I dedicate this to my family for all their love and support.
ACKNOWLEDGMENTS

This work is supported in part by the U.S. Department of Defense. Thanks go to my advisor, Dr. Alan D. George, for his advice and patience over the course of this research and my committee members, Dr. Herman Lam, Dr. Gregory Stitt, and Dr. Beverly Sanders, for their time and effort. I would also like to acknowledge current and former members of the Unified Parallel C group at UF, Max Billingsley III, Adam Leko, Hans Sherburne, Bryan Golden, Armando Santos, and Balaji Subramanian for their involvement in the design and development of the Parallel Performance Wizard system. Finally, I would like to express thanks to Dan Bonachea and the Unified Parallel C group members at U.C. Berkeley and Lawrence Berkeley National Laboratory for their helpful suggestions and cooperation.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>10</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>13</td>
</tr>
</tbody>
</table>

## CHAPTER

1. INTRODUCTION .......................... 14

2. OVERVIEW OF PARALLEL PROGRAMMING MODELS AND PERFORMANCE TOOLS ................. 16
   2.1 Parallel Programming Models .............................................. 16
   2.2 Performance Tools ......................................................... 18

3. BACKGROUND RESEARCH FINDINGS ................................................ 22

4. GENERAL-PURPOSE FRAMEWORK FOR PARALLEL APPLICATION PERFORMANCE ANALYSIS .......... 25
   4.1 Parallel Programming Event Model ........................................... 28
      4.1.1 Group-Related Operations ............................................... 32
      4.1.2 Data Transfer Operations ............................................... 35
      4.1.3 Lock, Wait-On-Value, and Locally Executed Operations .............. 38
      4.1.4 Implementation Challenges and Strategies ............................. 39
   4.2 Instrumentation and Measurement ........................................... 41
      4.2.1 Overview of Instrumentation Techniques ................................ 41
      4.2.2 The Global-Address-Space Performance Interface ........................ 42
      4.2.3 GASP Implementations ................................................... 46
   4.3 Automatic Analysis ......................................................... 46
   4.4 Data Presentation ............................................................ 47
   4.5 PPW Extensibility, Overhead, and Storage Requirement ........................ 48
      4.5.1 PPW, TAU, and Scalasca Comparision .................................... 51
      4.5.2 PPW Tool Scalability ..................................................... 51
   4.6 Conclusions ................................................................. 52

5. SYSTEM FOR AUTOMATIC ANALYSIS OF PARALLEL APPLICATIONS ......................... 55
   5.1 Overview of Automatic Analysis Approaches and Systems ...................... 56
   5.2 PPW Automatic Analysis System Design ....................................... 59
      5.2.1 Design Overview .......................................................... 60
      5.2.2 Common-Bottleneck Analysis .............................................. 61
      5.2.2.1 Bottleneck detection .................................................. 65
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Theoretical event model for parallel programs</td>
<td>30</td>
</tr>
<tr>
<td>4-2</td>
<td>Mapping of UPC, SHMEM, and MPI 1.x constructs to generic operation types</td>
<td>31</td>
</tr>
<tr>
<td>4-3</td>
<td>GASP events for non-blocking UPC communication and synchronization</td>
<td>43</td>
</tr>
<tr>
<td>4-4</td>
<td>Profiling/tracing file size and overhead for UPC NPB 2.4 benchmark suite</td>
<td>48</td>
</tr>
<tr>
<td>4-5</td>
<td>Profiling/tracing file size and overhead for SHMEM APSP, CONV, and SAR</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>application</td>
<td></td>
</tr>
<tr>
<td>4-6</td>
<td>Profiling/tracing file size and overhead for MPI corner turn, tachyon, and SAR</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>application</td>
<td></td>
</tr>
<tr>
<td>4-7</td>
<td>Profiling/tracing file size and overhead comparison of PPW, TAU, and Scalasca</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>tools for a 16-PE run of the MPI IS benchmark</td>
<td></td>
</tr>
<tr>
<td>4-8</td>
<td>Profiling/tracing file size and overhead for medium-scale UPC NPB 2.4 benchmark suite runs</td>
<td>52</td>
</tr>
<tr>
<td>5-1</td>
<td>Summary of existing PPW analyses</td>
<td>62</td>
</tr>
<tr>
<td>5-2</td>
<td>Common-bottleneck patterns currently supported by PPW and data needed to perform cause analysis</td>
<td>63</td>
</tr>
<tr>
<td>5-3</td>
<td>Example resolution techniques to remove parallel bottlenecks</td>
<td>73</td>
</tr>
<tr>
<td>5-4</td>
<td>Sequential analysis speed of NPB benchmarks on workstation</td>
<td>80</td>
</tr>
<tr>
<td>5-5</td>
<td>Analysis speed of NPB benchmarks on workstation</td>
<td>82</td>
</tr>
<tr>
<td>5-6</td>
<td>Analysis speed of NPB benchmarks on Ethernet-connected cluster</td>
<td>84</td>
</tr>
<tr>
<td>6-1</td>
<td>Performance comparison of various versions of UPC and SHMEM SAR programs</td>
<td>101</td>
</tr>
<tr>
<td>A-1</td>
<td>User function events</td>
<td>115</td>
</tr>
<tr>
<td>A-2</td>
<td>Memory allocation events</td>
<td>115</td>
</tr>
<tr>
<td>A-3</td>
<td>Exit events</td>
<td>118</td>
</tr>
<tr>
<td>A-4</td>
<td>Synchronization events</td>
<td>118</td>
</tr>
<tr>
<td>A-5</td>
<td>Work-sharing events</td>
<td>119</td>
</tr>
<tr>
<td>A-6</td>
<td>Library-related events</td>
<td>120</td>
</tr>
<tr>
<td>A-7</td>
<td>Blocking shared variable access events</td>
<td>121</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A-8</td>
<td>Non-blocking shared variable access events</td>
<td>121</td>
</tr>
<tr>
<td>A-9</td>
<td>Shared variable cache events</td>
<td>122</td>
</tr>
<tr>
<td>A-10</td>
<td>Collective communication events</td>
<td>123</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2-1</td>
<td>Measure-modify performance analysis approach of performance tool</td>
<td>19</td>
</tr>
<tr>
<td>4-1</td>
<td>PPW-assisted performance analysis process from original source program to revised (optimized) program</td>
<td>26</td>
</tr>
<tr>
<td>4-2</td>
<td>Generic-operation-type abstraction to facilitate the support for multiple programming models</td>
<td>27</td>
</tr>
<tr>
<td>4-3</td>
<td>Framework of Parallel Performance Wizard organized with respect to stages of experimental measurement and model dependency (multi-boxed units are model-dependent)</td>
<td>28</td>
</tr>
<tr>
<td>4-4</td>
<td>Events for group synchronization, group communication, initialization, termination, and global memory allocation operations (TS = Timestamp. Only a few of PE Y's events are shown to avoid clutter)</td>
<td>34</td>
</tr>
<tr>
<td>4-5</td>
<td>Events for one-sided communication and synchronization operations</td>
<td>37</td>
</tr>
<tr>
<td>4-6</td>
<td>Events for two-sided communication and synchronization operations</td>
<td>38</td>
</tr>
<tr>
<td>4-7</td>
<td>Events for (a) lock mechanisms, wait-on-value and (b) user-defined function/region, work-sharing, and environment-inquiry operations</td>
<td>39</td>
</tr>
<tr>
<td>4-8</td>
<td>Interaction of PGAS application, compiler, and performance tool in GASP-enabled data collection</td>
<td>43</td>
</tr>
<tr>
<td>4-9</td>
<td>Specification of gasp_event_notify callback function</td>
<td>44</td>
</tr>
<tr>
<td>4-10</td>
<td>(a) Load-balancing analysis visualization for CG 256-PE run, (b) Experimental set comparison chart for Camel 4-, 8-, 16-, and 32-PE runs</td>
<td>49</td>
</tr>
<tr>
<td>4-11</td>
<td>Annotated screenshot of new UPC-specific array distribution visualization showing physical layout of a 2-D 5x8 array with block size 3 for an 8-PE system</td>
<td>50</td>
</tr>
<tr>
<td>4-12</td>
<td>(a) Data transfers visualization showing communication volume between processing PEs for 256-PE CG benchmark tracing mode run, (b) Zoomed-in Jumpshot view of 512-PE MG benchmark</td>
<td>53</td>
</tr>
<tr>
<td>5-1</td>
<td>Tool-assisted automatic performance analysis process</td>
<td>56</td>
</tr>
<tr>
<td>5-2</td>
<td>PPW automatic analysis system architecture</td>
<td>61</td>
</tr>
<tr>
<td>5-3</td>
<td>Example analysis processing system with 3 processing units showing the analyses each processing unit performs and raw data exchange needed between processing units</td>
<td>63</td>
</tr>
<tr>
<td>5-4</td>
<td>Analysis process flowchart for a processing unit in the system</td>
<td>64</td>
</tr>
</tbody>
</table>
5-5 Barrier-redundancy analysis ........................................ 70
5-6 Frequency analysis .................................................... 72
5-7 PPW analysis user interface ......................................... 74
5-8 Annotated PPW scalability-analysis visualization .............. 75
5-9 PPW revision-comparison visualization ............................ 76
5-10 Annotated PPW high-level analysis visualization ............... 76
5-11 PPW event-level load-balance visualization ...................... 77
5-12 Annotated PPW analysis table visualization ...................... 78
5-13 Annotated PPW analysis summary report ......................... 79
5-14 A common use case of PPW where user transfers the data from parallel system to workstation for analysis .................................................. 80
5-15 Analysis workflow for the (a) sequential prototype (b) threaded prototype (c) distributed prototype ................................. 81
5-16 A use case of PPW where analyses are performed on the parallel systems .......................................................... 83
5-17 Memory usage of PPW analysis system on cluster ............... 84
6-1 Productivity study result showing (a) method with more bottlenecks identified (b) preferred method for bottleneck identification (c) preferred method for program optimization ................................................. 88
6-2 Annotated PPW Tree Table visualization of original FT showing code regions yielding part of performance degradation ............................. 89
6-3 Annotated Jumpshot view of original FT showing serialized nature of upc_memget at line 1950 ........................................ 90
6-4 PPW Tree Table for modified FT with replacement asynchronous bulk transfer calls ..................................................... 91
6-5 Multi-table analysis visualization for FT benchmark with annotated Jumpshot visualization .................................................. 93
6-6 Overview of Synthetic Aperture Radar algorithm ................... 94
6-7 Performance breakdown of UPC SAR baseline version run with 6, 12, and 18 computing PEs annotated to show percentage of execution time associated with barriers .................................................. 96
6-8 Timeline view of UPC SAR baseline version run with 6 computing PEs annotated to highlight execution time taken by barriers

6-9 High-level analysis visualization for the original version (v1) of SAR application with load-imbalance issue

6-10 Observed execution time for various UPC and SHMEM SAR reversions

6-11 Timeline view of UPC SAR flag synchronization version executed on system with 6 computing PEs annotated to highlight wait time of flags

6-12 High-level analysis visualization for the F2M version (v5) of SAR application with no major bottleneck
Developing a high-performance parallel application is difficult. Given the complexity of high-performance parallel programs, developers often must rely on performance analysis tools to help them improve the performance of their applications. While many tools support analysis of message-passing programs, tool support is limited for applications written in other programming models such as those in the partitioned global-address-space (PGAS) family, which is of growing importance. Existing tools that support message-passing models are difficult to extend to support other parallel models because of the differences between the paradigms. In this dissertation, we present work on the Parallel Performance Wizard (PPW) system, the first general-purpose performance system for parallel application optimization. The complete research is divided into three parts. First, we introduce a model-independent PPW performance tool framework for parallel application analysis. Next, we present a new scalable, model-independent PPW analysis system designed to automatically detect, diagnose, and possibly resolve bottlenecks within a parallel application. Finally, we discuss case studies to evaluate the effectiveness of PPW and conclude with contributions and future directions for the PPW project.
CHAPTER 1
INTRODUCTION

Parallel computing has emerged as the dominant high-performance computing paradigm. To fully support concurrent execution, many parallel computer systems such as the symmetric multiprocessor system, computer cluster, computational grid, and multi-core machines have been developed. In addition, to take advantage of these parallel systems, a variety of parallel programming models such as Open Multi-Processing (OpenMP), Message-Passing Interface (MPI), Unified Parallel C (UPC), and SHared MEMory (SHMEM) library have been created over the years. Using these technologies, programmers from many scientific and commercial fields are able to develop parallel applications that solve difficult problems more quickly or solve complex problems previously thought to be impossible.

Unfortunately, due to the added complexity of the parallel systems and programming models, parallel applications are more difficult to write than sequential ones and even harder to optimize for performance. Discovery and removal of performance issues require extensive knowledge on programmers’ part about the execution environment and involve a significant amount of effort. Programmers often must undergo a non-trivial, iterative analysis and optimization process that is cumbersome to perform manually in order to improve the performance of their applications. To facilitate this unwieldy process, many parallel performance analysis tools (henceforth refers to as performance tool) were developed that support a variety of parallel systems and programming models.

Among the available parallel programming models, MPI has received the majority of performance tool research and development as it remains the most well-known and widely used. Most existing parallel performance tools support MPI program analysis to some degree but are limited in supporting other parallel models such as OpenMP and those in the partitioned global-address-space (PGAS) family. While efforts have been made to improve support for these newer models, the progress has not kept up with the...
demand. Since most existing tools were specifically designed to support a particular model (i.e., MPI), they became too tightly coupled with that model, and as a result, require a significant amount of effort on the developers’ part to add new model support.

In this dissertation, we outline our work toward the Parallel Performance Wizard (PPW) system. The goal is to research and develop a general-purpose performance tool infrastructure that readily supports multiple parallel programming models and to develop advance techniques to enhance tool usability. The remainder of this document is organized as follows. In Chapter 2, we provide an overview of performance tools and parallel programming models. In Chapter 3, we describe our research methodology as well as some background research findings that shaped the development of the PPW infrastructure. In Chapter 4, we present the PPW framework and provide experimental results for a functional PPW tool that supports UPC, SHMEM, and MPI. In Chapter 5, we introduce a new automatic analysis system developed to enhance the usability of the PPW tool and provide experimental results for the sequential, threaded, and distributed versions of this system. In Chapter 6, we discuss case studies to validate the framework and techniques presented and conclude the document in Chapter 7.
CHAPTER 2
OVERVIEW OF PARALLEL PROGRAMMING MODELS AND
PERFORMANCE TOOLS

In this chapter, we provide an overview of parallel programming models and performance tools. To avoid confusion, the term processing element (PE) is used to reference a system component (e.g., a node, a thread) that executes a stream of instructions.

2.1 Parallel Programming Models

In this section, we provide an overview of parallel programming models and the three models directly relevant to this research, namely MPI, UPC, and SHMEM.

A parallel programming model is a collection of software technologies that allows programmers to explicitly express parallelisms and orchestrate interactions among PEs. The goal is to facilitate programmers in turning parallel algorithms into executable applications on parallel computers. Parallel programming models are generally categorized by how memory is used. In the shared-memory model (e.g., OpenMP, explicit threading libraries), each PE has direct access to a shared memory space and communication between PEs is achieved by reading and writing of variables that reside in this shared memory space. In the message-passing model (e.g., MPI), each PE has access only to its local memory. A pair of PEs communicates by sending and receiving messages to each other which transfers the data from the local memory of sender to the local memory of the receiver. Finally, the partitioned global-address-space (PGAS) model (e.g., UPC, SHMEM) presents the programmer with a logical global memory space divided into two parts: a private portion local to each PE and a global portion which can be physically partitioned among the PEs. PE communicates with each other by reading and writing the global portion of the memory via the use of put and get operations. In terms of implementation, these models are realized either as libraries, as sequential language extensions, or as new parallel languages.
Message Passing Interface (MPI) is a communication library used to program parallel computers with the goals of high performance, scalability and portability [1]. MPI has become the de facto standard for developing high-performance parallel applications; virtually every existing parallel system provides some form of support for MPI application development. There are currently two versions of the standard: MPI-1 (first standardized in 1994) that uses purely matching send and receive pairs for data transfer and provides routines for collective communication and synchronization, and MPI-2 (a superset of MPI-1, first standardized in 1996) which includes additional features such as parallel I/O, dynamic process management, and some remote memory access (put and get) capabilities.

Unified Parallel C (UPC) is an explicit parallel extension of the ANSI C language developed beginning in the late 1990s based on experience with several earlier parallel C-based programming models [2]. UPC exposes the PGAS abstraction to the programmer by way of several language and library features, including specially typed (shared) variables for declaring and accessing data shared among PEs, synchronization primitives such as barriers and locks, a number of collective routines, and a unique, affinity-aware work sharing construct (`upc forall`). The organization responsible for the continuing development and maintenance of the UPC language is a consortium of government, industry, and academia, which released the latest UPC specification version 1.2 in June 2005. This specification has been implemented in the form of vendor compilers, including offerings from HP and IBM, as well as open-source compilers such as Berkeley UPC [3] and the reference Michigan UPC. These provide for UPC support on a number of HPC platforms, including SMP systems, super-computers such as the Cray XT series, and Linux clusters using a variety of commodity or high-speed interconnects.

The SHared MEMory (SHMEM) library essentially provides the shared-memory abstraction typical of multi-threaded sequential programs to developers of high-performance parallel applications [4]. First created by Cray Research for use on the Cray T3D supercomputer and now trademarked by SGI, SHMEM allows PEs to read and write
all globally declared variables, including those mapped to memory regions physically
located on other PEs. SHMEM is distinct from a language such as UPC in that it
does not provide intrinsically parallel language features; instead, the shared memory
model is supported by way of a full assortment of API routines (similar to MPI). In
addition to the fundamental remote memory access primitives (get and put), SHMEM
provides routines for collective communication, synchronization, and atomic memory
operations. Implementations of the library are primarily available on systems offered by
SGI and Cray, though versions also exist for clusters using interconnects such as Quadrics
(recently went out of business). At present time, no SHMEM standardization exists so
different implementations tend to support a different set of constructs providing similar
functionalities. However, an effort to create an OpenSHMEM standard [5] is currently
underway.

2.2 Performance Tools

Performance tools are software systems that assist programmers in understanding
the runtime behavior of their application\(^1\) on real systems and ultimately in optimizing
the application with respect to execution time, scalability, or resource utilization. To
achieve this goal, the majority of the tools make use of a highly effective experimental
performance analysis approach, based on a measure-modify cycle (Figure 2-1), in which
the programmer conducts an iterative process of performance data collection, data
analysis, data visualization, and optimization until the desired application performance is
achieved [6]. Under this approach, the tool first generates instrumentation code that serves
as entry points for performance data collection (Instrumentation). Next, the application
and instrumentation code are executed on the target platform and raw performance data
are collected at runtime by the tool (Measurement). The tool organizes the raw data
and can optionally perform various automatic analyses to discover and perhaps suggest

---
\(^1\) Alternative approaches include simulation and analytical models.
Figure 2-1. Measure-modify performance analysis approach of performance tool resolutions to performance bottlenecks (Automatic Analysis). Both the raw and analyzed data are then presented in more user-friendly forms to the programmer through text-based or graphical interface (Presentation) to facilitate the manual analysis process (Manual Analysis). Finally, the tool or programmer applies appropriate optimization techniques to the program or the execution environment (Optimization) and the whole cycle repeats until the programmer is satisfied with the performance level.

A tool can use (fixed-interval) sampling-driven instrumentation or event-driven instrumentation, depending on when and how often performance data are collected. In sampling-driven tools, data are collected regularly at fixed-time intervals by one or more concurrently executing threads. At each time step, a predefined, fixed set of metrics (types of performance data) are recorded regardless of the current program behavior (e.g., same metrics recorded regardless of whether the program is performing computation or communication). The performance of the program is then estimated, often using only a subset of these metrics. In most cases, the monitoring threads access only a few
hardware counters and registers and perform limited calculations, thus introducing very low data collection overhead. As a result, this technique is less likely to cause changes in execution behavior that may lead to an inaccurate analysis of the program. However, sampling-driven tools typically have greater difficulty in presenting the program behavior with respect to the high-level source code, especially when the time interval is large enough to miss short-lived trends. In contrast, event-driven tools record data only when specified events (such as the start of a function or communication call) occur during program execution. Together, events and metrics make up the event model that the tool uses to describe application behavior; the complete set of events and metrics is used to reconstruct the behavior of the program in direct relation with high-level source code, easing the analysis and optimization process. For each event, the tool records a select number of metrics (e.g., time, PE ID, etc.) relevant to that particular event but requires significantly more processing time than simply accessing a few hardware counters in the sampling-driven case. As a result, event-driven tools generally introduce higher data collection overhead than sampling-driven tools and thus have a higher chance of introducing heisenbugs: bugs (caused by performance perturbation) that disappear or alter their behavior when one attempts to probe or isolate them. This problem is particularly applicable for frequently occurring, short-lived events that force substantial delay in order to collect performance data.

Another common tool classification, tracing versus profiling, distinguishes how a tool handles the metrics each time instrumentation code is executed. A tool operating in tracing mode stores metric values calculated at each time instance separately from one another. From this data, it is possible for the tool to reconstruct the step-by-step program behavior, enabling application analysis in great detail. However, the large amount of data generated also requires significant storage space per program run, and the sheer amount of data could be overwhelming if it is not carefully organized and presented to the user. In addition, due to memory limitations, the tool often must perform file I/O
during runtime, introducing additional data collection overhead on top of the unavoidable metric calculation overhead. Examples of tools that support the collection and viewing of trace data include Dimemas/Paraver [7], Intel Cluster Tools [8], MPE/Jumpshot [9], and MPICL/ParaGraph [10]. In contrast, a tool operating in profiling mode performs additional on-the-fly calculations\(^2\) (min, max, average, count, etc.) after metric values are calculated at runtime and only statistical (profile) data are kept. This data can usually fit in memory, avoiding the need to perform file I/O at runtime. However, profiling data often only provide sufficient information to perform high-level analysis and may be insufficient for determining the causes of performance bottlenecks. Examples of popular profiling tools include DynaProf [11], mpiP [12], and SvPablo [13].

Finally, as the system used to execute the application grows in size, the amount of performance data that a tool must collect and manage grows to a point that it become nearly impossible for users to manually analyze the data even with the help of the tool. To address this issue, several tools such as HPCToolkit [14], Paradyn [15], Scalasca/KOJAK [16], and TAU [17] also include mechanisms to have the tool automatically analyze the collected performance data and point out potential performance bottlenecks within the application (e.g., scalability analysis, common bottleneck analysis, etc.).

\(^2\) For tracing mode, these statistical calculations are often performed after execution time.
CHAPTER 3
BACKGROUND RESEARCH FINDINGS

A substantial background research process has led to the formulation and development of the PPW system. In this chapter we briefly describe this process and its resulting findings and insights that have shaped the PPW design.

We began our background research by studying the details of parallel programming model specifications and implementations in order to identify characteristics important in analyzing parallel application performance. In parallel, we surveyed existing works on performance tool research in order to identify characteristics important for the success of a performance tool [18]. Using the knowledge gained from these studies, we then evaluated the applicability of existing performance tool techniques to various programming models and leveraged techniques that could be re-used or adopted. Additionally, we performed comparisons between related performance analysis techniques, identified characteristics common to these techniques, and made generalizations based on the commonalities (such generalization is desirable as it reduces tool complexity). Finally, we recognized new obstacles pertaining to parallel performance analysis and formulated solutions to handle these issues.

A helpful performance tool must collect appropriate and accurate performance data. We found that it is useful for a tool to support both profiling and tracing measurement modes. Profile data guides users to program segments where they should focus their tuning efforts, while trace data provides detailed information often needed to determine the root causes of performance degradations. The tool should also make use of hardware counter monitoring systems, such as the portable Performance Application Programming Interface (PAPI) [19], which are valuable in analyzing non-parallel sections of an application but can also be used on parallel sections. Finally, to avoid performance perturbation, the data collection overhead introduced by the tool must be minimized. A general consensus from literature indicates that a tool with overhead of approximately
1-5% under profile mode and 1-10% under trace mode is considered to be safe from performance perturbation.

A productive tool must be easy to learn and use. While performance tools have proved effective in troubleshooting performance problems, they are often not used because of their high learning curve. To avoid this pitfall, a tool should provide an intuitive, familiar user interface by following an established standard or adopting visualizations used by existing performance tools. In addition, since source code is generally the only level which users have direct control over (the average user may not have the knowledge or permission to alter the execution environment), performance tools should present performance data with respect to the application source code. This feature helps in identifying specific source code regions that limited an application’s performance, making it easier for the user to remove bottlenecks. A tool’s ability to provide source-line correlation and to work closer to the source level is thus critical to its success.

To efficiently support multiple programming models, a successful performance tool design must include mechanisms to resolve difficulties introduced due to diverse models and implementations. We noticed that techniques used in the measurement and the presentation stages are generally not tied to the programming model. The types of measurements required and difficulties one must solve are very similar among programming models (i.e., get timestamp, clock synchronization issues, etc.) and visualizations developed are usually applicable or easily extensible to a variety of programming models. Furthermore, we noted that while each programming model supplies the programmer with a different set of parallel constructs to orchestrate work among PEs, the types of inter-PE interaction facilitated by these constructs are quite similar between models. For example, both UPC and SHMEM include constructs to perform, among other operations, barrier, put, and get operation. Thus it is desirable to take advantage of this commonality and devise a generalization mechanism to enable the development of system components that apply to multiple models, helping reduce the complexity of tool design.
In contrast, the choice of the best instrumentation technique is highly dependent on the strategy used to implement the target compiler. Many diverse implementation methods are used by compiler developers to enable execution of target model applications. For instance, all MPI and SHMEM implementations are in the form of linking libraries while UPC implementations range from a direct compilation system (e.g., Cray UPC) to a system employing source-to-source translation complemented with extensive runtime libraries (e.g., Berkeley UPC). We noticed that while it is possible to select an established instrumentation technique that works well for a particular implementation strategy (for example, using the wrapper instrumentation approach for linking libraries), none of these techniques work well for all compiler implementation strategies (see Chapter 4.2 for additional discussion). Thus, any tool that wishes to support a range of models and compilers must include mechanisms to handle these implementation strategies efficiently.
Parallel Performance Wizard (PPW) is a performance data collection, analysis, and visualization system for parallel programs. The goal is to provide a performance tool infrastructure that supports a wide range of parallel programming models with ease; in particular, we focus on the much-needed support for PGAS models [20].

PPW’s high-level architecture is shown in Figure 4-1, with arrows illustrating the steps involved in the PPW-assisted application optimization process. A user’s source program is first compiled using PPW’s commands to generate an instrumented executable. This executable is then run and either profiling or tracing data (as selected by the user) is collected and managed by PPW. The user then opens the resulting performance data file and proceed to analyze application performance in several ways: examining statistical performance information via profiling data visualizations supplied by PPW; converting tracing data to SLOG2 or OTF format for viewing with Jumpshot or Vampir; or using the PPW analysis system to search for performance bottlenecks.

PPW currently supports the analysis of UPC, SHMEM, and MPI 1.x applications and is extensible to support other parallel programming models. To facilitate support for a variety of models, we developed a new concept: the generic-operation-type abstraction. In the remainder of this section, we discuss the motivations behind and advantages of using this abstraction.

Existing performance tools are commonly designed to support a specific model or are completely generic. Model-specific tools interact directly with model-specific constructs and have the advantage of being able to collect a varying set of operation-specific data (such as memory address and data transfer size for put and get). However, the cost of adding support for additional models to these tools is usually high, often requiring updates to a significant portion of the system, due to the need to re-implement the same functionalities for each model. In contrast, completely generic tools (such as MPE [9])
Figure 4-1. PPW-assisted performance analysis process from original source program to revised (optimized) program
work with generic program execution states (i.e., the beginning and the end of function calls) and thus can be easily adopted to support a wide range of models. Unfortunately, being completely generic forces these tools to collect a standard set of metrics (e.g., source line, timestamp) each time data collection occurs, and as a result, these tools lose the capacity to obtain useful operation-specific metrics\(^1\) (e.g., data size for data transfer operations). To avoid the unnecessary tight-coupling of a tool to its supported programming models while still enabling the collection of useful operation-specific metrics, we developed a generic-operation-type abstraction that is a hybrid of the model-specific and the completely generic approaches. The idea is to first map model-specific constructs to a set of model-independent generic operation types classified by their functionality. For each generic operation, the tool can then collect operation-specific events and metrics and may later analyze and present these data differently depending on the operation type (Figure 4-2).

The generic-operation-type abstraction has influenced the development of many components of the PPW system, including its event model, instrumentation and

\(^1\) Note that it is possible but impractical to collect all metrics each time, as part of the collected metrics will not be meaningful.
measurement approach, and analyses and visualizations. We now describe these components (Figure 4-3) in the following sections.

### 4.1 Parallel Programming Event Model

The effectiveness of an event-driven tool is directly impacted by the events and metrics (i.e., event model) that it uses. Events signify important instances of program execution when performance data should be gathered, while metrics define the types of data that should be measured and subsequently stored by the tool for a given event. In this section, we present the generic parallel event model PPW uses to describe the behavior of a given parallel application.

PPW focuses on providing detailed information needed to analyze parallel portions of a program while maintaining sufficient information to enable a high-level sequential analysis. For this reason, the current PPW event model includes mostly parallel events. Table 4-1 summarizes the theoretical event model using the generic-operation-type
abstraction to describe the behavior of a given program; this event model is compatible
with both PGAS models and MPI while Table 4-2 shows the mapping of UPC, SHMEM,
and MPI 1.x constructs to generic operation types.

We organized Table 4-1 so that operation types with the same set of relevant events
are shown together as a group. In addition to metrics useful for any event (i.e., calling PE
ID, code location, timestamp, and operation-type identifier), for each event, we provide a
list of additional metrics that would be beneficial to collect. For one-sided communication,
metrics such as the data source and destination\(^2\) (PE ID and memory address), data
transfer size being transferred, and synchronization handler\(^3\) (for non-blocking operations
only) provide additional insights on the behavior of these operations. For two-sided
communication, it is necessary to match the PE ID, transfer size, message identifier, and
synchronization handler in order to correlate related operations. For lock acquisition
or release operations and wait-on-value change operations, the lock identifier or the
wait-variable address help prevent false bottleneck detection. For collective global-memory
allocation, the memory address distinguishes one allocation call from another. For group
communication, the data transfer size may help understand these operations. Finally,
for group synchronization and communication that do not involve all system PEs, group
member information is useful in distinguishing between distinct but concurrently executing
operations.

It is important to point out that event timestamp information is often the most
critical metric to monitor (as performance optimization usually aimed at minimizing the
observed execution time). With a proper set of events and accurate timing information
for these events, it is possible to calculate (or at least provide a good estimate of) the

\(^2\) The source and destination of a transfer may be different from the calling PE.

\(^3\) Identifier used by the explicit/implicit synchronization operations to force completion
of a particular put or get.
<table>
<thead>
<tr>
<th>Generic-operation type</th>
<th>Events</th>
<th>Additional metrics (may also include PAPI counters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group synchronization</td>
<td>Enter</td>
<td>Group info (sync/comm), address (memory), transfer size (comm)</td>
</tr>
<tr>
<td>Group communication</td>
<td>Notification_End</td>
<td></td>
</tr>
<tr>
<td>Initialization / termination</td>
<td>Wait_Be</td>
<td></td>
</tr>
<tr>
<td>Global memory allocation</td>
<td>Transfers_Received</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exit (Wait_End)</td>
<td></td>
</tr>
<tr>
<td>Atomic read/write</td>
<td>Enter</td>
<td>Source, destination, transfer size, synchronization handler (non-blocking)</td>
</tr>
<tr>
<td>Blocking Implicit put/get</td>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td>Blocking explicit put/get</td>
<td>Transfer_Begin</td>
<td></td>
</tr>
<tr>
<td>Non-blocking explicit put/get</td>
<td>Transfer_Complete</td>
<td></td>
</tr>
<tr>
<td>Explicit communication synchronization</td>
<td>Synchronization_Begin</td>
<td>Synchronization handler</td>
</tr>
<tr>
<td></td>
<td>Synchronization_End</td>
<td>Synchronization handler</td>
</tr>
<tr>
<td>Blocking send/receive</td>
<td>Enter</td>
<td>Matching PE, transfer size, message identifier, synchronization handler (non-blocking)</td>
</tr>
<tr>
<td>Non-blocking send/receive</td>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Matching_Enter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal_Received</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait_Be</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait_End</td>
<td>Synchronization handler</td>
</tr>
<tr>
<td>Lock acquisition or release</td>
<td>Enter</td>
<td>Lock identifier (lock), address (wait)</td>
</tr>
<tr>
<td>Wait-on-value change</td>
<td>Condition_Fulfilled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td>User-defined function/region</td>
<td>Enter</td>
<td></td>
</tr>
<tr>
<td>Work-sharing</td>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td>Environment inquiry</td>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td>Generic-operation type</td>
<td>UPC</td>
<td>SHMEM</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>Initialization</td>
<td>N/A</td>
<td>shmem_init()</td>
</tr>
<tr>
<td>Termination</td>
<td>upc_global_exit()</td>
<td>N/A</td>
</tr>
<tr>
<td>Environment inquiry</td>
<td>MYTHREAD, THREADS, upc_threadof(), ...</td>
<td>my_pe(), num_pes()</td>
</tr>
<tr>
<td>Group sync.</td>
<td>upc_notify(), upc_wait(), upc_barrier()</td>
<td>shmem_barrier(), shmem_barrier_all()</td>
</tr>
<tr>
<td>Group comm.</td>
<td>upc_all_broadcast(), upc_all_scatter(), ...</td>
<td>shmem_broadcast(), shmem_collect(), ...</td>
</tr>
<tr>
<td>Global memory management</td>
<td>Declaration with shared keyword, upc_alloc(), ...</td>
<td>shmalloc(), shfree()</td>
</tr>
<tr>
<td>Implicit put (one-sided)</td>
<td>Direct assignment (shared_int = 1)</td>
<td>N/A</td>
</tr>
<tr>
<td>Implicit get (one-sided)</td>
<td>Direct assignment (x = shared_int)</td>
<td>N/A</td>
</tr>
<tr>
<td>Explicit put (one-sided)</td>
<td>upc_memput(), upc_memset(), upc_memcpy()</td>
<td>shmem_put(), shmem_iput(), ...</td>
</tr>
<tr>
<td>Explicit get (one-sided)</td>
<td>upc_memget(), upc_memcpy()</td>
<td>shmem_get(), shmem_iget(), ...</td>
</tr>
<tr>
<td>Send (two-sided)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Receive (two-sided)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Explicit comm.</td>
<td>upc_fence()</td>
<td>shmem_fence(), shmem_wait_nb(), ...</td>
</tr>
<tr>
<td>Lock acquisition or release</td>
<td>upc_lock(), upc_unlock(), upc_lock_attempt()</td>
<td>N/A</td>
</tr>
<tr>
<td>Atomic comm.</td>
<td>N/A</td>
<td>shmem_int_fadd(), ...</td>
</tr>
<tr>
<td>Work-sharing</td>
<td>upcforall()</td>
<td>N/A</td>
</tr>
<tr>
<td>Wait-on-value change</td>
<td>N/A</td>
<td>shmem_wait(), shmem_wait_until(), ...</td>
</tr>
</tbody>
</table>

---

*upc_memcpy() can be either put and/or get depending on the source and destination*
duration for various computation, communication, and synchronization calls throughout program execution. In some cases, it is also possible to calculate program-induced delays\(^4\) (PI delays — delays caused by poor orchestration of parallel code such as uneven work-distribution, competing data access, or lock acquisition) that point to locations in the program that can be optimized via source-code modification. By examining the durations of various operations and identifying PI delays that can be removed, it is much simpler for a programmer to devise optimization techniques to improve the execution time of the application.

In the following subsections, we discuss the events and means to calculate operation duration and PI delay for each of the logical groups of generic-operation types shown in Table 4-1. For each group, we diagram some typical execution patterns via a set of operation-specific events (each indicated by an arrow with number at the end) ordered with respect to time (x-axis) for each of the processing PEs (PE X, Y, Z) involved. We discuss means to calculate the duration of non-blocking operations (duration for blocking operation is always the time difference between its Enter and Exit event and PI delay with these events, discuss why the inclusion of some events affects the accuracy of the calculations, and mention how a tool can track these events in practice. In addition, we point out performance issues typically associated with each operation group.

4.1.1 Group-Related Operations

In Figure 4-4, we illustrate the events for the category of operations that involves a group of PEs working together, including group synchronization, group communication, initialization, termination, and global memory allocation operations. The execution behavior of these operations is commonly described in terms of participating PEs running in one of two phases. First is the notification phase when the calling PE sends out signals

---

\(^4\) Example of a delay which is not a PI delay includes data transfer delay due to network congestion, slowdown due to multiple applications running at the same time, etc.
to all other PEs in the group indicating its readiness in performing the operation. The second is the wait phase where the calling PE blocks until the arrival of signals from all PEs before completing the operation. Two versions of these group operations are typically provided to programmers: the standard (blocking) single-phase version where a single construct is used to complete both phases and the more flexible (non-blocking) split-phase version using separate constructs for each phase that allows for overlapping operation (generally restricted to local computation). With respect to existing programming models, the single-phase version is available for all operations in this category while the split-phase version is typically only available for group synchronization and group communication. PI delays associated with these operations normally mark the existence of load-imbalance issues.

Events associated with this category of operations are the following:

- **Enter (Notification_Begin):** Event denoting the beginning of cooperative operation (beginning of notification phase). The calling PE starts sending out Ready signals (plus data for group communication operations) to all other PEs.

- **Notification_End:** Event denoting the point in time when the calling PE finishes sending Ready signals (plus data for group communication operations) to all other PEs (end of notification phase). For the split-phase version, the calling PE is free to perform overlapping operations after this point until the wait phase. In the single-phase version, this event is normally not traceable directly but is estimated to occur a short time after the Enter event.

- **Wait_Begin:** Event denoting the beginning of the wait phase (where the calling PE blocks until Ready signals are received from all other PEs). Normally only traceable for split-phase version.

- **Transfers_Received:** Event denoting the arrival of Ready signals from all other PEs on the calling PE. This event is usually not traceable directly but is estimated to occur a short time after the last participating PE enters the operation.

- **Exit (Wait_End):** Event denoting the completion of the cooperative operation.

An example execution pattern exhibiting bottlenecks (on PE X and Y) caused by uneven work-distribution for the single-phase version is diagramed in Figure 4-4a. In this scenario, PE Z entered the operation after it has received a Ready signal from
both PE X and Y (i.e., on PE Z, a Transfers_Received event occurred before an Enter event) so it was able to complete the operation without blocking. In contrast, PE X and Y finished sending out signals before receiving all incoming Ready signals so they were unable to complete the operation optimally; each PE became idle until it reached the Transfers_Received event. PI delay for the single-phase version is given by the time difference between the Transfers_Received and Notification_End events (Figure 4-4, bottom).

Figure 4-4b shows an example execution pattern for the split-phase version. In this scenario, PE Z received all signals before entering the notification phase so it is free of any PI delay. PE Y entered the wait phase before receiving all signals so it remained idle for a period of time before completing its operation (idle time given by the time difference between the Transfers_Received event and Wait_Begin event). Finally, PE X shows a situation where overlapping computation is used to remove potential delays (advantage of

Figure 4-4. Events for group synchronization, group communication, initialization, termination, and global memory allocation operations (TS = Timestamp. Only a few of PE Y’s events are shown to avoid clutter)
split-phase version). PE X entered the notification phase first so it logically required the longest wait time (i.e., largest difference between the Transfers_Received and the Enter event). However, by performing sufficient computation, PE X no longer needed to wait once it entered the wait phase and thus was free of delay. For the split-phase version, the total operation duration is given by the combined duration of the notification phase (time difference between the Enter and Notification_End event) and the wait phase (time difference between the Wait_Begin and Exit event) and the PI delay is given by the time difference between its Transfers_Received and Wait_Begin event (Figure 4-4, bottom).

4.1.2 Data Transfer Operations

In Figure 4-5, we illustrate the events for operations relating to one-sided, point-to-point data transfers such as atomic operations, blocking or non-blocking explicit or implicit put, and get operations, and explicit communication synchronization (e.g., fence, quiet). Note that we present the get operation as a reverse put\(^5\) (where the calling PE sends a request to the target PE and the target PE performs a put) since get operations are often implemented this way in practice in order to improve their performance. For this class of operations, we are interested in determining the time it takes for the full data transfer to complete; from beginning of read or write to when data is visible to the whole system. The precise duration for the non-blocking version could be calculated if the Transfer_Complete event is available; unfortunately, it is often not possible for model implementations to supply this event. For such systems, the duration can only be estimated from the end time of either the explicit or implicit communication synchronization (Synchronization_End event) that enforces data consistency among PEs. As illustrated in Figure 4-5c, this estimated duration could be much higher than the precise duration and as a result compromises the reliability of subsequent analyses. Furthermore, any PI delays caused by the synchronization operations increase

\(^5\) A canonical get would have events similar to those illustrated for a put operation.
the duration time further away from the actual transfer time. Finally, if an explicit synchronization operation is used to force the completion of multiple data transfers, PI delay in one transfer will affect the duration calculation for all other transfers as well, further decreasing the accuracy of the performance information.

To calculate the PI delay, either the Transfer_Begin or the Transfer_Complete event is needed; they are sometimes obtainable by examining the Network Interface Card (NIC) status. PI delay for blocking put (get) is the time difference between the Transfer_Begin and the Enter event. For non-blocking put (get) using implicit synchronization, PI delay is the time difference between the Transfer_Complete and the Synchronization_Begin event. For non-blocking put (get) using explicit synchronization, PI delay is the time difference between the Transfer_Begin and the Synchronization_Begin event. PI delays associated with these operations often signify the existence of competing data accesses.

In Figure 4-6, we illustrate the events for operations (of PE X) relating to two-sided, point-to-point data transfers such as blocking or non-blocking send and receive operations and explicit communication synchronization. As with one-sided communication, we are interested in determining the time it takes for the full data transfer to complete. The duration for the non-blocking versions is the time difference between the send (receive) Enter and the Exit event plus the time difference between the Signal_Received and the Wait_End event. Unfortunately, the Signal_Received event (event denoting the point in time when the calling PE received a Ready signal from the matching PE) is nearly impossible to obtain; thus duration can only be estimated from the exit time of the synchronization call that guarantees data transfer completion, resulting in much higher than normal duration calculation. PI delay for blocking send (receive) is the time difference between the Enter and the Matching_Enter events while PI delay for non-blocking send (receive) with matching blocking receive (send) is the time difference between the Wait_Begin and the Matching_Enter event. For a non-blocking send (receive) with matching non-blocking receive (send), PI delay cannot be calculated
Figure 4-5. Events for one-sided communication and synchronization operations

(a) Blocking put
(b) Blocking get
(c) Non-blocking put with implicit synchronization
(d) Non-blocking get with implicit synchronization
(e) Non-blocking put with explicit comm. synchronization
(f) Non-blocking get with explicit comm. synchronization

Duration(Blocking) = TS(Exit) – TS(Enter)
Duration(Non-blocking, actual) = TS(Actual_Completion) – TS(Enter)
Duration(Non-blocking, estimated) = TS(Synchronization_End) – TS(Enter)

PI Delay(Blocking) = TS(Transfer_Begin) – TS(Enter)
PI Delay(Non-blocking, implicit) = TS(Transfer_Complete) – TS(Synchronization_Begin)
PI Delay(Non-blocking, explicit) = TS(Transfer_Begin) – TS(Synchronization_Begin)
Figure 4-6. Events for two-sided communication and synchronization operations

(since Signal_Received is not available). These PI delays signify a potentially inefficient calling sequence of send and receive pairs that typically stem from a load-imbalance prior to calling of these operations.

4.1.3 Lock, Wait-On-Value, and Locally Executed Operations

In Figure 4-7a, we illustrate the events for the lock mechanisms and the wait-on-value operation. The duration is calculated from the time difference between the Enter and Exit events. To calculate the PI delay, the Condition_Fulfilled event is needed, which indicates when the lock becomes available (unlocked) or when a remote PE updates the variable to have a value satisfying the specified wait condition. This Condition_Fulfilled event is generally not traceable directly by the tool but instead can be estimated from other
operations’ events (i.e., the last unlock or data transfer completed). PI delays associated with these operations generally stem from poor orchestration among processing PEs (such as lock competition and late updates of wait variables).

Finally in Figure 4-7b, we illustrate the events for the locally executed operations such as user-defined function or region, work-sharing, and environment inquiry operations. Tracking the performance of these operations is important as it facilitates the analysis of local portions of the program. Since we can consider each to be a blocking operation, the duration is simply the time difference between the Enter and Exit events. Without extensive sequential performance tracking and analysis, it is not possible to determine if any PI delay exists.

4.1.4 Implementation Challenges and Strategies

In this subsection, we briefly discuss the challenges and strategies used to implement the event model with respect to data collection (instrumentation and measurement), automatic data analysis, and data presentation. A more detailed discussion of each of these stages will be given in Chapter 4.2, 5 and 4.4 respectively.

Together, the programming model, chosen instrumentation technique, and tool design decision determines the set of events that the tool collects during runtime. The
programming model supplies the Meaningful Event Set to collect as specified by the event model discussed previously. From this set, a subset of Measurable Event Set that can be collected directly during runtime given the constraints imposed by the chosen instrumentation technique is identified. Finally, some tool design decisions may further limits the Actual Event Set (a subset of Measurable Event Set) the tool supports. Once the Actual Event Set is known, metrics (common metrics plus additional metrics in Table 4-1) associated with events in this set are collected during runtime.

Depending on the Actual Event Set collected, analyses performed during the analysis phase will differ. For example, to calculate barrier duration and PI delay in MPI and SHMEM, the single-phase formulas are used, while for Berkeley UPC, the split-phase formulas are used. Programming model capabilities also play a role in what kind of analyses are performed. Analyses specific to barriers can be applied to all three models, while analyses specific to one-sided transfer operations (e.g., PI delay due to competing put and get) are applicable to both SHMEM and UPC, but not MPI 1.x.

For the presentation stage, each visualization is equipped to handle each type of operation defined in the event model. For some visualizations, the operation type plays no role in how the visualization handles the data, while other visualizations must include mechanisms to handle each type separately. For example, table-based views display data for all operation types in a similar fashion, but a grid-based view of data transferred between PEs makes specific use of communication-related metrics related to data-transfer operations exclusively. Other visualizations, such as the timeline view of trace data, operate differently for various operation types; for example, the timeline needs to appropriately handle two-sided operations, one-sided data transfers (for which a new approach is needed to handle this operation type), etc.
4.2 Instrumentation and Measurement

In this section, we introduce known approaches for instrumentation, discuss their strengths and limitations within the context of the goals of PPW, and then present our data collection solution based on a novel, standardized performance interface called GASP.

4.2.1 Overview of Instrumentation Techniques

While several techniques have proven to be effective in application instrumentation [21], the differences in compilation and execution among the divergent compilation approaches prevent the selection of a universal instrumentation strategy. With source instrumentation, the instrumentation code is added as part of the high-level source code prior to execution time. Because the source code is altered during the instrumentation process, this technique may prevent compiler optimization and reorganization and also lacks the means to handle global memory models where some semantic details of communication are intentionally underspecified at the source level to allow for aggressive optimization (for example, implicit read or write of a shared variable in UPC is difficult to handle using source instrumentation, especially under the relaxed memory consistency mode where a given compiler may reorder the implicit calls to improve performance).

With binary instrumentation, the instrumentation code is added to the machine code before or during program execution. A direct benefit of modifying the machine-code rather than the source-code is that recompilation is often not needed after each program modification. Unfortunately, binary instrumentation is unavailable on some architectures and yields performance data that is often difficult to correlate back to the relevant source code, especially for systems employing source-to-source translation. Finally, with library instrumentation (such as PMPI for MPI), wrappers are placed around functions implementing operations of interest. During execution time, a call to a function first executes the appropriate wrapper code that enables data collection and then invokes the original function. This approach is very easy to use but does not work for programming model constructs that are not in the form of a function call (such as an implicit put in
A brute force approach to having a tool simultaneously support multiple programming models and implementations is simply to select an existing instrumentation technique that works for each particular model implementation. Unfortunately, this approach forces the writers of performance tools to be deeply versed in the internal and often changing or proprietary details of the implementations, which can result in tools that lack portability. In addition, the use of multiple instrumentation techniques forces the tool to handle each model implementation disjointly and thus complicates the tool development process.

4.2.2 The Global-Address-Space Performance Interface

The alternative we have pursued is to define an instrumentation-measurement interface, called the Global-Address-Space Performance (GASP) interface (Appendix A), that specifies the relationship between programming model implementations and performance tools (Figure 4-8). This interface defines the events and arguments of importance for each model construct (see Table 4-3 for GASP events and arguments related to non-blocking UPC communication and synchronization calls). Insertion of appropriate instrumentation code is left to the compiler writers who have the best knowledge about the execution environment, while the tool developers retain full control of how performance data are gathered. By shifting the instrumentation responsibility from tool writer to compiler writers, the chance of instrumentation altering the program behavior is minimized. The simplicity of the interface minimizes the effort required from the compiler writer to add performance tool support to their system (and once completed, any tool that supports GASP and recognize these model constructs can

---

6 For example, existing UPC implementations include direct, monolithic compilation systems (GCC-UPC, Cray UPC) and source-to-source translation complemented with extensive runtime libraries (Berkeley UPC, HP UPC, and Michigan UPC).
support application analysis for that compiler). Concomitantly, this approach also greatly reduces the effort needed for performance tool writers to add support for a variety of model implementations; a single tool-side GASP implementation is sufficient for all compilers with GASP support.
Figure 4-9. Specification of `gasp_event_notify` callback function

The most important entry point in the GASP interface is the event callback function named `gasp_event_notify` (Figure 4-9) that compilers use to notify when events of potential interest occur at runtime and provide useful information (e.g., event identifier, source code location, and event-related arguments) to the performance tool. The tool then decides how to handle the information and what metrics to record. In addition, the tool is permitted to make calls to routines that are written in the source programming model or that use the source library to query model-specific information which may not otherwise be available. The tool may also consult alternative sources of performance information, such as CPU hardware counters exposed by PAPI, for monitoring serial aspects of computational and memory system performance in great detail. The `gasp_event_notify` callback includes a per-thread, per-model context pointer to an opaque, tool-provided object created at initialization time, where the tool can store thread-local performance data.

The GASP specification is designed to be fully thread-safe, supporting model implementations where arbitrary subsets of programming model threads may be implemented as threads within a single process and virtual address space. It is highly extensible by allowing a tool to capture model- and implementation-specific events at varying levels of detail and to intercept just the subset of events relevant to the current analysis task. It also allows for mixed-model application analysis whereby a single performance tool can record and analyze performance data generated by all programming models in use and present the results in a unified manner. Finally, GASP provides

```c
enum gasp_event_type {gasp_event_type_start, gasp_event_type_end,
                      gasp_event_type_atomic};

void gasp_event_notify(
           unsigned int event_id,
           enum gasp_event_type event_type,
           const char* source_file,
           unsigned int source_line,
           unsigned int source_col,
           ...
);
```
facilities to create user-defined, explicitly-triggered performance events which allow the user to give context to performance data. This user-defined context data facilitates phase profiling and customized instrumentation of specific code segments.

Several user-tunable knobs are also defined by the GASP specification to provide finer control over the data collection process. First, several compilation flags are included so user can control the event types tool will collect during runtime. For example, the `--inst-local` compilation flag is used to request instrumentation of data transfer operations generated by shared local accesses (i.e., one-sided accesses to local data which are not statically known to be local). Because shared local accesses are often as fast as normal local accesses, enabling these events can add a significant runtime overhead to the application so by default, the tool does not collect these data. However, shared local access information is useful in some analyses, particularly those that deal with optimizing data locality (a critical consideration in PGAS programming, see Section 5.2.3.5) and performing privatization optimizations, and thus may be worth the additional overhead. Second, instrumentation `#pragma` directives are provided, allowing the user to instruct the compiler to avoid instrumentation overheads for particular regions of code at compile time. Finally, a programmatic control function is provided to toggle performance measurement for selected program phases at runtime.

The complete GASP-enabled data collection process works as follows. First, the compiler-side GASP implementation (Instrumentation Unit) generates instrumentation code which is executed together with the application. The tool-side GASP implementation (Measurement Unit) then intercepts these calls and performs the desired measurement. Next, the raw data are passed to the Performance Data Manager that is responsible for storing this raw data, merging data from multiple PEs, and performing simple post-processing of data (e.g., calculating averages among PEs). These data are then used by the automatic analysis units and presentation units at the later stages of performance evaluation.
4.2.3 GASP Implementations

Here we briefly discuss considerations for the compiler-side implementation of the GASP interface, focusing on UPC as it is the more interesting case. There are several UPC compilers with existing GASP implementations: Berkeley UPC, GCC UPC, and HP UPC [22]. Berkeley UPC translates UPC code to standard C code with calls to the Berkeley UPC runtime system. As a result, much of the corresponding GASP implementation consists of appropriate GASP calls made within the runtime system. However, several features of the GASP specification must be implemented within the compiler itself, including the \#pragma directives for controlling instrumentation of program regions and support for instrumentation of user-function calls. In addition, to provide appropriate UPC source code correlation, the compiler must pass source code information down through the translation process. By contrast, the GCC UPC and HP UPC compilers both use a direct compilation approach, generating machine code directly instead of translating UPC into C. With this architecture, the GASP implementation involves more changes to the compiler itself than with Berkeley UPC. In the case of GCC UPC, for example, changes were needed in one of the UPC compilation phases (called the “gimplification” phase because intermediate representations of functions are converted to GCC’s GIMPLE language) to determine if instrumentation is enabled and generate appropriate code if so.

4.3 Automatic Analysis

The analysis module aims at providing the tool with automatic performance bottleneck detection and resolution capabilities. In this section, we briefly describe the capabilities of analysis units and leave the in-depth discussion on the automatic analysis system development to Chapter 5.

The High-Level Analysis Unit provides analyses of the overall program performance not easily associated with a given operation type (e.g., load-balancing analyses) as well as multiple experiment comparison (e.g., scalability analysis). To provide finer analyses,
the model-independent Bottleneck Detection Unit uses both profiling and tracing data to identify bottlenecks and determine their cause for a particular execution. Once identified, Bottleneck Resolution Units then try to provide suggestions on how to remove these bottlenecks from the application. These units are partially model-dependent, as a given resolution strategy may not always work for all programming models. For example, a technique to fix the performance degradation stemming from `upc_memget`, versus from `shmemb_get`, could be different even though they are both classified as one-sided get operations. Each of the analysis units generates new analysis data that are incorporated by the Performance Data Manager and later presented by the Visualization Manager.

### 4.4 Data Presentation

PPW provides both graphical and text-based interfaces to view collected profile data and generated analysis results. Most of these visualizations have been designed to have a similar look and feel to those provided by other tools so users already familiar with other tools can quickly learn and effectively use PPW. The following list summarizes the visualization-related features (with each supporting source-code correlation whenever possible) currently provided by PPW:

- A view summarizing the application execution environment (optimization flags used, machine hostnames, etc.).
- Charts to facilitate the identification of time-consuming application segments (10 longest-executing regions).
- Flat and call-path tables to display high-level statistical performance information.
- A visualization to detect and show event-level load-balancing issue (Figure 4-10a, see Chapter 5.2.3.3 for more detail).
- A chart to compare related experimental runs (Figure 4-10b) such as runs of the same program using various system sizes or runs of different versions of the same program.
- A display showing the inter-PE communication volume for all data transfer operations in the program (providing PE-to-PE or PE-to-global-memory communication statistics).
Table 4-4. Profiling/tracing file size and overhead for UPC NPB 2.4 benchmark suite

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (profile) (KB)</td>
<td>113</td>
<td>840</td>
<td>369</td>
<td>276</td>
<td>195</td>
</tr>
<tr>
<td>Size (trace) (MB)</td>
<td>0.15</td>
<td>34</td>
<td>142</td>
<td>1050</td>
<td>4560</td>
</tr>
<tr>
<td>Overhead (profile)</td>
<td>&lt; 0.1%</td>
<td>2.69%</td>
<td>&lt; 0.1%</td>
<td>1.66%</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Overhead (trace)</td>
<td>&lt; 0.1%</td>
<td>4.30%</td>
<td>&lt; 0.1%</td>
<td>2.84%</td>
<td>2.08%</td>
</tr>
</tbody>
</table>

- A unique, UPC-specific Array Distribution display that depicts the physical layout of shared objects in the application on the target system (Figure 4-11).
- Exports of trace data for viewing with the Jumpshot and Vampir [23] timeline viewers.

### 4.5 PPW Extensibility, Overhead, and Storage Requirement

We first develop the Parallel Performance Wizard tool to support UPC application analysis which took close to two years to develop. This tool is later extended to support SHMEM and MPI 1.x, each taken less than six months to complete with a large portion of time spend on system configuration. In Table 4-4, 4-5, and 4-6, we provide experimental data on the data-collection overhead and storage space needed for PPW on an Opteron cluster connected with a QsNet II Quadrics interconnect using the following test programs.

For UPC, we executed the George Washington University’s UPC NPB version 2.4 benchmark suite (class B) [24] with Berkeley UPC [3] version 2.6. For SHMEM, we executed the Quadrics’s APSP and CONV test programs and an in-house SHMEM SAR application (see Chapter 6.3) with Quadrics SHMEM [4]. For MPI, we used the Tachyon [25], an in-house MPI SAR [26] and an in-house Corner Turn applications with MPICH2 [27] v1.0.8. All programs were instrumented and monitored using PPW (v.2.2), with performance data collected for all UPC/SHMEM/MPI constructs and user functions in each program. In all cases, the data-collection overhead numbers (< 2.7% for profile, < 4.3% for trace) are comparable to existing performance tools. Tracing data size is linearly related to the total number of events instrumented by the tool; on average, PPW requires 17MB of storage space per 1 million trace events.
Figure 4-10. (a) Load-balancing analysis visualization for CG 256-PE run, (b) Experimental set comparison chart for Camel 4-, 8-, 16-, and 32-PE runs
Figure 4-11. Annotated screenshot of new UPC-specific array distribution visualization showing physical layout of a 2-D 5x8 array with block size 3 for an 8-PE system

Table 4-5. Profiling/tracing file size and overhead for SHMEM APSP, CONV, and SAR application

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>APSP</th>
<th>CONV</th>
<th>SAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (profile) (KB)</td>
<td>116</td>
<td>84</td>
<td>256</td>
</tr>
<tr>
<td>Size (trace) (MB)</td>
<td>0.14</td>
<td>0.12</td>
<td>0.26</td>
</tr>
<tr>
<td>Overhead (profile)</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Overhead (trace)</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
<td>&lt; 0.1%</td>
</tr>
</tbody>
</table>

Table 4-6. Profiling/tracing file size and overhead for MPI corner turn, tachyon, and SAR application

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Corner Turn</th>
<th>Tachyon</th>
<th>SAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (profile) (KB)</td>
<td>150</td>
<td>196</td>
<td>76</td>
</tr>
<tr>
<td>Size (trace) (MB)</td>
<td>0.15</td>
<td>0.20</td>
<td>0.18</td>
</tr>
<tr>
<td>Overhead (profile)</td>
<td>0.36%</td>
<td>0.62%</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Overhead (trace)</td>
<td>0.49%</td>
<td>0.71%</td>
<td>&lt; 0.1%</td>
</tr>
</tbody>
</table>
Table 4-7. Profiling/tracing file size and overhead comparison of PPW, TAU, and Scalasca tools for a 16-PE run of the MPI IS benchmark

<table>
<thead>
<tr>
<th></th>
<th>PPW</th>
<th>TAU</th>
<th>Scalasca</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (profile) (KB)</td>
<td>133</td>
<td>77.5</td>
<td>56</td>
</tr>
<tr>
<td>Size (trace, uncompressed) (MB/million events)</td>
<td>16.16</td>
<td>23.75</td>
<td>11.44</td>
</tr>
<tr>
<td>Overhead (profile)</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>Overhead (trace)</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
</tbody>
</table>

4.5.1 PPW, TAU, and Scalasca Comparison

In Table 4-7, we show a brief overhead and data-size comparison of the PPW, TAU, and Scalasca tools for a 16-PE run of the MPI IS benchmark on a Quad-core Xeon cluster using a Gigabit Ethernet interconnect. From this table, we see that the data-collection overhead for each of the three tools is negligible compared to the total execution time. For profiling, PPW requires a higher (but still quite small) amount of storage space than TAU and Scalasca. For tracing, PPW requires 16.16 MB of uncompressed storage space per one million trace events generated while TAU and Scalasca require 24.03 MB and 11.44 MB of uncompressed storage space, respectively.

4.5.2 PPW Tool Scalability

To evaluate the scalability of PPW, we conducted 128-, 256-, and 512-PE runs of GWU’s UPC NPB version 2.4 benchmarks (class B) using Berkeley UPC 2.8 (via the GASNet MPI conduit using MPICH2 version 1.0.8) on an 80-PE Intel Quad-core Xeon cluster with a Gigabit Ethernet interconnect (with all model constructs and user functions instrumented). In Figure 4-12, we show the communication statistics visualization for a 256-PE run of CG (Figure 4-12a) and the zoomed-in Jumpshot view of an MG 512-PE run (Figure 4-12b). As shown in Table 4-8, the data collection overhead numbers are higher than with runs on smaller system sizes but are still within the acceptable range (< 6.34% for profiling, < 5.61% for tracing). In all cases, profile data size remained in the manageable MB range. In contrast, trace data size for larger runs is significantly greater for some of these benchmarks that exhibit weak scaling, such as CG and MG (for
Table 4-8. Profiling/tracing file size and overhead for medium-scale UPC NPB 2.4 benchmark suite runs

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th></th>
<th>Tracing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td>CG</td>
<td>Overhead</td>
<td>0.87%</td>
<td>0.31%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Data size</td>
<td>4.0 MB</td>
<td>14.1 MB</td>
<td>N/A</td>
</tr>
<tr>
<td>EP</td>
<td>Overhead</td>
<td>6.34%</td>
<td>0.91%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td></td>
<td>Data size</td>
<td>3.2 MB</td>
<td>11.8 MB</td>
<td>45.1 MB</td>
</tr>
<tr>
<td>FT</td>
<td>Overhead</td>
<td>4.44%</td>
<td>N/A</td>
<td>4.31%</td>
</tr>
<tr>
<td></td>
<td>Data size</td>
<td>4.9 MB</td>
<td>N/A</td>
<td>164.6 MB</td>
</tr>
<tr>
<td>IS</td>
<td>Overhead</td>
<td>&lt; 1%</td>
<td>4.06%</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Data size</td>
<td>3.7 MB</td>
<td>12.8 MB</td>
<td>N/A</td>
</tr>
<tr>
<td>MG</td>
<td>Overhead</td>
<td>4.55%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td></td>
<td>Data size</td>
<td>8.0 MB</td>
<td>21.3 MB</td>
<td>64.1 MB</td>
</tr>
</tbody>
</table>

benchmarks such as IS that exhibit strong scaling, the data size stays relatively constant); this characteristic could become an issue as system size continues to increase.

4.6 Conclusions

The goal of the first part of this research was to investigate, design, develop, and evaluate a model-independent performance tool framework. While many tools support performance analysis of message-passing programs, tool support is limited for applications written in other programming models such as those in the PGAS family. Existing tools were specifically designed to support a particular model (i.e., MPI) and they became too tightly coupled with that model. As a result, a significant amount of effort from the developers is needed to add new model support. To address this issue, the PPW performance system with two novel concepts was developed. We introduced the generic-operation-type abstraction concept and illustrated how the generic-operation-type-based event model helps in minimizing the dependency of a tool to its supported programming models and discussed the need for the new GASP interface and how this interface simplifies the otherwise cumbersome data collection process. With the inclusion of these two concepts, our PPW tool framework supports and is easily extensible to support a wide range of parallel programming models.
Figure 4-12. (a) Data transfers visualization showing communication volume between processing PEs for 256-PE CG benchmark tracing mode run, (b) Zoomed-in Jumpshot view of 512-PE MG benchmark
To validate the proposed framework, we developed the PPW prototype tool that originally supports manual analysis of UPC applications and later extended to support manual analysis of SHMEM and MPI 1.x programs. We showed that while it took over two years to develop the first prototype, extending the prototype to support other programming models was achieved fairly quickly (less than 6 month for both SHMEM and MPI), proving that our proposed framework is highly extensible. In addition, we demonstrated that our PPW prototype incurred overhead (< 3% for profiling and < 5% for tracing for all supported models) well within the acceptable range, is comparable to other popular performance tools, and is still usable up to 512 PEs.

Future work on this part of the research includes integrating PPW into the Eclipse development environment; enhancing the scalability of existing PPW visualizations; improving data-collection overhead, management, and storage on larger systems; and providing lower-level (e.g., programming model runtime and network-related) performance information using GASP.
CHAPTER 5
SYSTEM FOR AUTOMATIC ANALYSIS OF PARALLEL APPLICATIONS

Performance tools that collect and visualize raw performance data have proven to be productive in the application optimization process. However, to be successful in this manual analysis process, the user must possess a certain degree of expertise to discover and fix performance bottlenecks — and thus limiting the usefulness of the tool, as non-expert programmers often do not have the skill set needed. In addition, as the size of the performance dataset grows, it becomes nearly impossible to manually analyze the data, even for expert programmers. One viable solution to this issue is an automatic analysis system that can detect, diagnose, and potentially resolve bottlenecks.

In this chapter, we present a new automatic analysis system that extends the capabilities of the PPW performance tool. The proposed system supports a range of analyses that (to our knowledge) no single existing system provides and uses novel techniques such as baseline filtering and a parallelized analysis process to improve execution time and responsiveness of analyses. In addition, because it is based on the generic-operation-type abstraction introduced earlier, the analysis framework is applicable to any parallel programming model with constructs that can be mapped to the supported operation types.

To avoid confusion, we begin by defining some important terms used in the remainder of this chapter. A performance property (or pattern) defines an execution behavior of interest within an application. A performance bottleneck is a performance property with non-optimal behavior. Bottleneck detection (or identification, discovery) is the process of finding the locations (PE, line of code, etc.) of performance bottlenecks. Cause analysis is the process of discovering the root causes of performance bottlenecks (e.g., late barrier entrance caused by uneven work distribution). Bottleneck resolution is the process of identifying potential strategies that may be applied to remove the bottlenecks. Automatic optimization refers to source code transformation and/or changes in the execution
environment made by the tool to improve application performance. Finally, a hotspot is a
portion of the application that took a significant percentage of time to execute and thus is
a good candidate for optimization.

5.1 Overview of Automatic Analysis Approaches and Systems

Automatic (or automated) analysis is a tool-initiated process to facilitate the finding
and ultimately the removal of performance bottlenecks within an application. The entire
process may involve the tool, with or without user interaction, performing some or all of
the tasks illustrated in Figure 5-1 on the application under investigation. Note that in
the figure, performance data collection refers to the gathering of additional data on top of
what the tool collects by default. In the remainder of this section, we provide an overview
of existing work relating to automatic analysis.

The APART Specification Language (ASL) [28] is a formal specification model
introduced by the APART [29] working group to describe performance properties via three
components: a set of conditions to identify the existence of the property, a confidence
value to quantify the certainty that the property holds, and a severity measure to describe
the impact of the property on performance. The group used this language to provide a

---

1 Because bottleneck detection and cause analysis are closely tied to each other, in some
literature they are together referred to as the bottleneck detection process.
list of performance properties for the MPI, OpenMP, and HPF programming models and noted the possibility of defining a set of base (model-independent) performance property classes.

HPCToolkit and TAU are examples of tools providing features to evaluate the scalability of an application using profiling data. HPCToolkit uses the timing information from two experiments to identify regions of code with scalability behavior that deviates from the weak or strong scaling expectation \[30\]. PerfExplorer is an extension of TAU that generates several types of visualizations that compare the execution time, relative efficiency, or relative speedup of multiple experiments \[31\]. In addition, PerfExplorer includes techniques such as clustering, dimension reduction, and correlation analysis to reduce the amount of performance data the user must examine.

Periscope, KappaPI-2, and KOJAK are knowledge-based tools that support the detection of well-known performance bottlenecks defined with respect to the programming model. The advantage of a knowledge-based system is that little or no expertise is required of the user to successfully analyze the program. Periscope supports online detection of MPI, OpenMP, and memory system related bottlenecks (specified using ASL) through a distributed hierarchy of processing units that evaluate the profiling data \[32\]. KappaPI-2 is a post-mortem, centralized, tree-based analysis system that supports bottleneck detection, cause analysis, and bottleneck resolution (via static source code analysis) using tracing data \[33\]. Finally, EXPERT is a part of KOJAK (now known as Scalasca) that supports post-mortem bottleneck detection and cause analysis of MPI, OpenMP, and SHMEM bottlenecks (specified using ASL). The developers recently introduced an event-reply strategy to allow parallel, localized analysis processing which has been successfully applied to MPI \[34\], but it remains questionable whether such a strategy works well for other programming models.

Hercules \[35\] is a prototype knowledge-based extension of TAU that detects and analyzes causes of performance bottlenecks with respect to the programming paradigm.
(such as master-worker, pipeline, etc.) rather than the programming model. An advantage of this system is that it can be used to analyze applications written in any programming model. Unfortunately, the system cannot handle applications developed using a mixture of paradigms or that do not follow any known paradigm at all, making it somewhat limited in applicability.

Paradyn’s online $W^3$ search model [36] was designed to answer three questions through iterative refinement: why is it performing poorly, where are the bottlenecks, and when did the problems occur. The $W^3$ search system analyzes instances of performance data at runtime, testing a hypothesis which is continually refined along one of the three question dimensions. The $W^3$ system considers hotspots to be bottlenecks, and since not all hotspots contribute to performance degradation (they could simply be performing useful work), the usefulness of this system is somewhat limited.

The main idea behind the design of NoiseMiner [37], a component of the Projections tool, is that events of similar type should have similar performance under ideal circumstances. Utilizing this assumption, the system makes a pass through the trace log, assigns an expected performance value to each event type, and then identifies specific trace events with performance that do not meet the expectations (i.e., noisy events).

Performance Assertions (PA) is a prototype source code annotation system for the specification of performance expectations [38]. Once performance assertions are explicitly added by the user, the PA runtime collects data needed to evaluate these expectations and selects the appropriate action (e.g., alert the user, save or discard data, call a specific function, etc.) during runtime. IBM has also developed an automated bottleneck detection system enabling the detection of arbitrary performance properties within an application [39]. The system supplies users with an interface to add new performance properties using pre-existing metrics and to add new metrics needed to formulate the new properties. With both of the above systems, a certain degree of expertise is required of the user to formulate meaningful assertions or properties.
Each of the above approaches has made a contribution to the field of automatic performance analysis. Each also has particular drawbacks that limit its effectiveness or applicability. In light of ongoing progress in and the ever-increasing complexity of parallel programming models and environments, we have sought to make corresponding progress in effective analysis functionality for a variety of modern programming models.

5.2 PPW Automatic Analysis System Design

The PPW automatic analysis system focuses on optimization of observed application execution time with the goal of guiding (and possibly provide hints to) users to specific regions of code to focus their optimization efforts on. The proposed system is novel in several aspects. First, the analysis system makes use of the same (model-independent) generic-operation-type abstraction underlying our PPW tool. As a result, our analysis system can be easily adapted to support a wide range of parallel programming models and naturally supports the analysis of mixed-model applications (i.e., programs written using two or more models). The use of this abstraction also improves the system’s capabilities by allowing in-depth analysis of some user-defined functions. For example, by simply instructing the system to treat a user-defined \texttt{upc\_user\_wait\_until} function in a UPC program as a wait-on-value-change operation (adding one line in the event type mapping), the system is able to determine the cause of any delays associated with this function.

Second, we introduce several techniques such as a new baseline filtering technique\textsuperscript{2} to identify performance bottlenecks via comparison of actual to expected performance values that is generally more accurate than the deviation filtering technique used in NoiseMiner. Third, our system performs a range of existing and new analyses including scalability analysis, load-balance analyses, frequency analysis, barrier-redundancy analysis, and common-bottleneck analysis, whereas other systems support only a few of these analyses. Finally, we have developed a scalable analysis processing technique to minimize the execution time and responsiveness of the analyses. This process is designed to allow multiple localized analyses to take place in parallel (involving minimal data transfers) and
is able to identify specific bottleneck regions using only profile data and determine the cause of the bottleneck when trace data is available. Compared to other parallel analysis systems in existence (such as the event-reply strategy introduced in [34]), our system is inherently portable, since the analysis process is not tied to the execution environment used to run the application.

5.2.1 Design Overview

The high-level architecture of the PPW automatic analysis system is depicted in Figure 5-2. The analyses supported by the system are categorized into two groups: application analyses which deal with performance evaluation of a single run and experiment set analyses to compare the performance of related runs. We designed this system to focus on providing analyses to help both novice and expert users in optimizing their application via source code modification. In particular, these analyses focus on finding operations that took longer than expected to run, operations that may be redundant, and operations that could be transformed into other operations to improve performance.

The parallelized analysis processing mechanism is a peer-to-peer system consists of up to N processing units (where N is the application system size): 0 to N-1 non-head processing units, each has (local) access to raw data from one or a group of PEs, and one head processing unit that require access to a small portion of data from all PEs to performs global analyses. This inherently parallel design is intended to support the analysis of large-scale applications in a reasonable amount of time. In Figure 5-3, we illustrate the types of analyses conducted and the raw data exchange needed for all processing units in an example 3-processing unit system.

2 The baseline filtering technique is new for automatic analysis but readily used in system performance evaluation.
The complete analysis process can be broken down into several distinct categories. Figure 5-4 depicts the analysis workflow for a processing unit in the system which includes common-bottleneck analysis, global analyses, frequency analysis, and bottleneck resolution. We now describe these categories in more detail in the following sections; a summary of analyses currently supported by PPW is presented in Table 5-1.

5.2.2 Common-Bottleneck Analysis

The goal of Common-Bottleneck Analysis is to identify commonly encountered performance bottlenecks which application developers have seen over the years; due to common occurrences, there are usually well-known optimization strategies that could be apply to remove these issues. For example, a common optimization strategy to remove
<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Required data type</th>
<th>Global or local</th>
<th>Related bottlenecks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability Analysis</td>
<td>Determine scalability of an application</td>
<td>Profile data (Multiple runs)</td>
<td>Global</td>
<td>Low application scalability</td>
</tr>
<tr>
<td>Revision Analysis</td>
<td>Compare performance of different revisions</td>
<td>Profile data (Multiple runs)</td>
<td>Global</td>
<td>N/A</td>
</tr>
<tr>
<td>High-Level Analysis</td>
<td>Compare comp., comm., sync among PEs</td>
<td>Profile data (All)</td>
<td>Global</td>
<td>PE-level load-balancing, Low comp/comm ratio</td>
</tr>
<tr>
<td>Block-Level Analysis</td>
<td>Detect load-balancing issue of individual program blocks</td>
<td>Tracing data (A2A)</td>
<td>Global</td>
<td>Block-level load-balancing</td>
</tr>
<tr>
<td>Event-Level Analysis</td>
<td>Detect load-balancing issue of individual event among PEs</td>
<td>Profile data (All)</td>
<td>Global</td>
<td>Event-level load-balancing</td>
</tr>
<tr>
<td>Barrier-Redundancy Analysis</td>
<td>Identify unnecessary barrier operations</td>
<td>Tracing data (A2A, data xfer)</td>
<td>Global</td>
<td>Block level load-balancing</td>
</tr>
<tr>
<td>Shared-Data Analysis</td>
<td>Evaluate data affinity efficiency</td>
<td>Profile data (Data xfer)</td>
<td>Global</td>
<td>Poor data locality</td>
</tr>
<tr>
<td>Frequency Analysis</td>
<td>Identify short-lived high-frequency operations</td>
<td>Tracing data (All)</td>
<td>Local</td>
<td>Inefficiency relating to multiple small transfers</td>
</tr>
<tr>
<td>Bottleneck Detection</td>
<td>Identify potential bottleneck locations</td>
<td>Profile data (All)</td>
<td>Local</td>
<td>Delayed operations</td>
</tr>
<tr>
<td>Cause Analysis</td>
<td>Identify causes and types of common-bottlenecks</td>
<td>Trace data (All)</td>
<td>Local</td>
<td>Bottlenecks listed in Table 5-2</td>
</tr>
</tbody>
</table>
Figure 5-3. Example analysis processing system with 3 processing units showing the analyses each processing unit performs and raw data exchange needed between processing units

Table 5-2. Common-bottleneck patterns currently supported by PPW and data needed to perform cause analysis

<table>
<thead>
<tr>
<th>Local data type</th>
<th>Bottleneck patterns</th>
<th>Request targets</th>
<th>Remote data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global sync./comm. (load-imbalance)</td>
<td>Wait on group sync./comm.</td>
<td>All other</td>
<td>Global sync./comm.</td>
</tr>
<tr>
<td>P2P lock</td>
<td>Wait on lock availability</td>
<td>All other</td>
<td>P2P unlock</td>
</tr>
<tr>
<td>P2P wait-on-value</td>
<td>Wait-on-value change</td>
<td>All other</td>
<td>One-sided data xfer</td>
</tr>
<tr>
<td>One-sided put/get</td>
<td>Competing put/get</td>
<td>All other</td>
<td>One-sided put/get</td>
</tr>
<tr>
<td>Two-sided send</td>
<td>Late sender</td>
<td>Receiver PE</td>
<td>Two-sided receive</td>
</tr>
<tr>
<td>Two-sided receive</td>
<td>Late receiver</td>
<td>Sender PE</td>
<td>Two-sided send</td>
</tr>
</tbody>
</table>
Figure 5-4. Analysis process flowchart for a processing unit in the system
Late Sender, a common-bottleneck pattern relating to two-sided data communication, is to move the send call forward in the execution sequence.

Common-bottleneck analysis is the most substantial and time-consuming analysis supported by the PPW analysis system. Unlike other knowledge-based system such as KappaPI-2 and KOJAK, our approach uses both profile and trace data in the analysis and is scalable by design. The PPW Common-Bottleneck Analysis process is separated into two phases: a Bottleneck Detection phase to identify specific bottleneck regions using only profile data; followed by a Cause Analysis phase to determine the cause of the bottleneck using trace data.

5.2.2.1 Bottleneck detection

The goal of Bottleneck Detection is to identify program regions that when optimized could improve the application performance by a noticeable amount. During this detection phase, each processing unit examines its portion of the (local) profiling data and identifies bottleneck profiling entries. For each of the profiling entries, the processing unit first checks whether or not that entry’s total execution time exceeds a preset percentage of the total application time (i.e., is a hotspot). The purpose of this filtering step is to focus the analysis effort on portions of program that would noticeably improve the performance of the application when optimized.

Next, the processing unit decides if the identified hotspot entry is a bottleneck by applying one of the following two comparison methods. With the baseline comparison method, the processing unit marks the entry as a bottleneck if the ratio of its average execution time to its baseline execution time — the minimal amount of time needed by a given operation to complete its execution under ideal circumstances — exceeds a preset threshold.

If the baseline comparison method is not applicable (e.g., because the entry is a user function or no baseline value has been collected for the entry), the processing unit uses the alternative deviation evaluation method. With this method we make use of the
following assumption: under ideal circumstances, when an event is executed multiple times, the performance of each instance should be similar to that of other instances (the same assumption is used in NoiseMiner). Thus for each hotspot entry, the processing unit calculates the ratio of its minimal execution time and of its maximum execution time to its average execution time. If one or both of the ratios exceeds a preset threshold, the processing unit marks the entry as a potential bottleneck.

5.2.2.2 Cause analysis

The list of potential bottlenecks identified in the detection phase points application developers to specific regions of code to focus their attention on but does not contain sufficient information to determine the causes of performance issues often needed to device an appropriate optimization technique. To provide these detail information, PPW’s Cause Analysis, using available trace data, aims at finding remote events that possibly caused the bottlenecks identified.

The underlying concept behind our approach is that if some remote events caused the local event to execute non-optimally (as opposed to caused by other factors such as network congestion not related to event ordering), then these remote events must have occurred between the start and end time of the local event. It is because of this concept that the amount of data exchange between processing units are minimized as only the related events that occurred during this time range need to be exchanged (compared to the event-reply strategy introduced in [34] where all relating events must be exchanged).

For example, for a upc.lock event on PE 0 with start time of 2 ms and end time of 5 ms, the request entry \{PE 0, 2ms, 5 ms, P2P unlock\} would be issued to all processing units. The logic behind this example is the following: if at the time of the lock request, another PE holds the lock, the P2P lock operation issued by PE 0 will block until it is released by the lock holder. To find out which PE(s) held the lock that caused the delay in the P2P lock operation, we simply look at the P2P unlock operations issued between the start and end time of the lock operation. If no P2P unlock operation was issued by any
other PEs, we conclude that the delay was caused by uncontrollable factors that cannot be resolved by the user, such as network congestion due to concurrent execution of other applications.

During the cause analysis phase, each processing unit carries out several activities using local tracing data in a two-pass scheme. In the first trace-log pass, the processing unit identifies trace events with source location matching any of the profiling entries discovered in the detection phase. For each matching trace event, the processing unit generates a request entry containing its name, start time, and end time along with the event’s operation type which is sent to other processing units to retrieve appropriate trace data (Table 5-2 illustrates the current set of common-bottlenecks — which is currently hard-coded — supported by our system). At the end of the first pass, the processing unit sends the requests out to all other processing units and waits for the arrival of requests from all other processing units.

Next, the processing unit makes a second pass through its trace log and generates the correct replies — consisting of \{event name, timestamp\} tuples — and sends them back to the requesting processing units. Finally, the processing unit waits for the arrival of replies and completes the cause analysis by assigning a bottleneck pattern name to each matching trace event, along with the remote operations that contributed to the delay.

In terms of execution time, we expect bottleneck detection to complete relatively quickly as the amount of profile entries is usually not large (data size in KB range). By contrast, we expect cause analysis to take significantly longer to complete due to its use of trace data; we expect the execution time of cause analysis to be linearly proportional to the number of trace events in the performance data file.

---

3 Processing units can choose to apply the filtering techniques on each trace event during this pass to further reduce the amount of data exchange needed between processing units.
5.2.3 Global Analyses

PPW supports several analyses that require a global view of the performance data; more specifically, the head processing unit needs to have access to some data from all PEs. Depending on the data types required by the analyses, the time required to carry out these analyses will vary. In the remainder of this section, we briefly discuss these global analyses which include analyses to compare the performance of multiple related experiments (scalability analysis and revision analysis) and analyses to evaluate the performance of a single run (barrier-redundancy analysis, shared-data analysis, and several analyses to evaluate load-balance among PEs).

5.2.3.1 Scalability analysis

Scalability of an application is a major concern for developers of parallel applications. With the ever-growing increase in parallel system size, it is becoming more important for applications to exhibit good scalability. Using profiling data from two or more experiments on different system sizes, PPW’s Scalability Analysis evaluates an application’s scalability (or more precisely, its parallel efficiency, the ratio of parallel performance improvement over the size increase). From the experiment with the smallest number of PEs, the head processing unit calculates the parallel efficiency for all other experiments. An efficiency of 1 indicates that the application exhibits perfect scalability, while a value approaching 0 suggests very poor scalability.

5.2.3.2 Revision analysis

During the iterative measure-modify process which a user performs to optimize his or her application, multiple revisions of an application are often produced by the user, with each revision containing code changes aimed at removing or minimizing the performance issues discovered. To assist in evaluating the performance effects of these code changes,

---

4 Note that these analyses can be performed at any time during the analysis process or as part of the other analyses.
PPW’s Revision Analysis facilitates performance comparison of the application and the 10 longest-running code regions between revisions; this analysis is used to determine whether or not code changes improved program performance and if so, what part of the program was improved.

5.2.3.3 Load-balancing analyses

Achieving good work distribution among PEs is difficult and often impacts the performance and scalability of the application significantly. To help in this aspect, PPW provides several analyses to investigate an application’s workload distribution at different levels. At the highest level, PPW’s High-Level Analysis calculates and compares the total computation, communication, and synchronization time among PEs. Since the PEs with largest computation time (i.e., highest workload) often determine the overall performance of the application, this analysis assists in the identification of bottleneck PEs (PEs that when optimized improve the overall application performance).

Next, the Block-Level Analysis aims at identifying specific program blocks with uneven work distribution and thus further guides users to parts of the program where they should focus their efforts. By ensuring that all program blocks have good load-balance, the user essentially achieved good load balance for the entire application.

Finally, at the lowest level, the Event-Level Analysis compares the workload of individual events (i.e., a specific line of code or code region) among PEs which is extremely useful when the event under investigation represents workload that was meant to be parallelized or is a global synchronization event (as an uneven global synchronization often stems from uneven workload distribution prior to the synchronization call).

---

5 A program block is defined as a segment of code between one global synchronization to the next similar to a block in the Bulk Synchronous Parallelism (BSP) computation model.
To ensure program correctness, programmers may often insert extra global barrier calls that give rise to the performance degradation depicted in Figure 5-5. To detect potentially redundant barriers, PPW’s Barrier-Redundancy Analysis examines the shared data accesses between barrier calls and identifies calls with no shared data accesses between the target call and the one before it as redundant. The idea is that since barriers are often used to enforce global memory consistency, a barrier call with no prior shared data accesses may not be needed. The output of this analysis is a list of potential redundant barrier calls (with source information) that user may consider removing from the program.

**5.2.3.5 Shared-data analysis**

Data affinity (or locality) is a very important factor in parallel programming and it is often a major deciding factor between a good and a poor performing parallel application. To assess an application’s data locality efficiency, PPW’s Shared-Data Analysis measures the ratio of local-to-remote access\(^6\) for all PEs and combines them into a single data access ratio that could be used to determine the application locality efficiency (typically, the higher the ratio, the better the program). This analysis could be refined to analyze the
locality efficiency of a specific shared region (such as UPC shared array) when the tool knows the specific memory regions that a particular data communication call touches. In the case of UPC, this refinement is extremely useful as it allows the determination of the best blocking factor leading to minimized remote data access on all PEs (part of the Bottleneck Resolution).

5.2.4 Frequency Analysis

The existence of short-lived, high-frequency events (henceforth referred to simply as high-frequency events) can affect the accuracy of the performance data collected, so it is useful to identify these high-frequency events that should not be tracked during the subsequent data collection process. More importantly, high-frequency events sometimes represent events which are highly beneficial to optimize (since they are called many times) or in the case of data communication operations, could potentially be transformed into more efficient bulk transfer operations (a major known optimization technique as illustrated in Figure 5-6). For these reasons, PPW includes a memory-bound Frequency Analysis aimed at identifying high-frequency events. By making a pass through the trace data, this analysis identifies a list of high-frequency events for each PE.

5.2.5 Bottleneck Resolution

In the final step of the analysis process, Bottleneck Resolution\(^7\), the processing unit aims at identifying hints useful to the user in removing the bottlenecks identified in one of the previous analyses (Table 5-3). This process is the only part of the system that may need to be model-dependent, as a given resolution strategy may not always work for all programming models. For example, a technique to fix the performance degradation

---

\(^6\) Note that due to factors such as variable aliasing, it may be very difficult to collect performance data relating to local accesses (and it is even more difficult to keep track of specific memory addresses being accessed) and thus not possible to carry out this analysis for some programming model implementations.
Figure 5-6. Frequency analysis

stemming from `upc_memget`, versus from `shmem_get`, could be different even though they are both classified as one-sided get operations.

One example of a model-specific resolution technique is the identification of the best blocking factor to use in declaring a high-affinity UPC shared array. When the system detects an excessive communication issue associated with a shared array, the processing unit would try to find an alternative blocking factor that would yield the best local-to-remote memory access ratio for all PEs in the system.

5.3 Prototype Development and Evaluation

Several analysis system prototypes supporting UPC, SHMEM, and MPI were developed and integrated into the latest version of the PPW tool. These prototypes add to PPW a number of analysis components, corresponding to those shown in Figure 5-2, to perform the necessary processing, management of analysis data, and presentation of analysis results to the tool user. To perform any of the analyses, the user brings up the analysis user interface (Figure 5-7), selects the desired analysis type, and adjusts any parameter values (such as percentage program threshold that defines the minimum hotspot percentage) if desired. Once all the analyses are completed, the results are sent to an analysis visualization manager which generates the appropriate visualizations.

---

7 Bottleneck resolution is currently an open research area.
<table>
<thead>
<tr>
<th>Bottleneck pattern</th>
<th>Potential resolution techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait on group sync./comm.</td>
<td>Modify the code to achieve better work distribution</td>
</tr>
<tr>
<td></td>
<td>Use multiple point-to-point synchronization operations</td>
</tr>
<tr>
<td>Wait on lock availability</td>
<td>Perform more local computation before the wait-on-lock operation</td>
</tr>
<tr>
<td></td>
<td>Use multiple locks if appropriate</td>
</tr>
<tr>
<td>Wait-on-value change</td>
<td>Perform more local computation before the wait-on-value operation</td>
</tr>
<tr>
<td>Competing put/get</td>
<td>Use non-blocking put/get</td>
</tr>
<tr>
<td>Late sender</td>
<td>Perform less local computation before the local send operation</td>
</tr>
<tr>
<td></td>
<td>Perform more local computation before the remote receive operation</td>
</tr>
<tr>
<td></td>
<td>Use non-blocking receive</td>
</tr>
<tr>
<td>Late receiver</td>
<td>Perform less local computation before the local receive operation</td>
</tr>
<tr>
<td></td>
<td>Perform more local computation before the remote send operation</td>
</tr>
<tr>
<td></td>
<td>Use non-blocking send</td>
</tr>
<tr>
<td>Consecutive blocking data transfers to unrelated targets (i.e., different PEs, different memory addresses on same PE)</td>
<td>Use non-blocking data transfers</td>
</tr>
<tr>
<td></td>
<td>Use bulk transfer operation if appropriate</td>
</tr>
<tr>
<td>Multiple small data transfers to same PE</td>
<td>Combine multiple small transfers into a single bulk transfer</td>
</tr>
<tr>
<td>Poor data locality</td>
<td>Modify shared data layout (e.g., use different blocking factor in UPC)</td>
</tr>
</tbody>
</table>
To acquire the appropriate baseline values needed for the baseline filtering technique, we created a set of bottleneck-free benchmark programs for each of the supported models. These benchmarks are then executed on the target system, and the generated data files are processed to extract the baseline value for each model construct.

PPW provides several new analysis visualizations to display the generated analysis results. To facilitate experiment set analyses, a scalability-analysis visualization that plots the calculated parallel efficiency values against the ideal parallel efficiency (Figure 5-8) and a revision-comparison visualization that facilitates side-by-side comparison of observed execution times for regions within separate versions of an application (Figure 5-9) are supported. To visualize analysis result of a single experiment, PPW includes a high-level analysis visualization displaying the breakdown of computation, communication, and synchronization time for each PE executing an application to evaluate the workload distribution at a high level (Figure 5-10), an event-level load-balance visualization to compare the workload of individual events across PEs (Figure 5-11), and a multi-table...
analysis visualization which displays the result from common bottleneck detection and
cause analysis supplemented with source-code correlation (Figure 5-12). Finally, PPW
generates a text-based report provides a summary of the analyses performed; this report
includes information such as the speed of analysis, the parameter values used, number of
and list of bottlenecks found on each PEs, and results from several analyses (block-level
load-balancing analysis, frequency analysis, barrier-redundancy analysis, shared-data
analysis) not displayed in the analysis visualizations just mentioned (Figure 5-13).

In the remainder of this section, we present details of the sequential, threaded, and
distributed prototypes developed and supply experimental results regarding the speed of
these prototypes.

5.3.1 Sequential Prototype

The proposed analysis system was first developed as part of the PPW Java front-end
to reflect a common PPW use case illustrated in Figure 5-14 where the user collects
Figure 5-9. PPW revision-comparison visualization

Figure 5-10. Annotated PPW high-level analysis visualization
application performance data on the parallel system using the PPW back-end, transfers
the combined performance data file to a personal workstation, and then visualizes the
collected data using the PPW front-end system.

In this initial prototype, a single processing unit was used to conduct all of the
selected analyses in a sequential fashion illustrated in Figure 5-15a, using main memory to
store intermediate (i.e., request and reply) and result data. To validate the correctness of
this prototype, we created a set of test programs written in UPC, SHMEM, and MPI in a
method similar that discussed in [40]. This analysis test suite consists of control programs
with no bottlenecks and test programs which each contain a bottleneck pattern of interest.
We applied the analysis process on these programs and verified that the system is able to
detect the target bottlenecks correctly.

In Table 5-4, the speed of the analysis for several of the NAS 2.4 benchmarks
executed with 128 or 256 PEs is shown. The testbed for this experiment is an Intel
Core i7 Quad-core (with Hyper-Threading support) 2.66 GHz processor workstation with 6
GB of RAM running 64-bit Windows 7. As expected, the analysis speed for trace-related
Figure 5-12. Annotated PPW analysis table visualization
Figure 5-13. Annotated PPW analysis summary report
analyses dominates the overall execution time (profile-based analyses all took less than 1 ms); we observed that the analysis speed is linearly proportional to the number of trace events (0.15-0.2 million trace events per minute).

5.3.2 Threaded Prototype

While these initial performance results were encouraging, we quickly realized that the sequential approach would not suffice for two reasons. First, the largest data size that the sequential prototype could analyze is limited by the amount of memory available; our attempt to analyze a 128-PE run of the CG benchmark (31.5 million trace events) was unsuccessful due to this reason. Second, the time required to complete the analysis may become unreasonably long for much larger data size; it may take hours or even days before the user can see the result of the analysis of an experiment with huge amount of trace events.
Figure 5-15. Analysis workflow for the (a) sequential prototype (b) threaded prototype (c) distributed prototype
Table 5-5. Analysis speed of NPB benchmarks on workstation

<table>
<thead>
<tr>
<th>Num. threads</th>
<th>FT</th>
<th>MG</th>
<th>EP (128)</th>
<th>EP (256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (seq.)</td>
<td>3821 s (63.7 min)</td>
<td>1705 s (28.4 min)</td>
<td>0.68 s</td>
<td>2.27 s</td>
</tr>
<tr>
<td>2</td>
<td>2007 s (33.5 min)</td>
<td>1128 s (18.8 min)</td>
<td>0.37 s</td>
<td>1.10 s</td>
</tr>
<tr>
<td>4</td>
<td>1263 s (21.1 min)</td>
<td>709 s (11.8 min)</td>
<td>0.41 s</td>
<td>0.78 s</td>
</tr>
<tr>
<td>8</td>
<td>1026 s (17.1 min)</td>
<td>603 s (10.1 min)</td>
<td>0.39 s</td>
<td>0.81 s</td>
</tr>
<tr>
<td>16</td>
<td>1234 s (20.6 min)</td>
<td>626 s (10.4 min)</td>
<td>0.79 s</td>
<td>1.35 s</td>
</tr>
</tbody>
</table>

Fortunately, since the design of PPW analysis system is inherently parallel, we were able develop parallel versions of our system to address these two issues; we developed a threaded prototype to take advantage of the dominating multi-core workstation architecture and a fully distributed prototype that can execute on a large cluster.

The modified analysis process of the (Java-based) threaded prototype is illustrated in Figure 5-15b. In this threaded prototype, each processing unit (1 to K) is assigned a group of PEs (1 to N) and is responsible for carrying out all the analyses for that group of PEs. The results produced by the threaded prototype were validated against those produced by the sequential prototype, and we again ran the analysis of NAS benchmarks on the Core i7 workstation to measure the analysis speed. The results are shown in Table 5-5; from this table, we see that the analysis speed (for reasonably sized data files) scales fairly well up to the number of cores (1 to 2 and 4 threads), shows a slight improvement (4 to 8 threads) using Hyper-Threading, and slows down somewhat when the thread count exceeded the number of processing units (16 threads). The analysis of the CG benchmark was again unable to complete as the threaded prototype also uses main memory to store all intermediate and result data structures.

5.3.3 Distributed Prototype

We have shown in the previous section that a threaded version of the analysis improves the speed of analysis fairly well up to the number of cores on the workstation. However, since the number of cores is limited on a single machine, we continued our prototyping effort to develop a version of the PPW analysis system capable of running on cluster systems that could contain thousands of PEs. There are several reasons for
Figure 5-16. A use case of PPW where analyses are performed on the parallel systems developing a distributed version of the analysis system. First, the distributed version is more scalable than the threaded version; it can support the analysis of larger data runs and improves the analysis speed further due to increase amount of available processors. Second, the distributed analysis process can now be executed as a batch job or as part of the data collection process as shown in Figure 5-16. When running as part of the data collection process, the result of the analysis could potentially be used to reduce the raw data size and thus improve the scalability of the PPW tool itself.

The workflow for the distributed prototype (Figure 5-15c) is very similar to that of the threaded prototype except now each processing unit is assign to process data of a single PE (each processing unit has local access to assigned PE’s data) and intermediate data (requests and replies) must now be exchange across the network. As shown in Figure 5-17, the amount of memory space required on each processing unit is reduced (from $NxNxM$ requests and replies to $2xNxM$ requests and replies) and is now able to support larger data file such as CG (contain 245974 trace events per PE) which was unable to run
Table 5-6. Analysis speed of NPB benchmarks on Ethernet-connected cluster

<table>
<thead>
<tr>
<th>Num. PEs</th>
<th>FT</th>
<th>MG</th>
<th>CG</th>
<th>EP</th>
<th>EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 proc. unit</td>
<td>2113 s</td>
<td>1019 s</td>
<td>N/A</td>
<td>0.15 s</td>
<td>0.85 s</td>
</tr>
<tr>
<td></td>
<td>(35.2 min)</td>
<td>(17.0 min)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PE proc. units</td>
<td>32.5 s</td>
<td>242 s</td>
<td>16668 s</td>
<td>6.38 s</td>
<td>40.12 s</td>
</tr>
<tr>
<td></td>
<td>(0.5 min)</td>
<td>(4.0 min)</td>
<td>(4.63 hrs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speedup</td>
<td>65.02</td>
<td>4.21</td>
<td>-</td>
<td>0.02</td>
<td>0.02 s</td>
</tr>
</tbody>
</table>

successfully on the Core i7 workstation. The results produced by the distributed prototype were again validated against those produced by the sequential prototype, and in Table 5-6 we show the analysis speed of the NAS benchmarks on an 80-PE Quad-core Xeon Linux cluster connected using MPICH-2 1.0.8 over Ethernet.

We made several observations from this data. We saw that the sequential analysis speed improved almost by a factor of 2 due to move from a Java-based to a C-based environment. More importantly, the analysis speed of the parallel version (128 or 256 processing units) is greatly improved for larger data files. We saw that the analysis speed for EP (82 trace events per PE) worsened but this behavior was expected as there is simply not enough work to be distributed. In the case of MG (44670 trace events per PE), the analysis speed improved by a factor of 4. Finally in the case of FT (72230 trace events per PE, more bottlenecks undergoing cause analysis than MG), the analysis speed was
improved by almost two orders of magnitude, demonstrating the performance benefit of the distributed prototype. We expect the performance improvement to be more apparent on systems with high-speed interconnects and for experiments with a larger number of trace events per PE.

5.3.4 Summary of Prototype Development

We have developed several versions of the PPW analysis system and provided experimental data on the speed of analysis. We observed that the analysis speed (in all versions) is dependent on the size of the trace data as expected but also affected by the number of bottlenecks undergoing cause analysis. We have shown the correctness of the PPW system design using a synthetic analysis test suite and proved the scalability of the design by demonstrating the analysis speed improvement of both threaded and distributed prototypes over the initial sequential version. We noted that while the sequential prototype, and to lesser extent the threaded prototype, exhibits some scalability issues, it is not without use. For analysis of experiments with small to moderate amount of data, the workstation prototypes are sufficient in completing the analysis in a reasonable amount of time. However, when the number of trace events per PE exceeds a certain amount, user of the PPW analysis system should use the more efficient distributed prototype.

5.4 Conclusions

The goal of the second part of this research was to investigate, design, develop, and evaluate a scalable, model-independent automatic analysis system. Performance-tool-assisted manual analysis facilitates the cumbersome application optimization process but does not scale. As the size of the performance dataset grows, it becomes nearly impossible for the user to manually examine the data and find performance issues using the visualizations provided by the tool. This problem exposes the need for an automatic analysis system that can detect, diagnose, and potentially resolve bottlenecks. While several automatic
analysis approaches have been proposed, each has particular drawbacks that limit its effectiveness or applicability.

To address this issue, we developed the model-independent PPW automatic analysis system that supports a variety of analyses. We presented the architecture of the PPW analysis system, introduced novel techniques such as the baseline filtering technique to improve detection accuracy, and discussed the scalable analysis processing mechanism designed to support large-scale application analysis. We showed correctness and performance results for a sequential version of the system that has been integrated into the PPW performance tool and then demonstrated the parallel nature of the design and its performance benefits in the discussion of the threaded and distributed versions of the system.

Future work for this system includes experimental evaluation on a larger parallel system, enhancements to the existing analyses (e.g. use temporary files to reduce memory requirements, faster algorithms to improve speed of trace analyses), support for additional analyses such as bottleneck resolution, expansion of the number of common-bottleneck patterns the system detects, and development of functionality to allow users to define new bottlenecks themselves.
CHAPTER 6
EXPERIMENTAL EVALUATION OF
PPW-ASSISTED PARALLEL APPLICATION OPTIMIZATION PROCESS

In this chapter, we present studies used to evaluate the effectiveness of the proposed PPW framework and automatic analysis system.

6.1 Productivity Study

To assess the usefulness and productivity of PPW, we conducted a study with a group of 21 graduate students who had a basic understanding of UPC programming but were unfamiliar with the performance analysis process. Each student was asked to spend several hours conducting manual (via the insertion of printf statements) and tool-assisted (using a version of PPW without automatic analysis support) performance analysis with a small UPC cryptanalysis program called CAMEL (with approximately 1000 lines of code) known to have several performance bottlenecks. Students were told to concentrate their effort on finding and resolving only parallel bottlenecks.

The results demonstrated that PPW was useful in helping programmers identify and resolve performance bottlenecks (Figure 6-1). On average, 1.38 bottlenecks were found with manual performance analysis while 1.81 bottlenecks were found using PPW. All students were able to identify at least as many bottlenecks using PPW, and one third of them identified more bottlenecks using PPW. In addition, most students noted that they had an easier time pinpointing the bottlenecks using PPW. However, only six students were able to correctly modify the original code to improve its performance (with an average performance gain of 38.7%), while the rest either performed incorrect code transformations or were unable to devise a strategy to fix the issues. This inability to modify the original code was not surprising, since the students were not familiar with the algorithms used in the CAMEL program, were novices with respect to parallel programming, and were asked to spend only a few hours on the task.

Students were also asked to compare the experiences they had with both approaches in terms of code analysis (bottleneck identification) and optimization. Overall, PPW was
viewed as a helpful tool by students, with most students preferring PPW over manual performance analysis for reasons listed below (summarized from student feedback).

- Manual insertion and deletion of timing calls is tedious and time-consuming. While not significantly difficult in this case, it can potentially be unmanageable for large applications with tens of thousands of lines of code.

- A significant amount of effort was needed in determining where to insert the timing calls, a process which was automated in PPW.

- Visualizations provided by the tool were much more effective in pinpointing the source of bottlenecks and even more so in determining the cause of the bottlenecks.

6.2 FT Case Study

For the first application case study, we ran the Fourier Transform (FT) benchmark (which implements a Fast Fourier Transform algorithm) from the NAS benchmark suite version 2.4 using GASP-enabled Berkeley UPC version 2.6. Initially no change was made to the FT source code, and the performance data were collected for the class B setting executed using 16 PEs on an Opteron cluster with Quadrics QsNet II high-speed interconnects.

From the Tree Table (Figure 6-2), it was immediately obvious that the fft function call (3rd row) constituted the bulk of the execution time (18s out of 20s of total execution time). Further examination of performance data for events within the fft function revealed
the \texttt{upc\_barrier} operations (represented as \texttt{upc\_notify} and \texttt{upc\_wait}) in \texttt{transpose2\_global} (6th row) as potential bottleneck locations. We came to this conclusion by observing that the actual average execution times for \texttt{upc\_barrier} at lines 1943 (78.71ms) and 1953 (1.06s) far exceed the expected value of 2ms on our system for 16 PEs (we obtained the expected value by running a simple benchmark). Looking at the code between the two barriers, we saw that multiple \texttt{upc\_memget} operations were issued and speculated that the bottleneck was related to these operations. However, we are unable to verify this speculation and determine the cause of this bottleneck based solely on this statistical data.

Thus, we then converted the trace data into the Jumpshot SLOG-2 format and looked at the behavior of \texttt{upc\_barrier} and \texttt{upc\_memget} operations in a timeline view. We discovered that the \texttt{upc\_barrier} at line 1953 was waiting for the \texttt{upc\_memget} operation.
to complete. In addition, we saw that \texttt{upc\_memget} operations issued from the same PE were unnecessarily serialized, as shown in the annotated Jumpshot screenshot (Figure 6-3; note the zigzag pattern for \texttt{memget} operations). Looking at the start and end times of the \texttt{upc\_memget} operations issued from PE 0 to all other PEs (see the info box in Figure 6-3), we saw that the later \texttt{upc\_memget} operations must wait for the earlier \texttt{upc\_memget} operations to complete before initiating, even though the data obtained were from different sources and stored locally at different private memory locations.

A solution to improve the performance of the FT benchmark is to use a non-blocking (asynchronous) bulk-transfer get such as \texttt{bupc\_memget\_async} provided by Berkeley UPC.
When this code transformation\(^1\) was made (shown in the lower portion of Figure 6-4), we were able to improve the performance of the program by 14.4% over the original version.

We later applied the automatic analysis process to the same FT data file to check whether or not the analysis system could find the bottlenecks that we identified. Looking at the multi-table analysis visualization, we saw that the system found 4 bottlenecks, including the most significant `upc_barrier` bottleneck. In addition, the system was able to determine the cause of delay for each occurrence of the barrier operation that took longer than expected. For example, the system found that the barrier called by PE 7 with a starting time of 2.6s took longer than expected to execute because PEs 8 and 15 entered the barrier later than PE 7 (Figure 6-5, left). As we observed in the annotated

\(^1\) Note that while this code transformation is not portable to other compilers, the optimization strategy of using non-blocking transfer is portable.
Jumpshot view (Figure 6-5, right), this pattern is verified. Switching to the high-level analysis visualization, we saw that each PE spent 5 to 15% of the total execution time inside the barrier call, further validating the existence of a barrier-related bottleneck. This percentage drops to 1 to 2% of the total execution time for the revised version using the non-blocking get operation.

In this case study, we have shown how PPW was used to optimize a UPC program. With little knowledge of how the FT benchmark works, we were able to apply the manual analysis process and remove a major bottleneck in the program within a few hours of using PPW. In addition, we showed that our automatic analysis system was able to correctly identify and determine the cause of significant bottlenecks in the FT benchmark.

### 6.3 SAR Case Study

For the second application case study, we performed analysis of both UPC and SHMEM in-house implementations of the Synthetic Aperture Radar (SAR) algorithm using GASP-enabled Berkeley UPC version 2.6 and Quadrics SHMEM on an Opteron cluster with a Quadrics QsNetII interconnect. SAR is a high-resolution, broad-area imaging processing algorithm used for reconnaissance, surveillance, targeting, navigation, and other operations requiring highly detailed, terrain-structural information. In this algorithm, the raw image gathered from the downward-facing radar is first divided into patches with overlapping boundaries so they can be processed independently from each other. Each patch then undergoes a two-dimensional, space-variant convolution that can be decomposed into two domains of processing, the range and azimuth, to produce the result for a segment of final image (Figure 6-6).

The sequential version from Scripps Institution of Oceanography and MPI version provided by two fellow researchers in our lab [26] were used as the templates for the development of UPC and SHMEM versions. The MPI version follow the master-worker approach where the master PE reads patches from the raw image file, distributes patches for processing, collects result from all PEs, and writes the result to an output file, while
Figure 6-5. Multi-table analysis visualization for FT benchmark with annotated Jumpshot visualization
the worker PEs perform the actual range and azimuth computation on the patches (note: master PEs also perform computation). For this study, we used a raw image file with parameters set to create 35 patches, each of size 128MB. While all patches could be executed in parallel in a single iteration on a system with more than 35 PEs, smaller systems, such as our 32-PE cluster, execute over multiple iterations (in each iteration, M patches are processed where M equals to the number of computing PEs). We assume only sequential I/O is available throughout the study, a fair assumption since neither UPC nor SHMEM currently includes standardized parallel I/O.

We began this case study by developing a UPC baseline version (which mimics the MPI version) using a single master PE to handle all the I/O operations and that also performs processing of patches in each iteration. Between consecutive iterations, all-to-all barrier synchronization is used to enforce the consistency of the data. After verifying the correctness of this version, we used PPW to analyze the performance on three systems sizes of computing PEs: 6, 12, and 18; these system sizes were chosen so that in each iteration, at most one worker PE is not performing any patch processing. By examining several visualizations in PPW (one of which is the Profile Metrics Bar Chart shown in Figure 6-7), we noticed that with 6 computing PEs, 18.7% of the execution time was spent inside the barrier and that the percentage increased with the number of computing PEs (20.4% for 12 PEs, 27.6% for 18 PEs). Using the timeline view to further investigate the issue (Figure 6-8), we then concluded that the cause of this bottleneck was that worker
PEs must wait until the master PE writes the result from the previous iteration to storage and sends the next patches of data to all processing PEs before they can exit the barrier.

Similar findings were seen when automatic analysis is applied. From the high-level analysis visualization (Figure 6-9), we observed that a significant amount of time is lost performing global synchronization. This observation is reconfirmed by examining the multi-table analysis visualization which lists two `shmembARRIER_all` bottlenecks.

We devised two possible optimization strategies to improve the performance of the baseline version. The first strategy was the use of dedicated master PE(s) (performing no patch processing) to ensure that I/O operations could complete as soon as possible. The second strategy was to replace all-to-all barrier synchronization with point-to-point flag synchronization (implementing the wait-on-value-change operation) so processing PEs could work on the patches as early as possible. We expected that the first approach would yield a small performance improvement while the second approach should greatly alleviate the issue identified.

We then developed five revisions of the program using one or both of these strategies: (1) dedicated-master, (2) flag synchronization, (3) dedicated master with flag synchronization, (4) two dedicated masters (one for read, one for write), and (5) two dedicated masters and flag synchronization. These revisions were again run on system sizes with 6, 12, and 18 PEs, and the performance of the revisions was compared to that of the baseline version (Figure 6-10). As expected, the dedicated master strategy alone did not improve the performance of the application. Surprisingly, the flag synchronization strategy by itself also did not improve the performance as we expected. After some investigation, we discovered that while we eliminated the barrier wait time, we introduced the same amount of idle time waiting for the shared flags to be set (Figure 6-11). The combination of both strategies, however, did improve the performance of the program by a noticeable amount, especially in the two dedicated masters and flag synchronization version where the percentage of patch execution time increased (from 77.95%, 78.09%, 70.71% for the
Figure 6-7. Performance breakdown of UPC SAR baseline version run with 6, 12, and 18 computing PEs annotated to show percentage of execution time associated with barriers.
baseline version) to 97.05%, 94.38%, and 87.97% of total time for 6, 12, and 18 processing PEs respectively (the remaining time is mainly spent on unavoidable sequential I/O and bulk data transfer). This observation was verified when we looked at the high-level analysis visualization (Figure 6-12) and saw that all PEs spent the majority of their time performing computation.

This case study was then performed using SHMEM implementations of SAR based on the same approaches outlined above for the UPC version (performance comparison for these versions are also shown in Figure 6-10). For SHMEM, we noticed that the dedicated master strategy improved the performance by a small amount, while the flag synchronization strategy still did not help. The combination of both strategies again improved the performance by a noticeable percentage, with the two dedicated masters and flag synchronization version exhibiting 6.1%, 13.6%, and 15.8% improvement over the baseline version for 6, 12, and 18 PEs respectively.
Figure 6-9. High-level analysis visualization for the original version (v1) of SAR application with load-imbalance issue

Figure 6-10. Observed execution time for various UPC and SHMEM SAR reversions
Figure 6-11. Timeline view of UPC SAR flag synchronization version executed on system with 6 computing PEs annotated to highlight wait time of flags

Additionally, PPW enabled us to observe that the performance of the SHMEM versions was 15-20% slower than the corresponding UPC versions (Table 6-1)). This observation was surprising since we used the same Quadrics interconnect with communication libraries built on top of the same low-level network API. We examined the performance of the data transfers for UPC and SHMEM versions and found that the performance of these operations is actually better in SHMEM. After some investigation, we determined that the difference between the two versions came from the significant increase in execution time for read and write of data and patch processing functions (i.e., the azimuth and range functions) in the SHMEM versions. We concluded that this behavior is most likely due to the overhead introduced by the Quadrics SHMEM library to allow access to the shared memory space, which incurred even for accesses of data physically residing on the calling PE. For UPC, a cast of shared pointer to local pointer made before entering these
functions eliminates the overhead associated with global memory access (not available in SHMEM).

In this case study, we have shown how PPW was used to facilitate the optimization process of an in-house SAR application. We were able to use PPW to discover performance bottlenecks, compare the performance of both UPC and SHMEM versions side-by-side, and discover properties of the Quadrics SHMEM environment that should be considered when dealing with global memory access.

6.4 Conclusions

The goal of the third part of this research was to experimentally evaluate the PPW-assisted parallel application optimization process. Through several case studies, we assessed the effectiveness of the PPW framework and the analysis system presented in previous chapters. A classroom productivity study was conducted and the result
Table 6-1. Performance comparison of various versions of UPC and SHMEM SAR programs

<table>
<thead>
<tr>
<th></th>
<th>UPC 6 PEs</th>
<th>SHMEM 6 PEs</th>
<th>%Diff</th>
<th>UPC 12 PEs</th>
<th>SHMEM 12 PEs</th>
<th>%Diff</th>
<th>UPC 18 PEs</th>
<th>SHMEM 18 PEs</th>
<th>%Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120.3s</td>
<td>149.8s</td>
<td>24.6</td>
<td>68.3s</td>
<td>84.5s</td>
<td>23.6</td>
<td>52.1s</td>
<td>63.4s</td>
<td>21.8</td>
</tr>
<tr>
<td>Transfer</td>
<td>4.3s</td>
<td>3.0s</td>
<td></td>
<td>6.1s</td>
<td>3.4s</td>
<td></td>
<td>7.8s</td>
<td>3.5s</td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120.9s</td>
<td>151.6s</td>
<td>25.4</td>
<td>68.3s</td>
<td>84.4s</td>
<td>23.6</td>
<td>51.8s</td>
<td>63.0s</td>
<td>21.6</td>
</tr>
<tr>
<td>Transfer</td>
<td>5.2s</td>
<td>3.7s</td>
<td></td>
<td>6.7s</td>
<td>3.7s</td>
<td></td>
<td>8.2s</td>
<td>3.7s</td>
<td></td>
</tr>
<tr>
<td>Flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120.8s</td>
<td>152.0s</td>
<td>25.8</td>
<td>68.2s</td>
<td>85.3s</td>
<td>24.9</td>
<td>51.5s</td>
<td>63.6s</td>
<td>23.4</td>
</tr>
<tr>
<td>Transfer</td>
<td>4.3s</td>
<td>3.0s</td>
<td></td>
<td>6.1s</td>
<td>3.4s</td>
<td></td>
<td>7.7s</td>
<td>3.5s</td>
<td></td>
</tr>
<tr>
<td>Master/Flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>118.7s</td>
<td>148.2s</td>
<td>24.8</td>
<td>65.7s</td>
<td>81.3s</td>
<td>23.8</td>
<td>49.6s</td>
<td>59.6s</td>
<td>20.0</td>
</tr>
<tr>
<td>Transfer</td>
<td>5.2s</td>
<td>3.7s</td>
<td></td>
<td>6.7s</td>
<td>3.7s</td>
<td></td>
<td>8.2s</td>
<td>3.7s</td>
<td></td>
</tr>
<tr>
<td>2 masters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>121.6s</td>
<td>150.5s</td>
<td>23.7</td>
<td>68.8s</td>
<td>84.0s</td>
<td>22.1</td>
<td>53.8s</td>
<td>61.4s</td>
<td>14.2</td>
</tr>
<tr>
<td>Transfer</td>
<td>5.2s</td>
<td>3.7s</td>
<td></td>
<td>6.7s</td>
<td>3.7s</td>
<td></td>
<td>8.2s</td>
<td>3.7s</td>
<td></td>
</tr>
<tr>
<td>2 masters/flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>113.7s</td>
<td>140.1s</td>
<td>23.3</td>
<td>60.7s</td>
<td>74.0s</td>
<td>21.9</td>
<td>45.7s</td>
<td>53.2s</td>
<td>16.4</td>
</tr>
<tr>
<td>Transfer</td>
<td>5.2s</td>
<td>3.7s</td>
<td></td>
<td>6.7s</td>
<td>3.7s</td>
<td></td>
<td>8.2s</td>
<td>3.7s</td>
<td></td>
</tr>
</tbody>
</table>
illustrated that most students preferred PPW over the `printf`-style performance analysis. In the FT case study, we showed how PPW assisted the manual analysis process and verified that the analysis system was able to correctly determine and find causes of bottlenecks identified during the manual analysis process. In the SAR case study, we demonstrated how the complete PPW tool (with manual and automatic analysis) was used in tuning an in-house, inefficient first implementation of SAR to yield an optimized application.
Researchers from many scientific fields have turned to parallel computing in pursuit of the highest possible application performance. Unfortunately, due to the combined complexity of parallel execution environments and programming models, applications must often be analyzed and optimized by the programmer before reaching an acceptable level of performance. Many performance tools were developed to facilitate this non-trivial analyze-optimize process but have traditionally been limited in programming model support; existing tools were often developed to specifically target MPI and thus are not easily extensible to support alternative programming models. To fill this need, we presented work on what we believe to be the first general-purpose performance tool system, the Parallel Performance Wizard (PPW) system, in this dissertation.

We first presented the PPW framework and discussed novel concepts to improve tool extensibility. We introduced the generic-operation-type abstraction and the GASP-enabled data collection process developed to minimize the dependence of the tool on its supported models, making the PPW design highly extensible to support a range of programming models. Using this framework, we created the PPW performance system that fully supports the much needed PGAS models (i.e., UPC and SHMEM) as well as MPI. Results from our experimental studies showed that our PPW system incurred an acceptable level of overhead and is comparable to other popular performance tools in terms of overhead and storage requirement. In addition, we demonstrated that our PPW system is scalable up to at least 512 processing elements.

We next presented a new scalable, model-independent analysis system to automatically detect and diagnose performance bottlenecks. We introduced new techniques to improve detection accuracy, discussed a range of new and existing analyses developed to find performance bottlenecks, and discussed a parallelized analysis processing mechanism designed to support large-scale application analysis. We validated the correctness of our
analysis system design using a test suite designed to verify the system’s capability in
detecting specific bottlenecks. We then demonstrated the parallelized nature of the design
by successfully developing both the threaded and distributed versions of the system. We
showed the performance improvement of the parallel versions over the sequential version;
in one case, we illustrated that the analysis speed was improved by almost two orders of
magnitude (from 35 minutes to 35 seconds).

Finally, we presented several case studies to evaluate the PPW framework and
the analysis system. In the classroom productivity study, we demonstrated that PPW
was viewed as a useful tool in helping programmers identify and resolve performance
bottlenecks. On average, participants were able to find more bottlenecks using PPW and
most participants noted that they had an easier time pinpointing bottlenecks using PPW.
In the FT case study, we first demonstrated how PPW was used in the manual analysis
process to improve the performance of the original FT benchmark by 14.4% within a few
hours of use and later showed that our automatic analysis system was able to correctly
identify and determine the cause of significant bottlenecks found during the manual
analysis process. In the SAR case study, we illustrated how the complete PPW system
was used to discover performance bottlenecks, compared the performance of both UPC
and SHMEM versions side-by-side, and discovered properties of the Quadrics SHMEM
environment that should be considered when dealing with global memory access.

The main contributions of this research include the PPW general-purpose performance
tool system for parallel application optimization and a scalable automatic analysis system.
With the creation of our PPW tool and infrastructure, we brought performance tool
support to the much needed UPC and SHMEM programming models and made it easier
to bring performance tool support to other/developing programming models. In addition,
we contributed to the on-going automatic performance analysis research by developing
a scalable and portable automatic analysis system and introducing new techniques and
analyses to find performance bottlenecks faster and more accurately.
There are several future areas of research related to this work. First, we have thus far tested and evaluated the PPW system on parallel systems up to hundreds of processing elements. To keep up with the ever-growing parallel system size, we plan to experimentally evaluate the usefulness of PPW on larger parallel systems (thousands of processing elements) and develop strategies to resolve potential scalability issues with the PPW system. Second, we are interested in extending the PPW system to support newer programming models; we have already start working on enabling support for X10 [41].

Third, we are interested in enhancing the capability of the automatic analysis system by developing algorithms to further improve the analysis speed and accuracy, investigating techniques to automatically resolve bottleneck, and developing mechanisms to allow users to define bottlenecks themselves. Finally, we will continue to work on improving the usability of PPW. For example, we are currently working on integrating PPW into the Eclipse development environment and we are planning to develop mechanisms to provide lower-level performance information using GASP.
A.1 Introduction

In this Appendix, we include the an adapted version of the GASP interface (version 1.5). The authors of this specification are Adam Leko, Hung-Hsun Su, and Alan D. George from the Electrical and Computer Engineering Department at the University of Florida and Dan Bonachea from the Computer Science Division at the University of California at Berkeley.

A.1.1 Scope

Due to the wide range of compilers and the lack of a standardized performance tool interface, writers of performance tools face many challenges when incorporating support for global address space (GAS) programming models such as Unified Parallel C (UPC), Titanium, and Co-Array Fortran (CAF). This document presents a Global Address Space Performance (GASP) tool interface that is flexible enough to be adapted into current global address space compiler and runtime infrastructures with little effort, while allowing performance analysis tools to gather much information about the performance of global address space programs.

A.1.2 Organization

Section A.2 gives a high-level overview of the GASP interface. As GASP can be used to support many global address space programming models, the interface has been broken down into model-independent and model-specific sections. Section A.3 presents the model-independent portions of the GASP interface, and the subsequent sections detail the model-specific portions of the interface.

A.1.3 Definitions

In this section, we define the terms used throughout this specification.

- **Model** – a parallel programming language or library, such as UPC or MPI.
- **Users** – individuals using a GAS model such as UPC.
• **Developers** – individuals who write parallel software infrastructure such as UPC, CAF, or Titanium compilers.

• **Tools** – performance analysis tools such as Vampir, TAU, or KOJAK.

• **Tool developers** – individuals who develop performance analysis tools.

• **Tool code** – code or library implementing the tool developer’s portion of the GASP interface.

• **Thread** – a thread of control in a GAS program, maps directly to UPCs concept of threads or CAFs concept of images.

## A.2 GASP Overview

The GASP interface controls the interaction between a users code, a performance tool, and GAS model compiler and/or runtime system. This interaction is event-based and comes in the form of callbacks to the `gasp_event_notify` function at runtime. The callbacks may come from instrumentation code placed directly in an executable, from an instrumented runtime library, or any other method; the interface only requires that `gasp_event_notify` is called at appropriate times in the manner described in the rest of this document.

The GASP interface allows tool developers to support GAS models on all platforms and implementations supporting the interface. The interface is used in the following three steps:

1. Users compile their GAS code using compiler wrapper scripts provided by tool developers. Users may specify which analysis they wish the tool to perform on their code through either command-line arguments, environment variables or through other tool-specific methods.

2. The compiler wrapper scripts pass appropriate flags to the compiler indicating which callbacks the tool wishes to receive. During the linking phase, the scripts link in appropriate code from the performance tool that handles the callbacks at runtime. This tool-provided code shall be written in C.

3. When a user runs their program, the tool-provided code receives callbacks at runtime and may perform some action such as storing all events in a trace file or performing basic statistical profiling.
The specifics of each step will be discussed in Section A.3. The model-specific portions of the GASP interface will be discussed in the subsequent sections. A GAS implementation may exclude any system-level event defined in the model-specific sections of this document if an application cannot be instrumented for that event (e.g., due to design limitations or other implementation-specific constraints). Any action resulting in a violation of this specification shall result in undefined behavior. Tool and model implementors are strongly encouraged not to deviate from these specifications.

A.3 Model-Independent Interface

A.3.1 Instrumentation Control

Instrumentation control is accomplished through either compilation arguments or compiler pragmas. Developers may use alternative names for the command-line arguments if the names specified below do not fit the conventions already used by the compiler.

A.3.1.1 User-visible instrumentation control

If a user wishes to instrument their code for use with a tool using the GASP interface, they shall pass one of the command-line arguments described in this section to the compiler wrapper scripts. GASP system events are divided into the following broad categories, for the purposes of instrumentation control:

- **Local access events**: Events resulting from access to objects or variables contained in the portion of the global address space which is local to the accessing thread.

- **User function events**: Events resulting from entry and exit to user-defined functions, as described in Section A.4.3.

- **Other events**: Any system event which does not fall into the above categories.

The `--inst` argument specifies that the users code shall be instrumented for all system events supported by the GAS model implementation which fall into the final category of events described above. The `--inst-local` argument implies `--inst`, and additionally requests that user code shall be instrumented to generate local access events supported by the GAS model implementation. Otherwise, such events need not be
generated. For models lacking a semantic concept of local or remote memory accesses, `--inst` shall have the same semantics as `--inst-local`, implying instrumentation of all global address space accesses. The `--inst-functions` argument implies `--inst`, and additionally requests that user code shall be instrumented to generate user function events supported by the GAS model implementation. Otherwise, such events need not be generated.

### A.3.1.2 Tool-visible instrumentation control

Compilers supporting the GASP interface shall provide the following command-line arguments for use by the tool-provided compiler wrapper scripts. The arguments `--inst`, `--inst-local` and `--inst-functions` have the same semantics as the user-visible instrumentation flags specified in Section A.3.1.1. An additional argument `--inst-only` takes a single argument filename which is a file containing a list of symbolic event names (as defined in the model-specific sections of this document) separated by newlines. The files contents indicate the events for which the performance tool wishes to receive callbacks. Events in this file may be ignored by the compiler if the events are not supported by the model implementation. Compiler implementations are encouraged to avoid any overheads associated with generating events not specified by `--inst-only`, however tools that pass `--inst-only` must still be prepared to receive and ignore events which are not included in the `--inst-only` list.

### A.3.1.3 Interaction with instrumentation, measurement, and user events

When code is compiled without an `--inst` flag, all instrumentation control shall be ignored and all user event callbacks shall be compiled away. Systems may link “dummy” versions of `gasp_control` and `gasp_create_event` (described in Section A.3.3 and A.3.4) for applications that have no code compiled with `--inst`.

Systems may support compiling parts of an application using one of the `--inst` flags and compiling other parts of an application normally; for systems where this scenario is not possible, this behavior may be prohibited. Applications compiled using an `--inst`
flag for at least one translation unit shall also pass the \texttt{--inst} flag during the linking phase to the compiler wrapper scripts. Any model-specific instrumentation control shall not have any effect on user events or on the state of measurement control. As a result, any model-specific instrumentation controls shall not prevent user events from being instrumented during compilation (e.g., \texttt{#pragma pupc} shall not change the behavior of the \texttt{pupc_create_event} and \texttt{pupc_event_start} functions in UPC programs).

A.3.2 Callback Structure

At runtime, all threads of an instrumented executable shall collectively call the \texttt{gasp_init} C function at the beginning of program execution after the model runtime has finished initialization but before executing the entry point in a user’s code (e.g., \texttt{main} in UPC). The \texttt{gasp_init} function shall have the following signature:

\begin{verbatim}
typedef enum {
    GASP_LANG_UPC,
    GASP_LANG_TITANIUM,
    GASP_LANG_CAF,
    GASP_LANG_MPI,
    GASP_LANG_SHMEM
} gasp_model_t;

struct _gasp_context_S;
typedef struct _gasp_context_S *gasp_context_t;

gasp_context_t gasp_init(gasp_model_t srcmodel,
                        int *argc, char ***argv);
\end{verbatim}

The \texttt{gasp_init} function and an implementation of the \texttt{_gasp_context_S} struct shall be provided by tool developers. A single running instance of an executable may collectively call \texttt{gasp_init} multiple times if the executable contains code written in
multiple models (such as a hybrid UPC and CAF program), with at most one call per model. The `gasp_init` function returns a pointer to an opaque, thread-specific, tool-implemented struct. This pointer shall be passed in all subsequent calls to the tool developer’s code made on behalf of this thread. This pointer shall only be used in event callbacks for events corresponding to the model indicated by the `srcmodel` argument. Tool code may modify the contents of the `argc` and `argv` pointers to support the processing of command-line arguments.

After the `gasp_init` function has been called by each thread of execution, the tool code shall receive all other callbacks through the two functions whose signatures are shown below. Both functions may be used interchangeably; the `VA` variant is provided as a convenience to developers.

```c
typedef enum {
    GASP_START,
    GASP_END,
    GASP_ATOMIC,
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t context, unsigned int evttag,
    gasp_evttype_t evttype, const char *filename,
    int linenum, int colnum, ...);

void gasp_event_notifyVA(gasp_context_t context, unsigned int evttag,
    gasp_evttype_t evttype, const char *filename,
    int linenum, int colnum, va_list varargs);
```

The `gasp_event_notify` implementation shall be written in C, but may make upcalls to code written in the model specified by the `srcmodel` argument passed to the `gasp_init` function on the thread that received the callback. If upcalls are used, the
The `gasp_event_notify` function implementation is responsible for handling re-entrant calls. Additionally, code that is used in upcalls shall be compiled using the same environmental specifications as the code in a users application (e.g., `gasp_event_notify` shall only perform upcalls to UPC code compiled under a static threads environment when used with a UPC program compiled under the static threads environment).

Any user data referenced by pointers passed to `gasp_event_notify` shall not be modified by tool code. For the first argument to `gasp_event_notify`, tool code shall receive the same `gasp_context_t` pointer that was returned from the `gasp_init` function for this thread. Tool developers may use the context struct to store thread-local information for each thread. The `gasp_event_notify` function shall be thread-safe in order to support model implementations that make use of `pthreads` or other thread libraries.

The `evttag` argument shall specify the event identifier as described in the model-specific sections of this document. The `evtttype` argument shall be of type `gaspEvtType_t` and shall indicate whether the event `evttag` is a begin event, end event, or atomic event.

The `filename`, `linenum`, and `colnum` arguments shall indicate the line and column number in the model-level source code most closely associated with the generation of the event `evttag`. If filename is non-NULL, it references a character string whose contents must remain valid and unmodified for the remainder of the program execution. The same `filename` pointer is permitted to be passed in multiple calls and by multiple threads, and it is also permitted for different `filename` pointers (passed in different calls) to indicate the same file name (this scenario implies the tool may store `filename` pointer values and use simple pointer comparison of non-NULL values to establish filename equality, but not inequality).

GAS model implementations that do not retain column information during compilation may pass 0 in place of the `colnum` parameter. GAS model implementations
that do not retain any source-level information during compilation may pass 0 for the 
filename, lineno, and colnum parameters. GAS model implementations are strongly 
encouraged to support these arguments unless this information can be efficiently and 
accurately obtained through other documented methods. GAS model implementations 
that use instrumented runtime libraries for GASP support may provide dummy 
implementations for the gasp_event_notify, gasp_event_notifyVA, gasp_init functions 
and _gasp_context_S struct to prevent link errors while linking a user’s application that 
is not being used with any performance tool. The contents of the varargs argument shall 
be specific to each event identifier and type and will be discussed in the model-specific 
sections of this document.

A.3.3 Measurement Control

Tool developers shall provide an implementation for the following function:

```c
int gasp_control(gasp_context_t context, int on);
```

The gasp_control function takes the context argument in the same manner as 
the gasp_event_notify function. When the value 0 is passed for the on parameter, 
the tool shall cease measuring any performance data associated with subsequent system 
or user events generated on the calling thread, until the thread makes a future call to 
gasp_control with a nonzero value for the on parameter. The gasp_control function 
shall return the last value for the on parameter the function received from this thread, or a 
nonzero value if gasp_control has never been called for this thread.

A.3.4 User Events

Tool developers shall provide an implementation for the following function:

```c
unsigned int gasp_create_event(gasp_context_t context,
            const char *name, const char *desc);
```

The gasp_create_event shall return a tool-generated event identifier. Compilers 
shall translate the corresponding model-specific _create_event functions listed in the
model-specific sections of this document into corresponding `gasp_create_event` calls. The semantics of the `name` and `desc` arguments and the return value shall be the same as defined by the `_create_event` function listed in the model-specific section of this document corresponding to the model indicated by context.

### A.3.5 Header Files

Developers shall distribute a `gasp.h` C header file with their GAS implementations that contains at least the following definitions. The `gasp.h` file shall be installed in a directory that is included in the compiler's default search path.

- Function prototypes for the `gasp_init`, `gasp_event_notify`, `gasp_control`, and `gasp_create_event` functions and associated typedefs, enums, and structs.
- A `GASP_VERSION` macro that shall be defined to an integral date (coded as YYYYMMDD) corresponding to the GASP version supported by this GASP implementation. For implementations that support the version of GASP defined in this document, this macro shall be set to the integral value 20060914.
- Macro definitions that map the symbolic event names listed in the model-specific sections of this document to 32-bit unsigned integers.

### A.4 C Interface

#### A.4.1 Instrumentation Control

Instrumentation for the events defined in this section shall be controlled by using the corresponding instrumentation control mechanisms for UPC code defined in Section A.5.1.

#### A.4.2 Measurement Control

Measurement for the events defined in this section shall be controlled by using the corresponding measurement control mechanisms for UPC code defined in Section A.5.2.

#### A.4.3 System Events

##### A.4.3.1 Function events

Table A-1 shows system events related to executing user functions. These events occur upon each call to a user function (after entry into that function), and before exit from a user function (before returning to the caller as a result of executing a return statement or reaching the closing brace which terminates the function). The `funcsig`
Table A-1. User function events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_C_FUNC</td>
<td>Start, End</td>
<td>const char* funcsig</td>
</tr>
</tbody>
</table>

Table A-2. Memory allocation events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_C_MALLOC</td>
<td>Start</td>
<td>size_t nbytes</td>
</tr>
<tr>
<td>GASP_C_MALLOC</td>
<td>End</td>
<td>size_t nbytes, void* returnptr</td>
</tr>
<tr>
<td>GASP_C_REALLOC</td>
<td>Start</td>
<td>void* ptr, size_t size</td>
</tr>
<tr>
<td>GASP_C_REALLOC</td>
<td>End</td>
<td>void* ptr, size_t size, void* returnptr</td>
</tr>
<tr>
<td>GASP_C_FREE</td>
<td>Start, End</td>
<td>void* ptr</td>
</tr>
</tbody>
</table>

Argument specifies the character string representing the full signature of the user function that is being entered or exited, or NULL if that information is not available.

If funcsig is non-NULL, it references a character string whose contents must remain valid and unmodified for the remainder of the program execution. The same funcsig pointer is permitted to be passed in multiple calls and by multiple threads, and it is also permitted for different funcsig pointers (passed in different calls) to indicate the same function signature (this scenario implies the tool may store funcsig pointer values and use simple pointer comparison of non-NULL values to establish function equality, but not inequality).

A.4.3.2 Memory allocation events

Table A-2 shows system events related to the standard memory allocation functions. The GASP_C_MALLOC, GASP_C_REALLOC, and GASP_C_FREE stem directly from the standard C definitions of malloc, realloc, and free.

A.4.4 Header Files

Supported C system events shall be handled in the same method as UPC events, which are described in Section A.5.5.
A.5 UPC Interface

A.5.1 Instrumentation Control

Users may insert 
#pragma pupc on

or

#pragma pupc off
directives in their code to instruct the compiler to avoid instrumenting lexically-scoped regions of a users UPC code. These pragmas may be ignored by the compiler if the compiler cannot control instrumentation for arbitrary regions of code. When an --inst argument is given to a compiler or compiler wrapper script, the #pragma pupc shall default to on.

A.5.2 Measurement Control

At runtime, users may call the following functions to control the measurement of performance data. The pupc_control function shall behave in the same manner as the gasp_control function defined in Section A.3.3.

int pupc_control(int on);

A.5.3 User Events

unsigned int pupc_create_event(const char *name, const char *desc);

void pupc_event_start(unsigned int evttag, ...);

void pupc_event_end(unsigned int evttag, ...);

void pupc_event_atomic(unsigned int evttag, ...);

The pupc_create_event function shall be automatically translated into a corresponding gasp_create_event call, as defined in Section A.3.4. The name argument shall be used to associate a user-specified name with the event, and the desc argument may contain either NULL or a printf-style format string. The memory referenced by both arguments need not remain valid once the function returns.

The event identifier returned by pupc_create_event shall be a unique value in the range from GASP_UPC_USEREVT_START to GASP_UPC_USEREVT_END, inclusive. The GASP_UPC_USEREVT macros shall be provided in the gasp_upc.h header file described in
Section A.5.5. The value returned is thread-specific. If the unique identifiers are exhausted for the calling thread, `pupc_create_event` shall issue a fatal error.

The `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` functions may be called by a users UPC program at runtime. The `evttag` argument shall be any value returned by a prior `pupc_create_event` function call from the same thread. Users may pass in any list of values for the `...` arguments, provided the argument types match the `printf`-style format string supplied in the corresponding `pupc_create_event` (according to the `printf` format string conventions specified by the target system). Any memory referenced by `...` arguments (e.g., string arguments) need not remain valid once the function returns. A performance tool may use these values to display performance information alongside application-specific data captured during runtime to a user. The UPC implementation shall translate the `pupc_event_start`, `pupc_event_end`, and `pupc_event_atomic` function calls into corresponding `gasp_event_notify` function calls.

When a compiler does not receive any `--inst` arguments, the `pupc_event` function calls shall be excluded from the executable or linked against dummy implementations of these calls. A users program shall not depend on any side effects that occur from executing the `pupc_event` functions. Users shall not pass a shared-qualified pointer as an argument to the `pupc_event` functions.

A.5.4 System Events

For the event arguments below, the UPC-specific types `upc_flag_t` and `upc_op_t` shall be converted to C `ints`. Pointers to shared data shall be passed with an extra level of indirection, and may only be dereferenced through UPC upcalls. UPC implementations shall provide two opaque types, `gasp_upc_PTS_t` and `gasp_upc_lock_t`, which shall represent a generic pointer-to-shared (i.e., `shared void *`), and a UPC lock pointer (i.e., `upc_lock_t *`), respectively. These opaque types shall be typedefed to `void` to prevent C code from attempting to dereference them without using a cast in a UPC upcall. The content of any `gasp_upc_PTS_t` or `gasp_upc_lock_t` location passed to an event is only
Table A-3. Exit events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_COLLECTIVE_EXIT</td>
<td>Start, End</td>
<td>int status</td>
</tr>
<tr>
<td>GASP_UPC_NONCOLLECTIVE_EXIT</td>
<td>Atomic</td>
<td>int status</td>
</tr>
</tbody>
</table>

Table A-4. Synchronization events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_NOTIFY</td>
<td>Start, End</td>
<td>int named, int expr</td>
</tr>
<tr>
<td>GASP_UPC_WAIT</td>
<td>Start, End</td>
<td>int named, int expr</td>
</tr>
<tr>
<td>GASP_UPC_BARRIER</td>
<td>Start, End</td>
<td>int named, int expr</td>
</tr>
<tr>
<td>GASP_UPC_FENCE</td>
<td>Start, End</td>
<td>(none)</td>
</tr>
</tbody>
</table>

guaranteed to remain valid for the duration of the `gasp_event_notify` call, and must not be modified by the tool.

A.5.4.1 Exit events

Table A-3 shows system events related to the end of a program’s execution. The `GASP_UPC_COLLECTIVE_EXIT` events shall occur at the end of a program’s execution on each thread when a collective exit occurs. These events correspond to the execution of the final implicit barrier for UPC programs. The `GASP_UPC_NONCOLLECTIVE_EXIT` event shall occur at the end of a program’s execution on a single thread when a non-collective exit occurs.

A.5.4.2 Synchronization events

Table A-4 shows events related to synchronization constructs. These events shall occur before and after execution of the notify, wait, barrier, and fence synchronization statements. The `named` argument to the notify, wait, and barrier start events shall be nonzero if the user has provided an integer expression for the corresponding notify, wait, and barrier statements. In this case, the `expr` variable shall be set to the result of evaluating that integer expression. If the user has not provided an integer expression for the corresponding notify, wait, or barrier statements, the `named` argument shall be zero and the value of `expr` shall be undefined.
Table A-5. Work-sharing events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_FORALL</td>
<td>Start, End</td>
<td>(none)</td>
</tr>
</tbody>
</table>

A.5.4.3 Work-sharing events

Table A-5 shows events related to work-sharing constructs. These events shall occur on each thread before and after upc_forall constructs are executed.

A.5.4.4 Library-related events

Table A-6 shows events related to library functions. These events stem directly from the UPC library functions defined in the UPC specification. The vararg arguments for each event callback mirror those defined in the UPC language specification.

A.5.4.5 Blocking shared variable access events

Table A-7 shows events related to blocking shared variable accesses. These events shall occur whenever shared variables are assigned to or read from using the direct syntax (not using the upc.h library functions). The arguments to these events mimic those of the upc_memget and upc_memput event callback arguments, but differ from the ones presented in the previous section because they only arise from accessing shared variables directly. If the memory access occurs under the relaxed memory model, the is_relaxed parameter shall be nonzero; otherwise the is_relaxed parameter shall be zero.

A.5.4.6 Non-blocking shared variable access events

Table A-8 shows events related to direct shared variable accesses implemented through non-blocking communication. These non-blocking direct shared variable access events are similar to the regular direct shared variable access events in Section A.5.4.5. The INIT events shall correspond to the non-blocking communication initiation, the DATA events shall correspond to when the data starts to arrive and completely arrives on the destination node (these events may be excluded for most implementations that use hardware-supported DMA), and the GASP_UPC_NB_SYNC function shall correspond to
<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_GLOBAL_ALLOC</td>
<td>Start</td>
<td>size_t nblocks, size_t nbytes</td>
</tr>
<tr>
<td>GASP_UPC_ALL_ALLOC</td>
<td>End</td>
<td>size_t nblocks, size_t nbytes, gasp_upc.PTS_t* newshrd_ptr</td>
</tr>
<tr>
<td>GASP_UPC_ALLOC</td>
<td>Start</td>
<td>size_t nbytes</td>
</tr>
<tr>
<td>GASP_UPC_ALLOC</td>
<td>End</td>
<td>size_t nbytes, gasp_upc.PTS_t* newshrd_ptr</td>
</tr>
<tr>
<td>GASP_UPC_FREE</td>
<td>Start, End</td>
<td>gasp_upc.PTS_t* shrd_ptr</td>
</tr>
<tr>
<td>GASP_UPC_GLOBAL_LOCK_ALLOC</td>
<td>Start</td>
<td>(none)</td>
</tr>
<tr>
<td>GASP_UPC_GLOBAL_LOCK_ALLOC</td>
<td>End</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_ALL_LOCK_ALLOC</td>
<td>Start</td>
<td>(none)</td>
</tr>
<tr>
<td>GASP_UPC_ALL_LOCK_ALLOC</td>
<td>End</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_LOCK_FREE</td>
<td>Start, End</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_LOCK</td>
<td>Start, End</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_LOCK_ATTEMPT</td>
<td>Start</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_LOCK_ATTEMPT</td>
<td>End</td>
<td>gasp_upc.lock_t* lck, int_result</td>
</tr>
<tr>
<td>GASP_UPC_UNLOCK</td>
<td>Start, End</td>
<td>gasp_upc.lock_t* lck</td>
</tr>
<tr>
<td>GASP_UPC_MEMCPY</td>
<td>Start, End</td>
<td>gasp_upc.PTS_t* dst, gasp_upc.PTS_t* src, size_t n</td>
</tr>
<tr>
<td>GASP_UPC_MEMGET</td>
<td>Start, End</td>
<td>void* dst, gasp_upc.PTS_t* src, size_t n</td>
</tr>
<tr>
<td>GASP_UPC_MEMPUT</td>
<td>Start, End</td>
<td>gasp_upc.PTS_t* dst, void* src, size_t n</td>
</tr>
<tr>
<td>GASP_UPC_MEMSET</td>
<td>Start, End</td>
<td>gasp_upc.PTS_t* dst, int c, size_t n</td>
</tr>
</tbody>
</table>
the final synchronization call that blocks until the corresponding data of the non-blocking operation is no longer in flight.

\texttt{gasp\_upc\_nb\_handle\_t} shall be an opaque type defined by the UPC implementation.

Several outstanding non-blocking 	extit{get} or 	extit{put} operations may be attached to a single

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Symbolic name & Event type & vararg arguments \\
\hline
\texttt{GASP\_UPC\_GET} & Start, End & \\texttt{int is\_relaxed, void\_* dst, gasp\_upc\_PTS\_t\_* src, size\_t n} \\
\hline
\texttt{GASP\_UPC\_PUT} & Start, End & \\texttt{int is\_relaxed, gasp\_upc\_PTS\_t\_* dst, void\_* src, size\_t n} \\
\hline
\end{tabular}
\caption{Blocking shared variable access events}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Symbolic name & Event type & vararg arguments \\
\hline
\texttt{GASP\_UPC\_NB\_GET\_INIT} & Start & \\texttt{int is\_relaxed, void\_* dst, gasp\_upc\_PTS\_t\_* src, size\_t n} \\
\hline
\texttt{GASP\_UPC\_NB\_GET\_INIT} & End & \\texttt{int is\_relaxed, void\_* dst, gasp\_upc\_PTS\_t\_* src, size\_t n, gasp\_upc\_nb\_handle\_t handle} \\
\hline
\texttt{GASP\_UPC\_NB\_GET\_DATA} & Start, End & \texttt{gasp\_upc\_nb\_handle\_t handle} \\
\hline
\texttt{GASP\_UPC\_NB\_PUT\_INIT} & Start & \\texttt{int is\_relaxed, gasp\_upc\_PTS\_t\_* dst, void\_* src, size\_t n} \\
\hline
\texttt{GASP\_UPC\_NB\_PUT\_INIT} & End & \\texttt{int is\_relaxed, gasp\_upc\_PTS\_t\_* dst, void\_* src, size\_t n, gasp\_upc\_nb\_handle\_t handle} \\
\hline
\texttt{GASP\_UPC\_NB\_PUT\_DATA} & Start, End & \texttt{gasp\_upc\_nb\_handle\_t handle} \\
\texttt{GASP\_UPC\_NB\_SYNC} & Start, End & \texttt{gasp\_upc\_nb\_handle\_t handle} \\
\hline
\end{tabular}
\caption{Non-blocking shared variable access events}
\end{table}
Table A-9. Shared variable cache events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_CACHE_MISS</td>
<td>Atomic</td>
<td>size_t n, size_t n_lines</td>
</tr>
<tr>
<td>GASP_UPC_CACHE_HIT</td>
<td>Atomic</td>
<td>size_t n</td>
</tr>
<tr>
<td>GASP_UPC_CACHE_INVALIDATE</td>
<td>Atomic</td>
<td>size_t n_dirty</td>
</tr>
</tbody>
</table>

gasp_upc_nb_handle_t instance. When a sync callback is received, the tool code shall assume all get and put operations for the corresponding handle in the sync callback have been retired. The implementation may pass the handle GASP_NB_TRIVIAL to GASP_UPC_NB_{PUT,GET}_INIT to indicate the operation was completed synchronously in the initiation interval. The tool should ignore any DATA or SYNC event callbacks with the handle GASP_NB_TRIVIAL.

A.5.4.7 Shared variable cache events

Table A-9 shows events related to shared variable cache events. The GASP_UPC_CACHE events may be sent for UPC runtime systems containing a software cache after a corresponding get or put start event but before a corresponding get or put end event (including non-blocking communication events). UPC runtimes using write-through cache systems may send GASP_UPC_CACHE_MISS events for each corresponding put event.

The size_t n argument for the MISS and HIT events shall indicate the amount of data read from the cache line for the particular cache hit or cache miss. The n_lines argument of the GASP_UPC_CACHE_MISS event shall indicate the number of bytes brought into the cache as a result of the miss (in most cases, the line size of the cache). The n_dirty argument of the GASP_UPC_CACHE_INVALIDATE shall indicate the number of dirty cache lines that were written back to shared memory due to a cache line invalidation.

A.5.4.8 Collective communication events

Table A-10 shows events related to collective communication. The events in Table A-10 stem directly from the UPC collective library functions defined in the UPC
Table A-10. Collective communication events

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_ALL_BROADCAST</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_SCATTER</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_GATHER</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_GATHER_ALL</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_EXCHANGE</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_PERMUTE</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, gasp_upc_PTS_t* perm, size_t nbytes, int upc_flags</code></td>
</tr>
<tr>
<td>GASP_UPC_ALL_REDUCE</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, int upc_op, size_t nelems, size_t blk_size, void* func, int upc_flags, gasp_upc_reduction_t type</code></td>
</tr>
</tbody>
</table>

specification. The vararg arguments for each event callback mirror those defined in the UPC language specification.
Table A-10. Collective communication events (Continued)

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Event type</th>
<th>vararg arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_ALL_PREFIX_REDUCE</td>
<td>Start, End</td>
<td><code>gasp_upc_PTS_t* dst, gasp_upc_PTS_t* src, int upc_op, size_t nelems, size_t blk_size, void* func, int upc_flags, gasp_upc_reduction_t type</code></td>
</tr>
</tbody>
</table>

For the reduction functions, the `gasp_upc_reduction_t` enum shall be provided by a UPC implementation and shall be defined as follows. The suffix to GASP_UPC_REDUCTION denotes the same type as specified in the UPC specification.

typedef enum {
    GASP_UPC_REDUCTION_C,
    GASP_UPC_REDUCTION_UC,
    GASP_UPC_REDUCTION_S,
    GASP_UPC_REDUCTION_US,
    GASP_UPC_REDUCTION_I,
    GASP_UPC_REDUCTION_UI,
    GASP_UPC_REDUCTION_L,
    GASP_UPC_REDUCTION_UL,
    GASP_UPC_REDUCTION_F,
    GASP_UPC_REDUCTION_D,
    GASP_UPC_REDUCTION_LD
} gasp_upc_reduction_t;

A.5.5 Header Files

UPC compilers shall distribute a `pupc.h` C header file with their GAS language implementations that contains function prototypes for the functions defined in Sections
A.5.2 and A.5.3. The pupc.h file shall be installed in a directory that is included in the UPC compiler’s default search path.

All supported system events and associated gasp_upc_* types shall be defined in a gasp_upc.h file located in the same directory as the gasp.h file. System events not supported by an implementation shall not be included in the gasp_upc.h file. The gasp_upc.h header file may include definitions for implementation-specific events, along with brief documentation embedded in source code comments.

Compilers shall define a compiler-specific integral GASP_UPC_VERSION version number in gasp_upc.h that may be incremented when new implementation-specific events are added. Compiler developers are encouraged to use the GASP_X_Y naming convention for all implementation-specific events, where X is an abbreviation for their compilation system (such as BUPC) and Y is a short, descriptive name for each event.

Compilers that implement the pupc interface shall predefine the feature macro __UPC_PUPC__ to the value 1. The macro should be predefined whenever applications may safely #include <pupc.h>, invoke the functions it defines and use the #pragma pupc directives, without causing any translation errors. The feature macro does not guarantee that GASP instrumentation is actually enabled for a given compilation, as some of the features might have no effect in non-instrumenting compilations.
REFERENCES


[8] Intel Corporation, “Intel cluster tools website,”


BIOGRAPHICAL SKETCH

Hung-Hsun Su is a Ph.D. graduate from the Department of Electrical and Computer Engineering (with a minor in computer engineering from the Department of Computer and Information Science and Engineering) at the University of Florida. He received two B.S. degrees in computer science and biochemistry from the University of California, Los Angeles in 1996, a M.S. in biochemistry from the University of Southern California in 1999, and a M.S. in computer science from the University of Southern California in 2002. His research focuses on the development and performance analysis of high-performance parallel applications; the realization of high-performance portable communication systems; and performance evaluation of high-performance systems.