names. Finally, the GUI, also described in Section 5.2.2, displays intuitive views of data-centric analysis results that highlight problematic data and accesses. Each component is designed to scale.
5.2.1 Online Call Path Profiler

As a program executes, HPCToolkit’s profiler triggers samples, captures full calling contexts for sample events, tracks variables, and attributes samples to both code and variables. To minimize synchronization overhead during execution, each thread records its own profile. The following four sections describe the implementation of data-centric support for each of these capabilities.

Triggering Samples

The profiler first programs each core’s PMU to enable instruction-based sampling or marked event sampling with a pre-defined period. When a PMU triggers a sample event, the profiler receives a signal and reads PMU registers to extract performance metrics related to the sampled instruction. To map these performance metrics to both code and data, the profiler records the precise instruction pointer of the sampled instruction and the effective address of the sampled instruction if it accesses memory.

Capturing Full Calling Contexts

As described in Section 2.3, HPCToolkit unwinds the call stack at each sample event. To do so, it uses on-the-fly binary analysis to locate the return address for each procedure frame on the call stack [133]. Call paths are entered into a calling context tree (CCT) [5]. A CCT reduces the space needed for performance data by coalescing common call path prefixes.

Supporting data-centric analysis required two changes to HPCToolkit’s call stack unwinder. First, we adjust the leaf node of the calling context we obtain by unwinding from the signal context to use the precise IP recorded by PMU hardware. This avoids “skid” between the monitored instruction and the sample event that typically
occurs on out-of-order processors. Second, we create multiple CCTs for each thread. Inserting a sample call path into a specific CCT depends on the sample feature. For example, we create a CCT that includes all samples that do not access memory; other CCTs include call paths that touch different types of variables, such as static and heap-allocated variables.

**Tracking Variables**

To support data-centric analysis, we augmented HPCToolkit to track the life cycle of each variable. We track the allocation and deallocation of static and heap allocated data. Variables that do not belong to static or heap allocated data are treated as unknown data.

**Static data** Data allocated in the `.bss` section in load modules are static data. Each static variable has a named entry in the symbol table that identifies the memory range for the variable with an offset from the beginning of the load module. The life cycle of static variables begins when the enclosing load module (executable or dynamic library) is loaded into memory and ends when the load module is unloaded. The profiler tracks the loading and unloading of load modules. When a load module is loaded into memory, HPCToolkit reads the load module’s symbol table to extract information about the memory ranges for all of its static variables. These memory ranges are inserted into a map for future use. All load modules in use are linked in a list for easy traversal. If a load module is unloaded, the load module together with its search tree of static data is removed from the list.

HPCToolkit tracks all static variables used during a program execution. Unlike other tools, HPCToolkit not only tracks static variables in the executable, but also
static variables in dynamically-loaded shared libraries. Moreover, it collects fine-grained information for each static variable rather than simply attributing metrics to load modules.

Heap-allocated data Variables in the heap are allocated dynamically during execution by one of the \texttt{malloc} family of functions (\texttt{malloc}, \texttt{calloc}, \texttt{realloc}). Since heap-allocated data may be known by different aliases, e.g., function parameters, at different points in an execution, HPCToolkit’s profiler uses the full call path of the allocation point for a heap-allocated data block to uniquely identify it throughout its lifetime. To associate a heap-allocated variable with its allocation call path, the profiler wraps memory allocation and free operations in a monitored execution. At each monitored allocation, the profiler enters into a map an association between the address range of a data block and its allocation point. At each monitored free, the profiler deletes an association from the map.

Unknown data Stack variables do not belong to static or heap allocated data. Such variables are either not easily tracked or have little impact for performance. HPCToolkit treats stack variables as unknown data because stack variables seldom become data locality bottlenecks. In addition, HPCToolkit does not track C++ template container allocations. C++ template containers directly use a low level system call \texttt{brk} to allocate memory. Because \texttt{brk} sets the data segment instead of returning the allocated ranges, it is difficult to track in HPCToolkit.

Recording address range information for static variables incurs little overhead because it happens once when a load module is loaded into the program’s address space. However, the overhead of tracking heap allocations and deallocations hurts the scalability of the profiler. If a program allocates and frees memory with high
frequency, wrapping allocates and frees and capturing the full calling context for each allocation may cause large overhead. For example, the execution time of AMG2006, one of our case study benchmarks, increases by 150% when monitoring all allocations and frees. To reduce such overhead, we use the following three strategies.

- We do not track all memory allocations. Usually, large arrays suffer from more severe locality problems than small ones. Typically, there are also more opportunities for optimizing data locality for large arrays. For that reason, HPCToolkit does not track any heap allocated variable smaller than a threshold, which we have chosen as 4K. However, we still track all calls to free to avoid attributing costs to wrong variables. Because we don't collect calling contexts for frees, wrapping all frees is not costly.

- We use inlined assembly code to directly read execution context information from registers to aid in call stack unwinding. Our assembly code incurs lower overhead than libc’s `getcontext`.

- Since unwinding the call stack for frequent allocations in deep calling contexts is costly, we reduce unwinding overhead by identifying the common prefix for adjacent allocations to avoid duplicate unwinds of prefixes that are already known. We accomplish this by placing a marker (known as a trampoline) in the call stack to identify the least common ancestor frame of the calling contexts for two temporally adjacent allocations [51]. Using this technique, each allocation only needs to unwind the call path suffix up to the marked frame.

Because the first method can lead to inaccuracy due to the information loss, we only use it when necessary to avoid unaffordable time overhead. We always enable the other two methods because they are always beneficial. In our case study of AMG2006,
these approaches reduce the time overhead of tracking variables from 150% to less than 10%.

Attributing Metrics to Variables

By correlating information about accesses from PMU samples with memory ranges for variables, HPCToolkit performs data-centric attribution on-the-fly. It first creates three CCTs in each profile, each recording a different storage class: static, heap, and unknown. This aggregation highlights which storage class has more problematic locality.

For each sample, HPCToolkit searches the map of heap allocated variables using the effective address provided by the PMU registers. If a sample in a thread touches a heap allocated variable, the thread prepends the call path for the variable allocation to the call path for the sample and then adds it to its local heap data CCT. It is worth noting that the memory allocation call path may reside in a different thread than the one the PMU sample takes for an access. Copying a call path from another thread doesn’t require a lock because once created, a call path is immutable. If a copied call path is already in the local thread CCT, the thread coalesces the common paths. Although a memory access is mapped to a heap allocated variable using the allocated memory ranges, the allocation call path uniquely identifies a heap allocated variable. This CCT copy-and-merge operation successfully addresses the problem of multiple allocations with the same call path, which we illustrated with Figure 5.2. If multiple heap allocated data objects have the same allocation call path, they are merged online and treated as a single variable. Memory accesses to different variables are separated into different groups identified by the variables’ allocation paths.

If HPCToolkit does not find a heap allocated variable matching an access, it looks
up the effective address for the access in data structures containing address ranges for static variables. The search is performed on each load module (the executable and dynamically loaded libraries) in the active load module list to look for a variable range. If the sample accesses a static variable, the profiler records the variable name from the symbol table in a dummy node and inserts it into the static data CCT of the thread that takes the sample. The sample with the full call path is inserted under the dummy node. Therefore, if multiple samples touch the same static variable, they all have the same dummy node as their common prefix for merging. Like heap allocated variables, the dummy nodes of static variable names separate the CCT into multiple groups.

If the sample does not access any heap allocated variable or static variable, we insert the sample to the thread’s unknown data CCT. All of a thread’s call paths touching unknown data are grouped together in this CCT.

The data-centric attribution strategy used by HPCToolkit’s profiler aggregates data access samples in both coarse-grain (storage class) and fine-grain (individual variables) ways. Because each thread records data into its own CCTs, no thread synchronization is needed and data-centric attribution incurs little overhead.

5.2.2 Post-mortem Analyzer and User Interface

HPCToolkit’s post-mortem analyzer takes load modules and profiles as inputs. It reads the symbol table from the executable and dynamically loaded libraries, and maps CCT nodes to either function or variable symbols. It then extracts line mapping information from debugging sections and maps symbol names to source code.

To generate compact profile results, which is important for scalability, HPCToolkit’s post-mortem analyzer merges profiles from different threads and processes.
Data-centric profiles, which consist of at most three CCTs per thread (one each for static, heap, and unknown storage classes), are amenable to coalescing. The analyzer merges CCT’s of the same storage class across threads and processes. Context paths for individual variables and their memory accesses can be merged recursively across threads and processes. For heap allocated variables, if their allocation call paths are the same, even they are from different threads or processes, they are coalesced. For static variables, the analyzer merges them from different threads and processes if they have the same symbol name. Because all memory access call paths have variable CCT nodes as prefixes, they can be automatically merged after the merging of variables. The profile merging overhead grows linearly with the increasing number of threads and processors used by the monitored program. HPCToolkit’s post-mortem analyzer uses a scalable MPI-based algorithm to parallelize the merging process using a reduction tree [130].

Finally, the analyzer outputs a database for HPCToolkit’s GUI. The GUI sorts performance metrics related to each variable and instruction. It provides a top-down view to explore the costs associated with each dynamic calling context in an execution. One can easily identify performance losses within full calling contexts for variables or instructions. Moreover, HPCToolkit’s GUI also provides a complementary bottom-up view. If the same malloc function is called in different contexts, the bottom-up view aggregates all performance losses and associates them with the malloc call site. In case studies, we show how these two views guide data locality optimization.

5.3 Case Studies

We evaluated HPCToolkit’s data-centric extensions for analyzing data locality issues on two machines. Our study focused on evaluating measurement and analysis of
highly multithreaded executions at the node level. HPCToolkit’s MPI-based post-
mortem analysis naturally scales for analysis of data from many nodes.

The first test platform for our study is a POWER7 cluster. Each node has four
POWER7 processors with a total of 128 hardware threads. Each POWER7 processor
in a node has its own memory controller so there are four NUMA nodes. We reserved
4 nodes, up to 512 threads to evaluate the scalability of HPCToolkit. The applications
we studied use one or both of MPI for inter-node communication and OpenMP for
intra-node communication. HPCToolkit uses marked events to collect performance
data. A second test platform for evaluating HPCToolkit’s data-centric analysis is
a single node server with four AMD Magny-Cours processors. There are 48 cores
in this machine and 8 NUMA locality domains. HPCToolkit uses instruction-based
sampling (IBS) to glean performance data.

We studied five well-known application benchmarks coded in C, C++ and For-
tran, covering OpenMP, MPI and hybrid programming models. We built these pro-
gams using GNU or IBM compilers with full optimization. These programs are
AMG2006 [79], Sweep3D [1], LULESH [77], Streamcluster [31], and Needleman-
Wunsch (NW) [31]. A detailed description of these benchmarks is in Section 2.4.
These benchmarks are configured to scale to the full size of our testbeds. The config-
uration and measurement overhead for these benchmarks is shown in Table 5.1. One
may ask how we chose events to monitor and why these events affect performance.
HPCToolkit either computes derived metrics [87] to identify whether a program is
memory-bound enough for data locality optimization or counts occurrences of a spe-
cific event with a traditional hardware counter and evaluates its performance impact.
We only apply data-centric analysis to memory-bound programs. As the table shows,
in our case studies HPCToolkit’s measurement overhead was 2.3–12%. The profile
<table>
<thead>
<tr>
<th>code</th>
<th>number of cores</th>
<th>monitored events</th>
<th>execution time</th>
<th>execution time with profiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2006</td>
<td>4 MPI x 128 threads</td>
<td>PM_MRK_DATA_FROM_RMEM</td>
<td>551s</td>
<td>604s (+9.6%)</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>48 MPI processes</td>
<td>AMD IBS</td>
<td>85s</td>
<td>90s (+5.8%)</td>
</tr>
<tr>
<td>LULESH</td>
<td>48 threads</td>
<td>AMD IBS</td>
<td>17s</td>
<td>19s (+12%)</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>128 threads</td>
<td>PM_MRK_DATA_FROM_RMEM</td>
<td>25s</td>
<td>27s (+8.0%)</td>
</tr>
<tr>
<td>NW</td>
<td>128 threads</td>
<td>PM_MRK_DATA_FROM_RMEM</td>
<td>77s</td>
<td>80s (+3.9%)</td>
</tr>
</tbody>
</table>

Table 5.1: Measurement configuration and overhead of benchmarks.

size (i.e., space overhead) ranged from 8–33 MB.

5.3.1 AMG2006

Figure 5.4 is an annotated top-down view from HPCToolkit’s hpcviewer for AMG2006, which executes with four MPI processes and 128 threads per process on our POWER7 cluster. To present data-centric measurements, we add variable names or allocation call stacks as context prefixes in the hpcviewer’s pane for program contexts. In the figures for our case studies, we show only inclusive metrics.

In Figure 5.4, the second line of the metric pane shows that 94.9% of the remote memory accesses are associated with heap allocated variables. Data allocated on line 175 of hyper_CAlloc$AF6.3 is the target of 22.2% of the remote memory accesses in the execution. Selecting the allocation call site in the navigation pane shows the source code for the allocation, enabling one to identify the variable allocated. The highlighted source code line shows that the column indices for non-zeros in a matrix (S_diag_j) are the target of these remote accesses. Deeper on the allocation call path, one can see that this variable is allocated by calloc. Immediately below the calloc is a dummy node heap_data_accesses; this node serves as the root of all memory accesses to this data. Calling contexts for two accesses are shown below this point; the leaf of each access call path is highlighted with a rectangle. One access accounts for 19.3%

---

This is a marked event that measures data access from remote memory.
Figure 5.4: The top-down data-centric view of AMG2006 shows a problematic heap allocated variable and its two accesses that suffer most remote events to this variable.

of total remote memory accesses and the other for 2.9%. One can select a highlighted access to display its source code in the top pane. Insets in Figure 5.4 show the source code for these accesses. $S_{\text{diag}}j$ is accessed in different loops. Since these loops are inside OpenMP outlined functions (with suffix $\$$0L$$\$$), they are executed by multiple threads in a parallel region.

The performance data in the GUI shows that there is a mismatch between the allocation of $S_{\text{diag}}j$ and initialization by the master thread in one NUMA domain,
and accesses by OpenMP worker threads executing in other NUMA domains. The workers all compete for memory bandwidth to access the data in the master’s NUMA domain. Besides $S_{\text{diag}_j}$, there are many other variables in AMG2006 with the same NUMA problem. To avoid contending for data allocated in a single NUMA domain, we launch the program with `numactl` [74, 84] and specify that all memory allocations should be interleaved across all NUMA domains. Table 5.2 shows the performance improvement. Using `numactl` reduces the running time of the solver phase from 105s to 87s. However, with `numactl` the initialization and setup phases are slower because interleaved allocations are more costly. The higher cost of interleaved allocation offsets its benefits for the solver phase.

A more surgical approach is to apply `libnuma`’s interleaved allocator [74] only to problematic variables identified by HPCToolkit. We used the bottom-up view provided by HPCToolkit’s GUI, shown in Figure 5.5, to identify problematic variables. This view shows different call sites that invoke the hypre allocator. Figure 5.5 shows that while $S_{\text{diag}_j}$ accounts for 22.2% of remote accesses, there are other six variables that are the target of more than 7% of remote accesses. We focus on these problematic variables. If a variable is allocated and initialized by the master thread, we use `libnuma`’s interleaved allocator instead. If a variable is initialized in parallel, we change the `calloc` to `malloc`, which enables the “first-touch” policy to allocate

<table>
<thead>
<tr>
<th>phases</th>
<th>initialization</th>
<th>setup</th>
<th>solver</th>
<th>whole program</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>26s</td>
<td>420s</td>
<td>105s</td>
<td>551s</td>
</tr>
<tr>
<td><code>numactl</code></td>
<td>52s</td>
<td>426s</td>
<td>87s</td>
<td>565s</td>
</tr>
<tr>
<td><code>libnuma</code></td>
<td>28s</td>
<td>421s</td>
<td>80s</td>
<td>529s</td>
</tr>
</tbody>
</table>

Table 5.2: Improvement for different phases of AMG2006 using coarse-grained `numactl` and fine-grained `libnuma`. 
Figure 5.5: The bottom-up data-centric view of AMG2006 shows problematic allocation call sites which may reside in different call paths.

memory near the computation. Table 5.2 shows that using libnuma to apply interleaved allocation selectively avoids dilating the cost of the setup and initialization phases. Moreover, the execution time of the solver phase is 8% faster than when using numactl and ubiquitous interleaved allocation. Using libnuma, we avoid interleaved allocation for thread local data, which eliminates remote accesses to that data that arise with ubiquitous interleaved allocation.
5.3.2 Sweep3D

We ran Sweep3D on our 48-core AMD system and used IBS to monitor data fetch latency, which highlights data locality issues. The second line of the navigation pane in Figure 5.6 shows that 97.4% of total latency is associated with heap allocated variables. The top three heap allocated variables, Flux, Src, and Face account for 39.4%, 39.1%, and 14.6% of the latency respectively. Because these three arrays account for 93.1% of total latency, we only focus on optimizing their locality. Figure 5.7 shows
Figure 5.7: The data-centric view of Sweep3D shows a memory access with long latency to array Flux. This access is in a deep call chain.

a problematic access to Flux, which accounts for 28.6% of the total latency. This access, residing in line 480, is deeply nested in the call chain and loops. The two inner-most loops in line 477 and 478 traverse Flux with left-most dimension first and right-most dimension second. Since Fortran uses column-major array layouts, the loop in line 477 has a long access stride. These long strides disrupt spatial locality and hardware prefetching, which leads to elevated cache and TLB miss rates. To improve data locality, one could consider changing the access pattern by interchanging loops or transforming the data layout. In this case, interchanging loops is problem-
atic. Examining the rest of the accesses to Flux, we see all are problematic, so we interchange the dimensions of Flux by inserting the last dimension between the first and second. With this data transformation, accesses to Flux have unit stride and improved spatial locality.

Both Src and Face suffer from the same spatial locality problem as Flux. Similarly, we transpose their layouts to match their access patterns. The optimization reduces the execution time of the whole program by 15%. It is worth noting that marked event sampling on POWER7 can also identify such optimization opportunities. One can sample PM_{MRK\_DATA\_FROM\_L3} event \(^\dagger\) to quantify the locality issue in Sweep3D. Because Sweep3D is a pure MPI program, no NUMA problem exists because MPI processes are always co-located with their data and thus there is no need to examine NUMA-related events.

5.3.3 LULESH

We ran LULESH on our 48-core AMD system. As with Sweep3D, we used IBS for data-centric monitoring. The first metric column in Figure 5.8 shows the data access latency associated with heap allocated variables in LULESH. The second metric column shows a NUMA-related metric that reflects accesses to remote DRAM; this event is analogous to the POWER7 marked event PM_{MRK\_DATA\_FROM\_RMEM}. The figure shows that heap allocated variables account for 66.8% of total latency and 94.2% of the execution’s remote memory accesses. The individual heap allocated variables are shown along the allocation call path. Annotations to the left of the allocation call sites show the names of the variables allocated. Each of the top seven heap allocated variables accounts for 3.0–9.4% of the total latency. The R_{DRAM\_ACCESS} \(^\dagger\)This is a marked event that monitors cache hits in L3 cache.
metric shows that most of these variables are accessed remotely. By examining the source code, we found that all heap allocated variables in LULESH are allocated and initialized by the master thread. According to the Linux “first touch” policy, all of these variables are allocated in the memory attached to the NUMA node containing the master thread. Consequently, the memory bandwidth of that NUMA node becomes a performance bottleneck. To alleviate contention for that bandwidth, we use libnuma to allocate all variables with high remote accesses in an interleaved fashion. This optimization speeds up the program by 13%.

We continue our analysis of LULESH by considering static variables. Figure 5.9 shows that static variables account for 23.6% of total access latency. The static
Figure 5.9: The data-centric view shows a problematic static variable and its accesses with high latency in LULESH. The ellipse shows that variable’s name.

Variable \( f_{elem} \) is a hotspot because it accounts for 17% of total latency. There are two principal loops that have high access latency for \( f_{elem} \). Both loops have exactly the same structure and one is shown in the source pane of Figure 5.9. From the figure, one can see that \( f_{elem} \) is a three-dimensional array. The first dimension (left-most) is an indirect access using array \( \text{nodeElemCornerList} \) in line 801. The last dimension (right-most) is computed from a function \( \text{Find Pos} \) in line 802. Thus, accesses to \( f_{elem} \) are irregular. Though optimizing data locality for irregular accesses is difficult, we found one opportunity to enhance the data locality for \( f_{elem} \) in this loop. The second dimension (highlighted by a rectangle) ranges from 0 to 2. We
transposed \texttt{felem} to make this dimension the last, which enables these three accesses to exploit spatial locality since C is row-major. This transposition reduces LULESH’s execution time by 2.2%.

### 5.3.4 Streamcluster

![Streamcluster benchmark](image)

**Figure 5.10**: The data-centric view associates a large number of NUMA related events to a problematic heap allocated variable and its inefficient accesses in Streamcluster benchmark.

We ran Streamcluster on a node of our POWER7 system with 128 threads. Streamcluster suffers from serious NUMA data locality issues. Figure 5.10 shows that 98.2% of total remote memory accesses are related to heap allocated variables.
The annotation to the left of the allocation that accounts for 92.6% of total remote accesses shows that it is associated with the variable block. Line 175 contains problematic accesses to p1.coord and p2.coord, which use pointers to access regions of block. This code is called from two different OpenMP parallel regions and accounts for 55.5% and 37% of total remote accesses from these two contexts. Examining the source code, we found that block is allocated and initialized by the master thread, so all worker threads access it remotely. We address this problem by leveraging the Linux “first touch” policy. Initializing block in parallel allocates parts of it near each thread using it. The optimization reduces both remote accesses and contention for memory bandwidth. We also applied this optimization to point.p, which accounts for 5.5% remote accesses. This optimization reduces Streamcluster execution time by 28%.

5.3.5 Needleman-Wunsch

We ran Needleman-Wunsch on a node of our POWER7 system with 128 threads. As with Streamcluster, this code suffers from a high ratio of remote memory accesses. A snapshot of HPCToolkit’s GUI shown in Figure 5.11 indicates that 90.9% of remote memory accesses are associated with heap allocated variables. Two variables, reference and input_itemsets, are hot spots that account for 61.4% and 29.5% of total remote accesses. The problematic accesses occur on lines 163–165. The maximum function called inside an OpenMP parallel region _Z7runTestiPPc.omp_fn.0 takes both reference and input_itemsets as inputs. However, both variables are allocated and initialized by the master thread. To address this problem, we use libnuma to distribute the allocation of these two variables across all NUMA nodes to alleviate contention for memory bandwidth. This optimization speeds up the program by 53%.
5.4 Discussion

Augmenting HPCToolkit with support for data-centric profiling of highly parallelized programs enables it to quantify temporal, spatial, and NUMA locality. HPCToolkit’s data-centric measurement and analysis capabilities leverage PMU hardware on IBM POWER7 and AMD Opteron processors. By using sampling and a compact profile representation, HPCToolkit has low time and space overhead; this helps it scale to a large number of cores. In case studies, we applied HPCToolkit’s data-centric analysis
capabilities to study five well-known parallel benchmarks and attribute data-centric metrics to both code and variables. These capabilities provided intuitive analysis results that enabled us to easily identify and optimize data locality bottlenecks. With data-centric feedback from HPCToolkit, we were able to improve the performance of these benchmarks by 13–53%.

Hardware sampling support for data-centric profiling is insufficient in today’s supercomputers. For example, ORNL’s Titan supercomputer uses AMD processors that support instruction-based sampling. However, the operating system on Titan disables IBS. LLNL’s Sequoia supercomputer is based on Blue Gene/Q processors. Though the A2 cores of a Blue Gene/Q ASIC have SIAR and SDAR registers as part of support for instruction sampling, this capability is not usable because full support for instruction sampling is missing from the multicore ASIC. The utility of measurement-based data-centric analysis of parallel program executions, which we demonstrate, motivates including hardware support for data-centric measurement in future generation systems.

We plan to extend our work in several ways. First, rather than overlooking heap allocations smaller than 4K, we think that monitoring some of them will enable HPCToolkit to provide useful data-centric feedback for programs with data structures built from lots of small allocations. Second, we plan to explore extensions that enable us to associate data-centric measurements with stack-allocated variables. Finally, we plan to enhance HPCToolkit’s measurement and analysis to provide guidance for where and how to improve data locality by pinpointing initializations that associate data with a memory module and identifying opportunities to apply transformations such as data distribution, array regrouping, and loop fusion.
Chapter 6

Analyzing Memory Bottlenecks with Cache Simulation

In modern processors, the speed gap between caches and memory is wide. For example, it takes 3–5 CPU cycles to access the L1 cache but several hundred cycles to access the memory. Thus, for good performance, one must exploit data locality and make good use of caches. There are two kinds of data locality: spatial and temporal. An access pattern exploits spatial locality when it accesses a memory location and then accesses a location within the same cache line soon afterwards. If a program’s access pattern does not match the data layout, spatial locality goes unexploited. Temporal locality occurs when the same memory location is accessed two or more times close together in time.

There are three ways that one might assess locality in a program. Cașcaval and Padua describe a compiler-based approach for estimating cache misses and locality [27]. Their method, which uses data dependence distances, can be accurate for regular programs; however, for programs with irregular access patterns, one must execute the code and measure or simulate locality. On some processors, a hardware performance monitoring unit (PMU) can sample a program execution with low overhead and collect data locality related metrics for memory accesses [42, 45, 126]. While hardware measurements can assess locality with little runtime overhead, they can’t diagnose underlying causes of poor locality. Using simulation, one can compute detailed measures of data locality, along with information that pinpoints the causes of
poor locality. However, detailed cache simulation of every memory access can slow an execution by $1000 \times$ [16].

Simulators quantify data locality by computing reuse distance, also known as LRU stack distance [54]. A long reuse distance for a cache line predicts a capacity cache miss, revealing poor data locality. However, existing tools that calculate reuse distance using cache simulators lack one or more capabilities needed for detailed diagnosis of locality problems. First, data locality optimization obtains non-trivial performance improvement iff data accesses are bottlenecks. Unless a code is sufficiently memory-bound, optimizing data locality may not improve performance. Second, reuse distance information alone isn’t enough. Diagnosing poor memory locality requires correlating reuse distance information back to the instructions sharing data and the contexts in which these instructions execute. Poor memory locality should be attributed to data structures as well. Third, compiler-based tools for measuring reuse distance can’t compute accurate information about data access patterns in executables produced by other optimizing compilers. Finally, collecting accurate reuse distance information for every memory access is very costly; thus, an important aspect of any tool that measures reuse distance is how it mitigates this cost.

In this chapter, we describe a novel, efficient, and effective tool for data locality measurement and analysis. Unlike other tools, our tool uses both statistical PMU sampling to quantify the cost of data locality bottlenecks and cache simulation to compute reuse distance to diagnose the causes of locality problems. This approach enables us to collect rich information to provide insight into a program’s data locality problems. Our tool attributes quantitative measurements of observed memory latency to program variables and dynamically allocated data, as well as code. Our tool also identifies data touched by reuse pairs as well as the accesses involved, identified with
their full calling context. Finally, our tool employs both sampling and parallelization to accelerate the computation of representative reuse distance information. In experiments with five serial codes, our average increase in execution time was 13%. To demonstrate the utility of our tool, we describe our experiences studying five benchmarks. Our tool pinpointed causes of poor locality in each. Our tool identified that three of these codes were memory bound and provided insights that enabled us to make non-trivial improvements to each of these codes.

The rest of this chapter is organized as follows. Section 6.1 describes derived metrics we use to identify data locality bottlenecks. Section 6.2 describes our mechanism for lightweight reuse distance collection. Section 6.3 describes our tool implementation. Section 6.4 describes case studies in which we use our tool to pinpoint data locality bottlenecks in five benchmarks. We then explore the effect of optimizations suggested by insights from our tool. Section 6.5 discusses some conclusions of this chapter and describes our plans for future work.

### 6.1 Identify Data Locality Bottlenecks

The first step in data locality optimization is to pinpoint data locality bottlenecks. A data locality bottleneck is a *time-consuming* code section whose performance is degraded significantly by the latency of accessing data in caches and memory. Only improving data locality for a data locality bottleneck can significantly reduce a program’s execution time.

To identify data locality bottlenecks, we compute $l_{ins}$, the average memory latency per instruction, as shown below in Equation 6.1.

$$l_{ins} = \frac{\text{latency}}{\#ins} = \frac{\text{latency}}{\#mem} \times \frac{\#mem}{\#ins}$$  (6.1)
In Equation 6.1, latency is the total memory access latency in cycles measured for all memory accesses by instructions in a code region; \#mem is the number of memory-related instructions; and \#ins is the total number of instructions of any kind. $l_{ins}$ represents the average memory hierarchy delay per instruction in a code region, computed as the product of two terms. The first term represents the average data latency per access. The second term represents the relative frequency of memory accesses. In our experience, CPU bound programs have values of $l_{ins} < 0.1$. If $l_{ins}$ is larger than 0.5 for the entire program, then the program is sufficiently memory-bound that it could benefit from data locality optimization.

We use our latency metric to quantify the severity of data locality bottlenecks. Table 6.1 shows how different $l_{ins}$ and latency combinations can be used to prioritize optimization efforts. If $l_{ins}$ is low, there is little to be gained from optimization. If $l_{ins}$ is high, reorganizing data or computation to reduce memory access latency could improve performance. If both $l_{ins}$ and latency are high, then program changes might dramatically reduce latency, which could significantly reduce execution time.

Our tool uses instruction-based sampling to accumulate the total memory latency observed for each memory instruction and the total number of dynamic instances observed for each instruction (memory access or otherwise). In post-mortem analysis, we use this raw data to calculate $l_{ins}$ for memory instructions, loops, function bodies, and call paths. This information enables us to pinpoint problematic memory access...
instructions in each code region that represents a data locality bottleneck.

We begin our analysis of a program execution by examining $l_{ins}$ for the outermost program scope, e.g., $\text{main}$. If $l_{ins}$ is sufficiently high, indicating a data locality bottleneck, we recursively narrow our focus to the child scopes (e.g., loops or called procedures) where the problem is most severe, i.e., where $l_{ins}$ and latency are high, until we identify the scope or scopes that are the root causes of the problem; these scopes merit tuning.

6.2 Lightweight Reuse Distance Measurement

Though we can use $l_{ins}$ and latency metrics to identify problematic memory accesses that degrade a program’s performance, these memory accesses are victims rather than causes. The real cause is that the data they access have been evicted from cache hierarchies. Cache eviction occurs when reuse distance is larger than the cache capacity.* We use reuse distance information to understand why specific data is evicted from cache. We focus our optimization efforts on references using and reusing the data. We must consider both the access patterns of each of these references and the layout of the data they touch.

To compute reuse distance measurements efficiently, we employ a sampling-based technique known as shadow profiling [105] to collect performance data with low time overhead. The principle of shadow profiling is to cut a program execution into slices and schedule computation of reuse distance measurements for each slice on an idle computational core in the system. Since the execution of shadow measurement processes can be overlapped with the native execution of a program, shadow profiling

*There are also other causes for cache evictions, such as cold and conflict misses; however, in this chapter, we only focus on capacity cache misses.
can occur with little time overhead. Each of our shadow processes executes under the control of Pin [91], which is a dynamic binary instrumentation tool. Pin provides a rich set of high-level APIs to instrument a program with analysis routines at different granularities. We use a Pin plug-in to instrument each memory access, call, and return. We describe the work performed by our instrumentation in the following sections. Figure 6.1 and our explanation below describe how our tool uses shadow profiling to collecting reuse distance measurements in practice:

1. The main process begins execution and is monitored using instruction-based sampling. At an IBS sample event, if the main process has executed for a sufficiently long interval since the beginning of the execution or the last fork event, the process will decide to fork.

2. The main process uses call stack unwinding to determine the full calling context
of the fork event† and then forks a child process with a clone of the current process state. The child process state includes information about the calling context of the fork along with information about live blocks of heap-allocated data. We explain the details of the information our tool maintains for dynamic memory allocations and how it is used in Section 6.2.3.

3. After the fork event, the child launches a copy of Pin and directs Pin to attach to the child. The rest of the child’s execution will occur under the control of Pin.

4. As Pin executes the child process, it uses a plug-in to monitor all memory accesses and compute reuse distance metrics for a predefined number of memory accesses. When the required number of memory accesses has been monitored, the child process records its reuse distance measurements into a file and exits.

The following sections describe our approach in greater detail. Section 6.2.1 describes the reuse distance metrics we compute to support effective analysis. Section 6.2.2 describes an innovative method for collecting full call paths of both use and reuse memory accesses. Section 6.2.3 describes data-centric analysis that our tool employs to identify data structures involved in the reuse distance analysis. Section 6.2.4 analyzes the accuracy and overhead of our sampling-based reuse distance measurement mechanism.

6.2.1 Reuse Metric Computation

Our Pin instrumentation for memory accesses gets invoked with the instruction pointer and the effective address each time an instruction loads from or stores to

†We use HPCToolkit call stack unwinder [133] to collect calling contexts.
memory. To compute reuse distance, we use a hash table to maintain the set of effective addresses touched by memory accesses in a child process. We number addresses accessed and compute the reuse distance when an address is retouched. The granularity of reuse distance information maintained can be configured to individual memory addresses or cache lines. We define two reuse metrics: average reuse distance and reuse frequency. To compute aggregate reuse information, we need to judge whether two reuse pairs are the same. If reuse pair $p_2$ has the same instruction pointers, touches the same data, and has the same call paths as a previous pair $p_1$, then we aggregate $p_2$ into $p_1$ by incrementing $p_1$’s frequency metric and adding the reuse distance of $p_2$ to that of $p_1$. In post-mortem analysis, we use the frequency count and total distance for a reuse pair to compute average reuse distance for that pair.

6.2.2 Call Path Collection

Beyls and D’Hollander show that call paths to both use and reuse instructions are important for data locality optimization to reduce reuse distances [16]. The reuse scope, defined as the least common ancestor (LCA) [16] of call paths to use and reuse instructions, is where optimization needs to occur. To reduce the reuse distance and improve data locality, when legal, one should hoist the use and reuse instructions into the LCA and apply loop fusion.

A full call path to a memory access instruction is represented as a chain of call sites beginning with main and ending with the function containing the memory access instruction. Our tool uses a two-part technique to obtain the call path to each monitored memory access at low expense. We initialize the call path for a child process with the full calling context of the parent at the time it forks. We collect
the parent process’s calling context using HPCToolkit’s call stack unwinder, which employs on-the-fly binary analysis [133]. When our Pin plug-in gains control of a child process, the initial execution context of the child is in the signal handler where it was forked. Our Pin plug-in advances the execution of the child process until the child returns to the context in the application code where the signal was received. At that point, our Pin plug-in instruments all function calls and returns in the child process to maintain the call path as execution proceeds. From that point onward, our Pin plug-in incrementally updates the call path for the current execution point upon each function call and return; this approach is similar to one used by DeadSpy [30], which also incrementally maintains a call stack in a Pin plug-in to attribute memory accesses to a full call path. Our tool appends a new call site to the call path when it encounters a call instruction and removes the last call site on the path when it encounters a return instruction. Using this approach, we know the full call path to the currently executing instruction in the child process at all times.

6.2.3 Data-centric Analysis

As we have shown previously [85], knowing which source statement is responsible for high-latency accesses often isn’t enough to diagnose data locality problems if the statement touches multiple data objects. To understand data locality problems in detail, one must associate reuse information not only with instructions accessing a data object, but also with data objects themselves. Below we describe how we associate accesses with static data, heap-allocated data, and stack allocated data.

**Static data** Static variables include global and local variables declared using the `static` keyword in C/C++ or the `save` keyword in Fortran. Static variables are
live throughout the whole program execution. In an executable’s symbol table, static
variables are visible as symbols tagged with OBJECT. In the main process, our tool
reads the symbol table to extract the data range and name for each static variable and
records these signatures in internal data structures for later use. When the parent
process is forked, the information associating address ranges with static variables
becomes available in the child’s address space so that each access to a static variable
can be recognized as such.

Heap data Variables in the heap are allocated dynamically during execution by
one of the malloc family of functions. Since heap-allocated data may be known by
different aliases, e.g., function parameters, at different points in an execution, we use
the full call path of the allocation point of a heap-allocated data block to uniquely
identify it throughout its lifetime. To associate a heap-allocated variable with its
allocation call path, we wrap all memory allocation and free operations in both the
main and child processes. At each allocation, we enter an association between the
address range of a data block and its allocation point. At each free, we delete an
association from the map. In the main process, we use call stack unwinding (as
described earlier) at the allocation point to determine the allocation context. When
the main process forks a child process for monitoring, the child inherits the state of
the allocation map of the main process. Any allocation or free performed by the child
process updates the map inherited from the main process. When the child process
encounters an allocation point, it uses the execution context maintained by our Pin
plug-in instead of gathering the context with call stack unwinding.

Stack data Local variables for a procedure are allocated on the call stack upon
entry to the procedure. We use the following strategy to associate memory accesses
with stack variables. Before an executable is launched, we use an offline tool to analyze the debugging information for the executable. This information is available in the executable's DWARF debugging sections. For each procedure, we compute a sequence of tuples that represent its local variables. Each tuple contains the name of the procedure, the name of a local variable, its offset in the procedure's frame, and the source file and line on which it is declared. If DWARF information is missing for a procedure, we only record a tuple that indicates the extent of a procedure's frame. We write these tuples to a file that will be read by each shadow profiler process. During execution, when we encounter an access to an address in the stack while executing a child process with our Pin plug-in, we examine the call stack for the current execution context, we use information about the stack pointer and base pointer for each procedure on the call path to identify the procedure frame on the stack that contains the target address, and then use the DWARF information to associate the access with the local variable whose offset range within the frame contains the target address.

6.2.4 Overhead and Accuracy Analysis

The overhead of our measurement approach comes from both the main and child processes. In the main process, we use instruction-based sampling to collect information needed to associate $l_{ins}$ and $latency$ with call paths, as described in Section 6.1. With one sample per 64K micro ops, its overhead is less than 5%. Another source of overhead in the parent process is the wrapping of heap data allocation operations to support data-centric analysis. This overhead was not particularly large for any of our experiments. We define the total overhead of monitoring in the parent process as $m$.

The costs associated with child monitoring processes forked to measure reuse
distance are affected by the frequency with which they are forked and the number of memory accesses that they monitor before quitting. The period between forking monitoring processes should be chosen to avoid oversubscribing available cores. To quantify the execution cost of monitoring processes, we define several parameters: the number of cores in the system $p$, the average execution time for an individual child process $t$ and the period $r$ with which monitoring processes are created. It is worth noting that $t$ is relatively consistent because we specify that a child process will monitor a fixed number of memory accesses before terminating. Equation 6.2 shows the criteria for choosing an appropriate $t$ and $r$. Since the native execution of the main process takes up one core, remaining cores can be used for monitoring processes.

$$\frac{t}{r} \leq p - 1$$ (6.2)

As shown in Figure 6.1, the executions of child processes overlap with one another and the parent process as well. The only non-overlapped overhead is that of the last child process. Since the last child process can be triggered in an interval ranging from $(0, r)$ from the end of the main process, the expected non-overlapped execution time of child processes is $t - \frac{r}{2}$. Therefore, we quantify the non-overlapped execution overhead $o$ of our method in Equation 6.3.

$$o = m + t - \frac{r}{2}$$ (6.3)

The accuracy of our sampling approach also depends upon $t$ and $r$. On the one hand, $t$ is proportional to the number of memory accesses monitored by one child process. The more memory accesses monitored, the more accurate our reuse
measurements will be. On the other hand, \( r \) reflects the code coverage by our method. A small \( r \) means high sampling frequency, which will yield high coverage of the execution. However, according to Equation 6.2, \( t \) and \( r \) trade off against one another and should be carefully chosen. In our tool, \( t \) and \( r \) may be chosen by users to trade off overhead vs. accuracy.

Our tool also incurs memory overhead that affects the forking frequency of child processes. Because our tool periodically forks processes that inherit the parent process’s context, it requires more memory than the native execution. We should make sure that the memory footprint of the main process and co-existing child processes does not exceed the total memory in the system.

### 6.3 Tool Organization

Our tool consists three parts: a call path profiler, a Pin plug-in, and a post-mortem analyzer. The workflow of our tool is shown in Figure 6.2.

#### 6.3.1 Call Path Profiler

The call path profiler injects monitoring code into the address space of an application process as it begins execution. Before profiling begins, it parses the symbol table and DWARF sections in the binary to extract information about static data and procedure frames. This information is passed to each child process as part of the state it inherits from the main process when it is forked. Within the main process, the profiler initiates instruction-based sampling and attributes dynamic instruction counts and latency profiles to their full calling contexts. Periodically, the main process forks off shadow processes that collect detailed reuse information for a pre-determined number of accesses.
6.3.2 Pin Plug-in

When a child process begins, it invokes Pin to monitor its execution. The data space of the child process contains the calling context from which the child process was forked. Our Pin plug-in maintains the calling context at every call and return so that the full calling context is available as each memory access executes. For each memory access, our Pin plug-in is presented with the instruction pointer and the effective address of the memory access. The memory access handler in the plug-in uses a cache simulator to collect reuse distance information. Our plug-in associates each memory access with the full call path in which it occurs. Once a child process has performed the pre-determined number of accesses, the plug-in records reuse distance measurement results in a file.
<table>
<thead>
<tr>
<th>code</th>
<th>LULESH</th>
<th>Sweep3D</th>
<th>LU</th>
<th>Sphot</th>
<th>S3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>overhead</td>
<td>15.5%</td>
<td>12.2%</td>
<td>16.7%</td>
<td>11.3%</td>
<td>8.1%</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>12.8%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Measurement overhead for our tool.

6.3.3 Post-mortem Analysis Tool

The post-mortem analysis tool reads the latency and instruction measurements from the call path profiler and the reuse distance measurements gathered by our Pin plug-in. Our post-mortem analysis tool maps the latency and instruction profile results to the source code using the line information in DWARF sections. A graphical user interface enables the user to explore the latency and reuse distance measurement data for an application interactively. The post-mortem analyzer aggregates all reuse data measurement files collected by the Pin plug-in and correlates the measurements with full call paths. Reuse distance information from our tool enables a developer using our graphical user interface to understand reuse patterns that correspond to high latency measurements.

6.4 Case Studies

To evaluate our tool’s utility, we studied five HPC application benchmarks: LULESH [77], Sweep3D [1], NAS LU [10], Sphot [79], and S3D [32]. The platform for our experiments is a single node system with four AMD 12-core Magny-Cours processors. Our tool collects an instruction-based sample every 64K instructions, invokes shadow processes every 0.8 second, and collects 100M memory accesses in a shadow process. Our platform has 128GB memory, which is large enough for our measurement configurations. The execution time overhead of applying our tool to
Figure 6.3: Pinpoint problematic code sections and instructions in LULESH with their full calling contexts.

the aforementioned applications is shown in Table 6.2.

6.4.1 LULESH

Figure 6.3 shows our graphical user interface — hpcviewer, highlighting one of the data locality bottlenecks in LULESH. In this case, we only consider the inclusive metrics computed by hpcviewer. The LATENCY/INS metric ($l_{\text{ins}}$) for the whole
The text output from the reuse distance measurement for LULESH. It is an example for a reuse pair. The reuse instruction at line 1316 shows high memory access latency in Figure 6.3.

program is 4.84. It means that LULESH is a memory-bound benchmark that could benefit from data locality optimization. Drilling down the call path, we use the LATENCY metric to pinpoint problematic code sections and instructions. A parallel region at line 1229 invokes a loop (shown on the top of the source code pane) that accounts for 35.7% of the total latency. The individual memory access instructions in this loop at lines 1316, 1310 and 1320 are top contributors to this latency.

With knowledge of the problem location, we correlate it with our reuse distance measurements. Reuse distance measurements are not yet available within our graph-
Figure 6.5: LULESH source code for the use and reuse described in Figure 6.4. It shows an optimization opportunity for a pair of OpenMP parallel regions. Arrow points to the use and reuse of \texttt{z8n}. The parallel loops bounded with blue rectangles should be fused to improve data locality.

Figure 6.4 shows textual information about a reuse pair corresponding to one problematic memory access instruction on line 1316 (as the reuse instruction) which accounts for 7.9% of the total memory access latency. The high values for frequency and distance metrics indicate that there is a serious data locality problem associated with this memory access instruction. Our data-centric analysis indicates that the variables touched by both use and reuse are not stack variables. The data allocation call stacks for the use and reuse accesses indicate that both accesses touch \texttt{z8n}. By examining the full calling contexts of both the use and reuse instructions, we see that their least common ancestor is the function \texttt{CalcForceForNodes} called on line 1685 (not shown in the figure). From the least common ancestor, the call paths to the use and reuse lead to two different parallel loops. The source code of loops containing the two accesses is shown in Figure 6.5. Line numbers in Figure 6.5
are consistent with the reuse distance measurement result in Figure 6.4. The arrow in Figure 6.5 marks the use and reuse statements. The variable $z_{8n}$ is reused in a function call and the real reuse points in the function body are enclosed with ellipses. It is worth noting that $x_{8n}$ and $y_{8n}$ also suffer long reuse distances in these two loops.

To reduce the reuse distance between the uses and reuses of $x_{8n}$, $y_{8n}$ and $z_{8n}$, we bring the two parallel loops to the LCA, which means that we remove the function call on line 1539 and inline the body of function $\text{CalcForceForNodes}$. Then we fuse the two parallel loops. The fusion is straightforward because the two loops have the same trip count and stride. The optimization to address the data locality problem highlighted by our tool improves the performance of LULESH by 12.9% on 32 cores.

6.4.2 Sweep3D

Sweep3D is a memory bound benchmark. Figure 6.6 shows that Sweep3D has a latency per instruction of 2.52 over the full execution. Two problematic loops occur at lines 387 and 477, which are deeply nested in a common call path. These loops have latencies per instruction of 3.51 and 3.29, respectively, which are larger than the average. The top pane in the figure shows the source code for both loops. Our reuse distance measurements show that the heap variables $\text{src}$ and $\text{flux}$ suffer long reuse distances, touch more than $4\times 10^5$ cache lines, and have a high reuse frequency of $2.9\times 10^8$. An obvious data locality problem for both variables is the mismatch between their data layout and access patterns. The last dimension of both arrays should be transposed to the second one. This simple data layout transformation improves the performance of the whole program by 9.2%.

More data reuse opportunities can still be observed for both $\text{src}$ and $\text{flux}$. They are reused in the outer loop at line 326. A more intricate loop restructuring described
in [94] improves the performance of Sweep3D by 60%.

6.4.3 NAS LU

Figure 6.7 shows that LU is also memory-bound, but less severely than LULESH and Sweep3D. LU has a 0.87 cycle latency per instruction. Drilling down the call path shows that the most problematic memory accesses are in the inner-most loop beginning at line 194 in file rhs.f. This loop has a latency of more than 10 cycles per instruction. There are two serious data locality problems in this loop. The
Figure 6.7: Identify data locality optimization opportunity in NAS LU benchmark.

access stride for a static array $u$ is large because the memory access pattern and data layout are mismatched. However, it is not easy to fix. Our reuse distance measurements show that $u$ is reused from another loop that is also shown in Figure 6.7. A further examination of the code shows that the use and reuse are in two separate OpenMP parallel loops that have the same loop bounds and stride. Fusing these two parallel loops to reduce the reuse distance of $u$ improves the overall performance of the program by 5.1\% in an 8-thread execution.
Figure 6.8: Identify problematic memory accesses with reuse annotations.

6.4.4 Sphot

Figure 6.8 shows the call stack view of problematic memory accesses in Sphot. Memory accesses in the deeply nested loop highlighted in the figure account for 70.7% of the measured access latency. The top four memory accesses in this loop show relatively high reuse distances. Red arrows annotate the use and reuse points indicated by our reuse distance measurements. The arrow in the circle means that the data is reused by the same statement in different iterations. The poor locality for itype, bom
and sqm is due to the mismatch between access patterns and data layouts: the index of the inner-most loop ks is the last dimension of the three arrays. Additionally, our tool also pinpoints these three arrays are allocated on the stack in subroutine execute.

We changed the data layouts in Sphot by exchanging the first and second dimensions of the problematic arrays. The transformation reduces the latency in this loop by 41.2%. However, the performance of total program improves by just 1.2%. The reason for the tiny improvement is that Sphot is not memory-bound. Shown in Figure 6.8, the latency per instruction metric for the whole program is less than 0.1; therefore, optimizing data locality yields little overall improvement, as one would expect.

### 6.4.5 S3D

Figure 6.9 shows that S3D is not a memory-bound application: its latency per instruction is only 0.02. For that reason, optimizing data locality should not improve performance much. To validate this hypothesis, we tried to improve some long latency memory accesses. By drilling down the call path from main, we identified a loop at line 391 in file chemkin.m.f90 that accounts for 9.6% of the total latency. The four dimensional array yspecies allocated on the heap is re-touched in the same instruction within a different iteration, shown with the red circle in Figure 6.9.

The long distance is due to the mismatch of memory access pattern and data layout of yspecies. Since Fortran is column-major, traversing the last dimension of the array causes memory accesses with large stride. We transposed the data layout of yspecies to match the access pattern. However, the overall performance improvement with the optimization was less than 1%, which confirmed our expectation that improving the
Figure 6.9: Pinpoint one of problematic memory accesses in the S3D program with its full calling context.

data locality of a non-memory-bound code would yield little benefit.

6.5 Discussion

This chapter describes the design and implementation of a data locality profiler. It combines various sampling mechanisms to achieve very low measurement overhead and rich optimization information. With several case studies, we show that our tool can successfully identify poor memory access patterns and problematic variables.
We envision extending our tool in two ways. First, our profiler operates on single-threaded programs. It could be extended to multithreaded programs by carefully parking threads before forking and creating matching threads in each child process. Second, our tool could be extended to analyze latency and reuse measurement data, as well as code structure, to provide guidance about how to optimize data locality.
Chapter 7

Analyzing NUMA Inefficiencies in Threaded Programs

As microprocessors have become faster and multiple cores per chip have become the norm, memory bandwidth has become an increasingly critical rate-limiting factor for system performance. For multiprocessor systems, scaling memory bandwidth proportional to processing power has led to designs in which microprocessors include memory controllers on chip. As a result, the aggregate memory bandwidth of systems scales with the number of microprocessors.

Multiprocessor systems in which some memory is locally-attached to each processor are known as Non-Uniform Memory Access (NUMA) architectures because it is faster for a microprocessor to access memory that is locally attached rather than memory attached to another processor. Some microprocessors, e.g., IBM’s POWER7 [123], have NUMA organizations on-chip as well, with some cores having lower latency access to some directly-attached cache and/or memory banks than others. To simplify discussion of systems where NUMA effects may exist within and/or between microprocessors, we simply refer to cache/memory along with all CPU cores that can access it with uniform latency as a NUMA domain. There is not only a difference in latency when accessing data in local vs. remote NUMA domains, there is also a difference in bandwidth: cores typically have significantly higher bandwidth access to local cache/memory than remote.

Systems with multiple NUMA domains are challenging to program efficiently.
Without careful design, multithreaded programs may experience significant performance losses when running on systems with multiple NUMA domains if they access remote data too frequently. On such systems, multithreaded programs achieve best performance when threads are bound to specific cores and each thread mostly processes data co-located in the same NUMA domain as the core in which it executes. In addition to latency, contention can also hurt the performance of multithreaded programs. If many of the data accesses performed by a multithreaded program are remote, contention for limited bandwidth between NUMA domains can be a bottleneck. This problem can be particularly acute if a large data array is mapped to a single NUMA domain and many threads compete for the limited bandwidth in and out of that domain. This situation is more common than one might think. By default, today’s Linux systems employ a “first-touch” policy to bind pages of memory newly-allocated from the operating system to memory banks directly attached to the NUMA domain where the thread that first accesses the page resides. As a result, if a single thread initializes a data array, but multiple threads process the data later, severe contention can arise.

Tailoring a program for efficient execution on systems with multiple NUMA domains requires identifying and adjusting data and computation layouts to minimize each thread’s use of remote data and avoid contention for bandwidth between NUMA domains. Due to the myriad of opportunities for mis-steps, tools that can provide insight into NUMA-related performance losses are essential for guiding optimization of multithreaded programs.

There are two broad classes of techniques for identifying NUMA bottlenecks in a program: simulation and measurement. Tools such as MACPO [113] and NUMA-grind [141] use simulation to identify NUMA bottlenecks in a program. A drawback
of tools that simulate all memory accesses is that they are slow, which makes them of limited use for programs with significant running time. To address this shortcoming, a new class of tools, e.g., ThreadSpotter [115], apply simulation sparingly to selected memory accesses to reduce execution overhead.

In contrast, measurement-based tools, such as TAU [121], Intel Vtune Amplifier [66], IBM Visual Performance Analyzer (VPA) [65], AMD CodeAnalyst [4], and CrayPat [43] can gather data to provide insight into NUMA performance at much lower cost. On today’s microprocessor-based systems, such tools can monitor NUMA-related events using a microprocessor’s on-chip Performance Monitoring Unit (PMU). These tools measure and aggregate NUMA-related events and associate them with source code contexts, such as functions and statements. We call this approach code-centric analysis. A shortcoming of code-centric measurement and analysis is that it often fails to provide enough guidance for NUMA optimization [96]. Data-centric analysis tools, which can provide deeper insight into NUMA bottlenecks, use advanced PMU capabilities to gather instruction and data address pairs to associate instructions that access memory with the variables that they touch. Existing data-centric tools, such as HPCToolkit [85, 87, 89], Memphis [96] and MemProf [75] can identify both instructions and variables that suffer from NUMA problems.

There are three principal shortcomings of existing tools for measurement-based data-centric analysis. First, since PMU support for data-centric analysis differs significantly across microprocessors from different vendors and even processor generations, most data-centric tools only support only a single family of PMU designs. Second, these tools do not assess the impact of NUMA bottlenecks on overall program performance. Without such information, one may invest significant effort to improve the design of a code for NUMA systems to address measured inefficiencies and net only a
small performance gain. Finally, while existing tools can identify NUMA-related bottlenecks, they fail to provide information to guide NUMA-aware code optimization.

To address these shortcomings, we developed a lightweight tool for measurement and analysis of performance problems on multicore and multiprocessor systems with multiple NUMA domains. Our profiler outperforms existing tools in three ways. First, our tool is widely applicable to nearly all modern microprocessor-based architectures. Second, we define several derived metrics that can be computed by tools to assess the severity of NUMA bottlenecks. These metrics can effectively identify whether NUMA-related performance losses in a program are significant enough to warrant optimization. Third, our tool analyzes memory accesses and provides a wealth of information to guide NUMA optimization, including information about how and where to distribute data to maximize local accesses and reduce memory bandwidth contention.

We describe case studies using four well-known multi-threaded codes that highlight the capabilities of our tool and demonstrate their utility. For three of the programs, our tool provided unique insights unavailable with other tools. These guided us to code changes that yielded non-trivial performance improvements. In the course of our studies, we found, somewhat surprisingly, that stack variables sometimes play a significant role in NUMA bottlenecks. Another case study demonstrates the utility of our novel metrics for assessing the severity of NUMA bottlenecks.

The rest of this chapter is organized as follows. Section 7.1 describes NUMA problems and introduces NUMA optimization strategies. Section 7.2 describes hardware support for data-centric measurement in modern microprocessors, which provides the foundation for our tool. Section 7.3 describes derived metrics our tool computes to quantify the impact of NUMA-related performance losses. Section 7.4 shows how
we attribute metrics with different views for NUMA analysis. Section 7.5 describes how we efficiently pinpoint locations in the source code that need modification to effect NUMA-aware data distributions. Section 7.6 describes the implementation of our tool. Section 7.7 presents case studies that illustrate the use of our tool to identify and fix NUMA-related bottlenecks in four multithreaded programs. Finally, Section 7.8 summarizes our conclusions of this chapter and plans for future work.

### 7.1 NUMA-aware Program Design

There are two causes of NUMA bottlenecks: excessive remote accesses and uneven distribution of requests to different NUMA domains. In modern processors, remote accesses have more than 30% higher latency than local accesses [122]. If one co-locates data in the same NUMA domain with a thread that manipulates it most frequently, the program can benefit from the fast local accesses. We use the term *co-location* to refer to the process of mapping part or all of a data object and the thread that accesses it most frequently to the same NUMA domain.

On the other hand, uneven distribution of memory accesses to NUMA domains can lead to contention and unnecessary bandwidth saturation in both on-chip and off-chip interconnects, caches, and memory controllers. Contention for interconnect and memory controller bandwidth has been observed to increase memory access latency by as much as a factor of five [41]. Instead of mapping large data objects to a single NUMA domain, in many cases one can reduce contention and associated bandwidth saturation by distributing large data objects across all NUMA domains. We call this optimization *contention reduction*.

Figure 7.1 illustrates various strategies for mapping data to NUMA domains and discusses the latency, contention, and performance issues associated with alternative
Figure 7.1: Three common data distributions in a program on a NUMA architecture. The first distribution allocates all data in NUMA domain 1. It suffers from both locality and bandwidth problems. The second distribution maps data to each of the NUMA domains and avoids centralized contention. The third distribution co-locates data with computation, which both maximizes low-latency local accesses and reduces bandwidth contention.

distributions. One can identify excessive remote accesses to program data objects by measuring NUMA-related events using techniques described in the next section. One can identify contention by counting requests to different NUMA domains. Co-locating data and computation is the most powerful optimization as it reduces the need for bandwidth to remote domains, reduces bandwidth contention, and reduces latency by exploiting the lower latency and higher bandwidth access to local data. In cases where there is not a fixed binding between threads and data and/or concurrent computations may focus on only parts of a data object at a time, then using memory interleaving to avoid contention for a single NUMA domain may be beneficial. However, in some cases, using interleaving to balance memory requests to NUMA domains may hurt
locality and performance [75, 89].

When data objects are not allocated using interleaved memory pages, the Linux “first touch” policy binds a new memory page obtained from the OS to a physical page frame managed by a memory controller when the page is first read or written. To ensure that data will be mapped to the proper NUMA domain, one must carefully design code that first touches each of the principal data structures in a multithreaded program. To control allocation without completely refactoring an application, one can introduce an initialization pass right after a variable is allocated to control its page distribution.

To help application developers tailor a program for NUMA systems, application developers need tools that

- pinpoint the variables suffering from uneven memory requests, so one can use different allocation methods (e.g., interleaved allocation) to balance the memory requests,

- analyze the access patterns across threads to guide NUMA locality optimization, and

- identify where data pages are bounded to NUMA domains.

To our knowledge, no prior profiling tool provides all of these capabilities. In the next section, we describe address sampling—a key technique needed to build tools with these desired capabilities.

### 7.2 Address Sampling

Address sampling, which involves collecting instruction and data address pairs to associate memory references with the data that they touch, is essential for profiling
NUMA access patterns. PMUs on most recent processors from AMD, Intel, and IBM support address sampling. A processor can support efficient NUMA profiling if it has the following three capabilities.

- It can record the effective address touched by a sampled instruction that accesses memory. It is important for this support to guarantee that memory accesses are uniformly sampled.

- It can sample memory-related events, such as load/store instructions and cache accesses/misses, as well as measure memory access latency. Such information is useful for quantifying NUMA-related bottlenecks.

- It can capture the precise instruction pointer for each sample. Special hardware support is required for correct attribution of access behavior to instructions on processors with out-of-order cores [45].

We know of five hardware-based address sampling techniques used in modern processors: instruction-based sampling (IBS) [45] in AMD Opteron and its successors, marked event sampling (MRK) [126] in IBM POWER5 and its successors, precise event-based sampling (PEBS) [68] in Intel Pentium 4 and its successors, data event address sampling (DEAR) [69] in Intel Itanium, and precise event-based sampling with load latency extension (PEBS-LL) [68] in Intel Nehalem and its successors. These hardware sampling mechanisms are described in detail in Section 2.1.

Since support for address sampling is not universally available, e.g., on ARM processors, we developed a software-based strategy for address sampling that we call Soft-IBS. Soft-IBS can simulate address sampling with memory access instrumentation. Using an instrumentation engine based on LLVM [90], we instrument every memory access instruction using a function that captures the effective address and
instruction pointer of the sampled instruction. The engine invokes an instrumenta-
tion stub function that our profiler overloads to monitor memory accesses. Rather
than recording information for each memory access, our profiler records information
for every \( n^{th} \) memory access, where the value of \( n \) can be selected when the profiler
is launched.

These six address sampling mechanisms are the foundation for our NUMA pro-
file tool. The aforementioned hardware and software mechanisms for address sam-
pling each have their own strengths and weaknesses for NUMA profiling. Naturally,
the hardware mechanisms are much more efficient than Soft-IBS. In Section 7.7, we
compare the overhead of address sampling with these various mechanisms.

7.3 NUMA Metrics

Using information we gather with address sampling, we compute several metrics to
gain insight into NUMA bottlenecks. In the next section, we describe metrics to
identify remote accesses and imbalanced memory requests. These metrics can be
computed with information gathered using any mechanism for address sampling. In
Section 7.3.2, we describe some additional metrics that can be derived using informa-
tion available from only some PMU implementations.

7.3.1 Identifying Remote Accesses and Imbalanced Requests

To understand remote accesses, our profiler computes two derived metrics: \( M_l \) and
\( M_r \). \( M_l \) is the number of sampled memory accesses touching the data in the local
NUMA domain, while \( M_r \) is the number of sampled memory accesses touching the
data in a remote NUMA domain. Unless \( M_r \ll M_l \) for a code region, the code region
may suffer from NUMA problems. To compute \( M_l \) and \( M_r \), our tool needs two pieces
of information for each address sample: the NUMA domain that is the target of an effective address and the NUMA domain of the thread performing the access. Our profiler uses the `move_pages` API from `libnuma` [74] to query the NUMA domain for the effective address. To identify the NUMA domain for a thread, we have two mechanisms. With PMU support for address sampling, a sample includes the CPU ID performing the access. For Soft-IBS, we bind each thread to a core and maintain a static mapping between thread and CPU ID that we query at runtime. Our tool uses `libnuma`'s `numa_node_of_cpu` to map the CPU ID to its NUMA domain. For each address sample, if the effective address and thread are in the same NUMA domain, we increment $M_i$; otherwise, we increment $M_r$.

To evaluate memory request balance, our tool counts the number of sampled memory accesses to each NUMA domain by each thread. As before, we identify the NUMA domain for an access using the `libnuma move_pages` API. If the aggregate number of accesses to some NUMA domains is much larger than others, the access pattern may cause memory bandwidth congestion.

By themselves, the aforementioned metrics can be misleading. For example, if a thread loads a variable allocated into its private cache and touches it frequently, though no further remote accesses occur, the $M_r$ caused by this thread is high, because the variable is deemed in the remote NUMA domain by `move_pages`. Therefore, one needs to use other metrics to eliminate this bias. In the next section, we describe a latency-per-instruction metric, which can be computed on some architectures, that addresses this shortcoming.
7.3.2 NUMA Latency per Instruction

IBS and PEBS-LL support measuring the latency of remote accesses. When this information is available, our tool computes the NUMA latency per instruction to provide additional insight into NUMA bottlenecks. We compute the NUMA latency per instruction, $l_{pi_{NUMA}}$, to quantify a NUMA bottleneck’s impact on overall program performance. If a NUMA bottleneck’s $l_{pi_{NUMA}}$ is small, even if $M_r$ is large, NUMA optimization of the program can’t improve performance much. Equation 7.1 defines $l_{pi_{NUMA}}$.

$$l_{pi_{NUMA}} = \frac{l_{NUMA}}{I}$$

$$= \frac{l_{NUMA}}{I_{NUMA}} \times \frac{I_{NUMA}}{I_{MEM}} \times \frac{I_{MEM}}{I}$$

In the equation, $l_{NUMA}$ is the total latency of all remote accesses (including remote caches and memory); $I_{NUMA}$, $I_{MEM}$, and $I$ represent the number of remote accesses, memory accesses, and instructions executed, respectively. This metric can be computed for the whole program or any code region. The equation can be computed as the product of three terms: the average latency per remote access, the fraction of memory accesses that are remote, and the ratio of memory accesses per instruction executed. The resulting quantity indicates whether NUMA performance losses are significant for a code region.

Equation 7.2 shows how we approximate $l_{pi_{NUMA}}$ using address sampling with IBS.

$$l_{pi_{NUMA}} \approx \frac{l_{NUMA}^s}{I^s}$$

$l_{NUMA}^s$ is the accumulated latency for sampled remote accesses; $I^s$ is the number
of sampled instructions. Calculating $lpi_{NUMA}$ this way yields an approximate value because $l^s_{NUMA}$ and $I^s$ are representative subsets of $l_{NUMA}$ and $I$. Equation 7.3 shows how we approximate $lpi_{NUMA}$ using address sampling with PEBS-LL.

$$lpi_{NUMA} \approx \frac{l^s_{NUMA}}{E^s_{NUMA}} \times \frac{E_{NUMA}}{I}$$  (7.3)

As PEBS-LL samples memory access events, we can obtain an absolute event number $E_{NUMA}$ and an average latency per sampled remote access event $\frac{l^s_{NUMA}}{E^s_{NUMA}}$ for the whole program or any code region. With a conventional PMU counter, we can collect the absolute number of instructions $I$ executed by the monitored program or any code region. Experimentally, we have found that if $lpi_{NUMA}$ is larger than 0.1 cycle per instruction, the NUMA losses for a program or important code region are significant enough to warrant optimization.

### 7.4 Metric Attribution

To help a user understand NUMA performance losses, our tool attributes metrics using three different approaches.

- **Code-centric** attribution correlates performance metrics to instructions, loops, and functions that have high access latency.

- **Data-centric** attribution associates performance metrics with variables that have high access latency.

- **Address-centric** attribution summarizes a thread’s memory accesses, which is useful for understanding data access patterns.
Each attribution technique highlights different aspects of NUMA performance losses. Together, they provide deep insight into NUMA bottlenecks. We describe these attribution methods in the next two sections.

### 7.4.1 Code- and Data-centric Attribution

Using hardware or software support for address sampling, our tool can accurately attribute costs to both code and data. Our approach for code- and data-centric attribution leverages existing support in HPCToolkit [89], which is described in Chapter 5.

For code-centric attribution, we determine the call path for each address sample by unwinding the call stack. We then associate NUMA metrics with the call path.

For data-centric attribution, we directly associate metrics with variables, including static variables and dynamically allocated heap data. Our tool identifies address ranges associated with static variables by reading symbols in the executable and dynamically loaded libraries. We identify address ranges associated with heap-allocated variables by tracking memory allocations and frees. Our tool attributes each sampled heap variable accesses to the full calling context where the heap variable was allocated.

### 7.4.2 Address-centric Attribution

Address-centric attribution provides insight into memory access patterns of each thread. Such information is needed by application developers to help them adjust data distributions to minimize NUMA overhead. Prior data-centric tools, e.g., [96, 89, 75], identify problematic code regions and data objects, but don’t provide insight into data access patterns to guide optimization. Below, we first describe a naive address-centric attribution strategy and then introduce refinements to make this approach...
For each memory access $m$ to a variable $x$, e.g., an array, we compute the addresses that form the lower and upper bounds of the range accessed by $m$ and update the lower and upper bounds of $x$ accessed for each procedure along the call path to $m$. At analysis time, we plot the upper and lower bounds of the data range accessed by each thread for any variable in any calling context to gain insight into the data access patterns across threads for code executed in that context.

This approach is too simplistic to be useful. Often data ranges are accessed non-uniformly because of loops, conditionals, and indirect references. For some program regions, a hot variable segment may account for 90% of a thread’s accesses, while other segments of this variable only account for 10%. Therefore, instead of computing the minimum and maximum effective addresses for the whole memory range allocated for the variable, we represent a variable’s address range with a sequence of bins, each bin representing a subrange. We treat each bin as a separate synthetic variable that gets its own data- and address-centric attributions. As performance measurements are associated with individual bins, one can easily identify the hot bins of a variable. We only use the access patterns of the hot bins to represent the access patterns of the whole variable. It is worth noting that selecting the number of bins for variables is important. A large number of bins for a variable can show fine-grained hot ranges but may ignore some important patterns. Currently, our tool divides a variable with an address range larger than five pages into five bins by default; one can change this number via an environment variable.

The aforementioned approach for maintaining address ranges merges them online to keep profiles compact. However, different memory accesses experience different
latencies. For that reason, when analyzing access ranges for variables at different levels of abstraction (statement, loop, procedure, and various levels in the call path), one should not give equal weight to access ranges in all contexts. One can use aggregate latency measurements attributed to a context as a guide to identify what program contexts are important to consider for NUMA locality optimization, and then use address range information for those contexts as a guide to understand what changes to data and/or code mappings will be needed to improve NUMA performance.

### 7.5 Pinpointing First Touches

As discussed in Section 7.1, identifying the source code first touching variables associated with NUMA performance losses is essential for optimization. However, manually identifying code performing first touches to variables is often difficult for complex programs. To automatically identify first touches, our tool uses a novel approach that employs page protection in Linux. Our strategy does not require any instrumentation of memory accesses, so it has low runtime overhead.

Figure 7.2 shows how we trap a first touch access to a large variable, e.g., an array. Our tool first installs a SIGSEGV handler before a monitored program begins execution. Then, our tool monitors memory allocations in the program using wrappers for allocation functions. Inside the wrapper, our tool masks off the read and write permission of the allocated memory range between the first and last page boundaries within the variable’s extent. When the monitored executable accesses the protected pages, the OS delivers a SIGSEGV signal. Our tool’s SIGSEGV handler catches the signal and does three things. First, it uses the signal context to perform code-centric attribution. Second, it uses the data address that caused the fault (available in the signal info structure) to perform data-centric attribution. With both code- and data-
Figure 7.2: Identifying the first touch context for each heap variable. Our NUMA profiler applies both code-centric and data-centric attribution for first touches.

centric attribution of the SIGSEGV signal, one knows the location of first touches to every monitored variable. Finally, it restores read and write permissions for the variable’s monitored pages. It is worth noting that multiple threads may initialize a variable concurrently in a parallel loop, so more than one thread may enter the SIGSEGV handler. Thus, multiple threads may concurrently identify first touches and record code- and data-centric attributions. Call paths of first touches to the same variable from different threads are merged post-mortemly.
7.6 Tool Implementation

We implemented support for analyzing NUMA inefficiencies as extensions to HPC-Toolkit [2]—an open-source performance tool for call path profiling of parallel programs. HPCToolkit is described in Section 2.3. In the rest of this section, we describe how we extended HPCToolkit’s measurement, analysis, and presentation tools to support analysis of NUMA performance problems. We refer to our modified version of HPCToolkit for pinpointing and analyzing NUMA bottlenecks as HPCToolkit-NUMA.

7.6.1 Online Profiler

Our extensions to HPCToolkit for NUMA performance analysis perform three tasks. First, they configure PMU hardware for address sampling. We extended HPCToolkit to leverage Perfmon [35] to control PMUs that employ IBS, PEBS and DEAR. Our extensions use Linux Perf_events [139] to configure PMUs for architectures that support MRK and PEBS-LL. For software-based sampling, we extended HPCToolkit to override callbacks for instrumentation hooks inserted for loads and stores by LLVM; these instrumentation callbacks record information each time a predefined threshold of accesses occurs.

Second, HPCToolkit’s hpcrun utility captures these address samples and attributes them to code and data, recording them in augmented calling context trees (CCTs) [89]. The augmented CCT our NUMA extensions record is a mixture of variable allocation paths, memory access call paths, and first touch call paths. Dummy nodes in the augmented CCT separate segments of calling context sequences recorded for different purposes.

Third, the profiler collects NUMA metrics including $M_l$, $M_r$, metrics that show
the number of sampled accesses touching each NUMA domain, latency metrics, and address-centric metrics that summarize each thread’s variable accesses in each subtree of the CCT.

### 7.6.2 Offline Analyzer and Viewer

Adapting HPCToolkit’s `hpcprof` offline profile analyzer for NUMA measurement was trivial. The only enhancement needed was the ability to perform \([\text{min}, \text{max}]\) range computations when merging different thread profiles. Instead of accumulating metric values associated with the same context, \([\text{min}, \text{max}]\) merging requires a customized reduction function.

We added an address-centric view to HPCToolkit’s `hpcviewer` interface to display address-centric measurements for all threads. The view plots the minimum and maximum address accessed for a variable vs. the thread index. The address range for a variable is normalized to the interval \([0, 1]\). The upper right pane in Figure 7.3 shows an example of this view for a heap-allocated variable. In the next section, we show how this novel view provides insight that helps guide optimization for NUMA platforms.

### 7.7 Experiments

We tested HPCToolkit-NUMA on five different architectures to evaluate its functionality using different hardware and software support for address sampling. Table 7.1 shows our choices for events and sampling periods for evaluating each of the address sampling mechanisms. The criteria for choosing events is based on (1) sampling every memory access (not only NUMA-related events or instructions) to avoid biased results for access patterns and (2) sampling all instructions (if possible) to support computing
Sampling mechanisms | Processors | Threads | Events | Sampling periods
---|---|---|---|---
Instruction-based sampling (IBS) | AMD Magny-Cours | 48 | IBS op | 64K instructions
Marked event sampling (MRK) | IBM POWER 7 | 16 | PM_MRK, PM_FROM_L1MISS | 1
Precise event-based sampling (PEBS) | Intel Xeon Harpertown | 8 | INST_RETIRED_ANY | 1000000
Data event address registers (DEAR) | Intel Itanium 2 | 8 | DATA, EAR, CACHE_LAT | 20000
PEBS with load latency (PEBS-LL) | Intel Ivy Bridge | 8 | LATENCY ABOVE_THRESHOLD | 500000
Software-supported IBS (Soft-IBS) | AMD Magny-Cours | 48 | memory accesses | 1000000

Table 7.1 : Configurations of different sampling mechanism on different architectures.

<table>
<thead>
<tr>
<th>Methods</th>
<th>LULESH</th>
<th>AMG2006</th>
<th>Blackholes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBS</td>
<td>295s (+24%)</td>
<td>89s (+37%)</td>
<td>192s (+6%)</td>
</tr>
<tr>
<td>MRK</td>
<td>93s (+5%)</td>
<td>27s (+7%)</td>
<td>132s (+4%)</td>
</tr>
<tr>
<td>PEBS</td>
<td>65s (+45%)</td>
<td>96s (+52%)</td>
<td>82s (+25%)</td>
</tr>
<tr>
<td>DEAR</td>
<td>90s (+7%)</td>
<td>120s (+12%)</td>
<td>73s (+4%)</td>
</tr>
<tr>
<td>PEBS-LL</td>
<td>35 (+6%)</td>
<td>57s (+8%)</td>
<td>67s (+3%)</td>
</tr>
<tr>
<td>Soft-IBS</td>
<td>604s (+200%)</td>
<td>220s (+180%)</td>
<td>270s (+30%)</td>
</tr>
</tbody>
</table>

Table 7.2 : Runtime overhead measurement of HPCToolkit-NUMA. The number outside the parenthesis is the execution time without monitoring and the percentage in the parenthesis is the monitoring overhead. The absolute execution time on different architectures is incomparable because we adjust the benchmark inputs according to the number of cores in the system.

NUMA latency per instruction. For the tests we ran, the sampling period we chose for each event except MRK gives 100–1000 samples per second per thread.* To evaluate the tool, we used four multi-threaded benchmarks: LULESH [77], AMG2006 [79], Blackholes [17], and UMT2013 [78]. These benchmarks are described in detail in Section 2.4.

Some NUMA optimization of LULESH and AMG2006 are described in Chapter 5. Guided by insights from HPCToolkit-NUMA, we were able to develop significantly better NUMA-aware optimizations for both LULESH and AMG2006, advancing the state of the art.

Table 7.2 shows the measurement overhead when running HPCToolkit-NUMA.

---

*Marked event sampling on POWER7 with the fastest sampling rate under user control generates less than 100 samples/second per thread.
with different architectures with different sampling methods. From the table, we can see that the overhead of HPCToolkit-NUMA differs using different sampling methods. Soft-IBS incurs the highest runtime overhead because it is based on software instrumentation; PEBS incurs the second highest overhead because we compensate for off-by-1 attribution by the PMU using online binary analysis to identify the previous instruction, which is difficult for x86 code; IBS has the third highest overhead because it samples all kinds of instructions and its sampling rate is high; and other sampling methods incur very low runtime overhead. At runtime, the aggregate runtime footprint of HPCToolkit-NUMA’s data structures was less than 40MB for any sampling method on any of the architectures.

In the case studies that follow, we only show measurements obtained using IBS and MRK because HPCToolkit-NUMA can provide similar analysis results using any sampling method. Moreover, IBS is one of the two PMU hardware types that supports the $lpi_{NUMA}$ metric, while MRK can show how we analyze NUMA bottlenecks using NUMA metrics we derived without the help of latency information. For experiments using IBS, we use a system with four 12-core AMD Magny-Cours processors. Overall, the system has 48 cores and 128GB memory, which is evenly divided into eight NUMA domains. For experiments using MRK, we use a system with four eight-core POWER7 processors. Overall, the system has 128 SMT hardware threads and 64GB memory. In this study, we consider each socket a NUMA domain. We run LULESH, AMG2006 and Blackholes on all hardware threads; we run UMT2013 with 32 threads because its standard input size is limited to 32 threads.
Figure 7.3: Identifying NUMA bottlenecks in LULESH using code-centric, data-centric and address-centric attributions.

7.7.1 LULESH

We first measure LULESH with IBS on the AMD machine. Figure 7.3 shows the results of our NUMA performance analysis result displayed in a modified version of HPC Toolkit’s hpcviewer, which is described in Section 2.3. Unlike the original hpcviewer, The top right pane shows the address-centric view, which represents memory ranges accessed by individual threads. The bottom left pane shows the program structure of synthetic CCTs. Annotations in this pane show a mixture of call paths in the CCT.

†It would be better to perform this correction during post-mortem analysis.
The overall program’s NUMA latency per instruction ($lpi_{NUMA}$) is 0.466. This is significantly larger than our 0.1 rule of thumb, which means the NUMA problems in LULESH are significant enough to warrant optimization. We first investigate the heap-allocated variables and then other kinds of variables. The heap-allocated variables have a $lpi_{NUMA}$ of 11.7 and 74.2% of the total latency is caused by remote NUMA domain accesses. We drill down the call path in the bottom left pane for the call sites (operator new[] in the figure) of variable allocations, discovering three variables with more than 8% of total latency caused by NUMA accesses. One can identify the variable names from the source code pane by clicking their allocation sites (marked as 2160, 2164 and 2159 respectively to the left of operator new[]).

Here, we study the variable $z$, which causes the most significant NUMA losses.

Overall, $z$ accounts for 11.3% of the total latency caused by remote accesses. We observe two facts (1) $M_r$ (labeled as NUMA_MISMATCH in the metric pane) is roughly seven times of $M_l$ (labeled as NUMA_MATCH in the metric pane); and (2) all accesses to $z$ come from NUMA domain 0 ($NUMA_{NODE}0$ equals to the sum of $M_l$ and $M_r$). Therefore, we infer that pages for $z$ are all allocated in NUMA domain 0 but accessed by threads in other domains. The top right pane in Figure 7.3 plots the min and max addresses accessed in $z$ by each thread. From the figure, we can see that other than thread 0, each thread touches a subset of $z$. Threads with higher ranks touch higher address intervals in $z$. Visualizing the results of address sampling provides the key insight about how to adjust the layout of heap data to improve performance, namely by co-locating data and computation upon it in the same NUMA domain.

Based on this address-centric view, it is clear that one could improve the NUMA performance of LULESH by dividing the memory range allocated for $z$ into eight
continuous regions, segmented by rectangles shown in the top right pane of Figure 7.3. One should allocate each block to the appropriate NUMA domain so it will be co-located with the threads that access it. We implemented this strategy by adjusting the code that first touches \( z \), identified by our tool, and shown in the top left pane of Figure 7.3. This optimization exploits the higher bandwidth and lower latency of local memory. It also reduces the bandwidth consumed between NUMA domains. We similarly optimize other heap-allocated variables, including \( x, y, xd, yd \) and \( zd \).

LULESH also makes heavy use of a stack-allocated variable \texttt{nodelist}. Since our tool does not currently provide detailed NUMA analysis of stack data, we modified the source code for the program to declare \texttt{nodelist} as a static variable. HPCToolkit-NUMA’s data-centric analysis shows that \texttt{nodelist} accounts for 20.3% of total latency caused by remote accesses and 13.3% mismatching of memory accesses (\( M_r \)). There is high \( lpi_{\text{NUMA}} \) associated with \texttt{nodelist}, meaning that it has high NUMA latency that warrants optimization. The \( M_r \) metric associated with \texttt{nodelist} is about seven times as large as \( M_l \) and all accesses come from NUMA domain 0, which means that \texttt{nodelist} is initialized by the master thread but accessed by worker threads of other NUMA domains in parallel. Address-centric analysis identifies that \texttt{nodelist} has the same access pattern per thread as \( z \) does shown in Figure 7.3. Like optimization for \( z \), such an access pattern reveals that a block-wise distribution of pages allocated for \texttt{nodelist} would be appropriate, as before for \( z \).

Guided by our tool, the block-wise data distribution we implemented for both heap-allocated and stack variables, we were able to speedup LULESH by 25% on our AMD system. Using with the page interleaving strategy suggested by our prior work described in Section 5.3.3 gave only a 13% improvement over the original code not tuned for NUMA architectures.
Measuring LULESH with MRK on POWER7 showed similar NUMA problems. 66% of L3 cache misses access remote memory. Heap allocated arrays, such as x, y, z, xd, yd and zd, account for 65% of remote accesses, while the stack variable nodelist accounts for 31%. On our POWER7 system, HPCToolkit-NUMA’s address-centric view showed that these variables have access patterns similar to those we observed on our AMD system. Using a block-wise page distribution for these variables improved execution time for LULESH by 7.5% on our POWER7 system. In contrast, using an interleaved page allocation (as suggested by prior work described in Section 5.3.3) degraded performance on POWER7 by 16.4%. \(^\dagger\)

### 7.7.2 AMG2006

We ran AMG2006 with 48 threads on our AMD system, measuring it using IBS. HPCToolkit-NUMA showed that AMG2006 has a \(lpi_{NUMA}\) of more than 0.92, which means it has significant NUMA problems (more severe than LULESH) and worthy of investigation. The heap-allocated variables of AMG2006 account for 61.8% of the total memory latency caused by remote accesses. By looking at heap variable allocation call paths, we identified one problematic variable \(RAP\_\text{diag\_data}\). \(RAP\_\text{diag\_data}\) accounts for 18.6% of total latency, with a \(lpi_{NUMA}\) of 15.9 and 8.1% of total \(M_r\). By examining the sampled accesses and the first-touch access to \(RAP\_\text{diag\_data}\), we found that \(RAP\_\text{diag\_data}\) was allocated and initialized by the master thread but accessed by all worker threads in other NUMA domains.

The address-centric view in Figure 7.4 shows the access patterns of \(RAP\_\text{diag\_data}\) across all 48 threads aggregated over the whole program. However, these threads do

\(^\dagger\)The NUMA optimization for LULESH benchmark may not apply to the real LULESH application code because its input data can be highly irregular. Our optimization for LULESH benchmark aims to validate our tool, not improve the real LULESH code.
not show an obvious access pattern that can guide page distribution for this variable. We further investigate threads’ access patterns for RAP_diag_data in individual OpenMP parallel regions rather than the whole program. The most interesting parallel region shown in the call path is hypre_boomerAMGRelax_omp, which accounts for 74.2% (13.8/18.6) of NUMA access latency caused by RAP_diag_data. Figure 7.5 shows the access patterns of RAP_diag_data in this parallel region. Obviously, threads have a regular access pattern of RAP_diag_data in this parallel region. Because accesses in hypre_boomerAMGRelax_omp dominate the costs of accessing RAP_diag_data, we can use this access pattern to direct the data distribution. Like optimization for LULESH, we apply block-wise distribution at the first touch place identified by our tool to evenly allocate RAP_diag_data across the NUMA domains.

If we analyze the source code through the code-centric attribution, we find that accesses to RAP_diag_data in hypre_boomerAMGRelax_omp use the values of another array as indices (i.e., RAP_diag_data[A_diag_i[i]]), leading to indirect memory accesses. Without our address-centric analysis analysis, one cannot determine where data layout changes are needed and how to refine them to improve performance.

We examined other hot variables and found another NUMA bottleneck:
Figure 7.6: Address-centric view showing the overall access patterns of \texttt{RAP.diag\_j} in AMG2006 across all threads.

\texttt{RAP.diag\_j} accounts for 10.6% of total latency caused by NUMA accesses. Figure 7.6 and Figure 7.7 show its address-centric analysis results by considering all accesses and accesses in the most significant parallel region respectively. Obviously, the access patterns on the bottom are much more regular than the one on the top. As memory accesses in this parallel region account for 73.6% of total latency for \texttt{RAP.diag\_j} in the whole program, we use its regular access pattern to allocate pages of \texttt{RAP.diag\_j} in a block-wise fashion at its first touch location.

Besides these two variables, there are other three heap-allocated variables suffering from high remote access latency. According to access patterns from address-centric analysis, one of them can be optimized using block-wise distribution as for \texttt{RAP.diag.data} and \texttt{RAP.diag.j}. The other two show that each thread accesses the whole range of the variable, leading to an optimization of using interleaved page allocation. Our optimizations achieve a 51% reduction in the running time of the solver phase of AMG2006. In production codes that employ this software, the running time of the solver is most important. Without guidance from our address-centric analysis, our prior NUMA optimization of AMG2006 shown in Section 5.3.1 used interleaved allocation for every problematic variable, which only improved the solver phase per-
formance of AMG2006 by 36%.

7.7.3 Blackscholes

We measured Blackscholes on our AMD system using IBS. HPCToolkit-NUMA shows a much smaller $lpi_{NUMA}$ value (0.035 cycle per instruction) than the threshold (0.1) over the entire program, indicating that Blackscholes would not benefit from NUMA optimization. To validate this assessment, we eliminated NUMA bottlenecks in Blackscholes and showed that this optimization barely improved the program’s performance.

HPCToolkit-NUMA identified that heap-allocated variables account for 66.8% of total latency caused by NUMA accesses and 51.6% of the latency associated with the variable buffer. With the values of $M_l$ and $M_r$, together with the data source metrics, we identified that buffer is allocated in only one NUMA domain by the master thread and evenly accessed by all threads in the system.
(a) Original access pattern.

(b) Improved access pattern.

Figure 7.9: Memory access patterns across threads in BlackScholes.

HPCToolkit-NUMA’s address-centric analysis in Figure 7.8 shows a regular access pattern across all threads. Each thread touches a sub-range of buffer in an ascending order, with large overlaps. To understand why the program reveals such pattern, we analyzed the source code. The top of Figure 7.9 shows the memory layout for buffer. The program sets five pointers to point different sections of buffer. All threads access to each section in parallel, leading to the access pattern shown in Figure 7.9a. According to our address-centric analysis, the three example threads in the figure touch the address ranges of (0x100, 0x700), (0x200, 0x800) and (0x300, 0x900) respectively, matching the pattern revealed by address-centric results shown in Figure 7.8.

As threads show non-local accesses to buffer, we regroup these sections into an array of structures as shown in Figure 7.9b. With this optimization, the three example threads touch (0x100, 0x300), (0x400, 0x600), (0x700, 0x900) respectively, with no overlap.

To allocate data block-wise, we changed the buffer initialization loop that our tool identified as the first touch location. Originally, only the master thread initializes buffer. We parallelized the initialization loop using OpenMP to make sure that
do c=1,nCorner
  do ig=1,Groups
    source=Z%STotal(ig,c)+Z%STime(ig,c,Angle)
  enddo
enddo

Figure 7.10: A loop kernel in UMT2013 that has many remote accesses to STime.

each thread first touches its own data. With this optimization, there is no longer any latency related to buffer caused by remote accesses.

Although we largely eliminated the NUMA latency in the program by co-locating data and computation, the execution time of Blackscholes improves less than 0.1%. The trivial improvement proves that our derived metric $lpi_{NUMA}$ reflects the severity of NUMA problems. One can estimate potential gains from NUMA optimization by examining $lpi_{NUMA}$.

7.7.4 UMT2013

We ran UMT2013 on our POWER7 system with 32 threads, sampling instructions that cause L3 data cache misses. We bounded each thread to each hardware core in each of four NUMA domains. According to $M_l$ and $M_r$, HPCToolkit-NUMA showed that 86% L3 cache misses lead to remote memory accesses and 47% of remote accesses due to the references of heap allocated variables.

Using HPCToolkit-NUMA, we identified the allocation call path of a hot variable – STime, which is a three-dimensional array that accounts for 18.2% of total remote accesses. Code-centric analysis in HPCToolkit-NUMA associates all remote accesses to STime with the reference shown in Figure 7.10. The reason for the high remote access ratio is that STime is allocated in one NUMA domain but accessed by
threads in all NUMA domains. With address-centric analysis, HPCToolkit-NUMA identifies that $\text{STime}$ has a staggered access pattern across threads, similar to the variable $\text{buffer}$ in BlackScholes, shown in Figure 7.8. A deep analysis of the source code showed that the nested loop iterating over $\text{STime}$ shown in Figure 7.10 is in an OpenMP parallel region. Two-dimensional planes of $\text{STime}$ indexed by $\text{Angle}$ are assigned to threads in a round-robin fashion. To ensure the data is co-located with its computation, we parallelized the initialization loop of $\text{STime}$ identified by first touch analysis. This strategy has each thread initialize the part of $\text{STime}$ that it accesses in the computation stage. This optimization eliminates most remote accesses to $\text{STime}$ and yielded a 7% speedup for the program as a whole.

### 7.8 Discussion

In this chapter, we present a profiling tool that identifies and analyzes NUMA bottlenecks in multi-threaded programs and helps guide program performance tuning by providing new metrics and insights into data access patterns. Using PMU support in modern microprocessors, our measurement-based tool can gather the information it needs with low runtime overhead. We demonstrate the utility of our tool and the information it provides to optimize four well-known benchmarks. While one code didn’t warrant NUMA optimization (according to our metrics) or benefit significantly when it was applied anyway, our tool delivered insight and guidance that enabled us to significantly improve the performance of the other three codes.

In our experiments using different hardware for address sampling, we observed that not all mechanisms are equally-suited for our analysis. Although both IBS and MRK sample instructions, IBS samples all kinds of instructions, so one needs to filter out samples not of interest in software, which adds extra overhead. With IBS it is
trivial to compute the load/store fraction in the whole instruction stream to assess the performance impact of memory instructions. In contrast, MRK can only sample instructions causing specific events (such as L3 cache misses or remote accesses). Consequently, MRK can highlight problematic memory instruction with low overhead. DEAR, PEBS and PEBS-LL sample events. PEBS and PEBS-LL can directly sample NUMA events, while DEAR does not support NUMA events. Both instruction sampling and event sampling can effectively identify problematic memory accesses. Finally, IBS and PEBS-LL can measure latency for sampled load instructions. This information can be used to derive the metrics described in Section 7.3.

Our future work is five-fold. First, we plan to add full support for monitoring stack variables instead of requiring them to be changed to static or heap allocated ones for detailed measurements. Second, we plan to extend our tool to analyze more complex access patterns. Third, we plan to collect trace-based measurements to study time-varying NUMA patterns in addition to profiles. Fourth, we plan to augment hpcviewer with a new view to better present code-and data-centric measurements. Finally, our strategy for pinpointing first touches is only implemented at present for heap-allocated variables. We plan to extend it for static variables by protecting their pages when the executable or libraries are loaded before execution begins.
Chapter 8

Conclusions

This chapter shows some conclusions drawn from this dissertation. Section 8.1 summarizes the contributions of this dissertation; Section 8.2 shows some open problems that are opportunities for future work.

8.1 Summary of Contributions

This dissertation demonstrates that with lightweight sampling mechanisms, one can build effective and efficient tools that analyze performance losses in programs running on modern parallel architectures with many hardware threads, deep memory hierarchies, and NUMA systems. By applying lightweight data collection, novel metrics, and new metric attribution, this dissertation addresses the difficulty of identifying performance bottlenecks and providing useful insights to guide optimization.

Lightweight data collection  Lightweight data collection is essential for monitoring large-scale parallel programs, incurring little intrusion to the native execution. This dissertation shows the feasibility of collecting performance samples and their calling contexts with low overhead. On the one hand, I have shown a variety of sampling techniques, supported by traditional hardware counters, modern PMUs, and software, can collect samples with low overhead. On the other hand, I have shown two methods that can efficiently determine the full calling context of samples: one is call stack unwinding, based on the dynamic binary analysis; the other is a hy-
brid method, combining call stack unwinding and function call/return monitoring. Lightweight sampling with full calling contexts forms the basic performance data pool that for any further analysis.

**Novel metrics** This dissertation has shown that new metrics derived from raw data are important for identifying and quantifying performance bottlenecks. First, decomposing timing metrics into *work, idleness*, and *lockwait* is useful to understand the bottlenecks in OpenMP programs, shown in Chapter 4. These metrics are helpful to assess the performance gains due to optimization. Second, Chapters 5, 6 and 7 illustrate the utility of latency metrics. These latency metrics quantify the severity of memory bottlenecks and assess their influence to the whole program’s performance degradation. These metrics focus optimization efforts on memory bottlenecks that can lead to significant improvement after optimization.

**New metric attribution** Attributing metrics is essential to understand root causes of performance bottlenecks and provide optimization strategies. This dissertation proposes three novel metric attribution methods: code-centric attribution for root causes, data-centric attribution, and address-centric attribution. First, code-centric attribution associates performance losses in OpenMP programs with their root causes. It identifies the code regions that suffers from load imbalance and over synchronization. Second, data-centric attribution associates performance metrics with variables. Combining code-centric attribution, it can pinpoint problematic data access patterns and suggest optimal data layout transformation. Finally, address-centric attribution is developed for choosing best data distribution across NUMA domains that can maximize local accesses and balance memory requests.
8.2 Open Problems

In Sections 4.4, 5.4, 6.5, and 7.8, I discussed straight-forward extensions of the approaches that I described. In this section, I present some high-level ideas for future work. I plan to extend approaches described in this dissertation in four ways: generalizing multithreading support for performance analysis beyond OpenMP, improving data-centric profiling to provide additional performance insights, providing high-level feedback to guide optimization, and augmenting analysis for performance problems in non-HPC programs.

Performance analysis for multithreading In the future, we plan to apply our knowledge obtained from standardizing OMPT to design standard tools APIs for other multithreading or programming models for accelerated computing, such as OpenCL [128] and OpenACC [28].

Data-centric tracing Data-centric analysis described in this dissertation is coarse-grained on the level of aggregate profiles. Because access patterns may change in different phases of a program, the best data layout may not be the same for the whole program execution. In the future, I plan to add a timestamp for each sampled memory access, known as tracing, and apply dynamic layout transformation to match changes of access patterns.

High-level optimization guidance Our approaches implemented in HPCToolkit currently focus on data collection and attribution. Although raw performance data provide deep insight into program’s performance, they cannot easily guide code optimizations, especially for non-expert users. Therefore, it is important to provide high-level analysis to bridge the gap of raw data and user’s knowledge. PerfEx-
pert [25, 124, 49] and ThreadSpotter [115] are two tools that attempt to address this problem. These tools report high-level optimization suggestions by matching a program’s performance data with predefined patterns. In the future, I plan to use data mining and machine learning methods to study more performance patterns, and enclose automatic code transformation for optimization in HPCToolkit framework.

**Performance issues beyond HPC programs**  In the future, I plan to develop new analysis methods for non-HPC programs, such as Big Data analytics programs, server workloads, and mobile applications. These kinds of programs have performance issues beyond the memory sub-system, such as network and disk I/O. Moreover, energy consumption is a key problem that needs consideration. For that reason, power profiling is an interesting topic.
Bibliography


Programming Language Design and Implementation, pages 212–223, Montreal, Quebec, Canada, June 1998.


