1-1-2011

Low-Impact Profiling of Streaming, Heterogeneous Applications

Joseph Lancaster
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Roger Chamberlain, chair
Jeremy Buhler
Patrick Crowley
Ron Cytron
Chris Gill
Ron Indeck
Henric Krawczynski

LOW-IMPACT PROFILING OF STREAMING, HETEROGENEOUS APPLICATIONS

by

Joseph Marion Lancaster

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2011
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Low-Impact Profiling of Streaming,

Heterogeneous Applications

by

Joseph Marion Lancaster

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2011

Research Advisor: Roger D. Chamberlain

Computer engineers are continually faced with the task of translating improvements in fabrication process technology (i.e., Moore's Law) into architectures that allow computer scientists to accelerate application performance. As feature-size continues to shrink, architects of commodity processors are designing increasingly more cores on a chip. While additional cores can operate independently with some tasks (e.g. the OS and user tasks), many applications see little to no improvement from adding more processor cores alone.

For many applications, heterogeneous systems offer a path toward higher performance. Significant performance and power gains have been realized by combining specialized processors (e.g., Field-Programmable Gate Arrays, Graphics Processing Units) with general purpose multi-core processors. Heterogeneous applications need to be programmed differently than traditional software. One approach, stream processing, fits these systems particularly well because of the segmented memories and explicit expression of parallelism. Unfortunately, debugging and performance tools that support streaming, heterogeneous applications do not exist.

This dissertation presents *TimeTrial*, a performance measurement system that enables performance optimization of streaming applications by profiling the application deployed on a heterogeneous system. TimeTrial performs low-impact measurements by dedicating computing resources to monitoring and by aggressively compressing performance traces into statistical summaries guided by user specification of the performance queries of interest.

Acknowledgments

I would like to extend my deep appreciation to my adviser, Dr. Roger Chamblerlain for his guidance throughout my graduate career. Without his encouragement I doubt I would have pursued a doctoral degree. The many hours of discussions regarding research directions, feedback on manuscripts, and career advice truly helped contribute to my successfully completing this doctoral degree.

I would like to thank Dr. Jeremy Buhler for acting as a secondary adviser, always keeping the engineer in me in check with a healthy dose of theoretical approaches. Dr. Buhler helped me to expand my research skills and dive into problems that I might have been reluctant to otherwise explore. His patience and support throughout my graduate career will always be appreciated.

I am extremely grateful for financial support provided by NIH award R42 HG003225 and NSF award CNS-0931693. This support allowed me to concentrate exclusively on my research.

I would like to thank the members of my dissertation committee for their insightful questions that helped improve and focus my research.

I would like to thank all of my graduate colleagues, especially the members of the High Performance Computational Biology group. It has been a pleasure to learn and grow with all of you. I want to mention one colleague in particular, Arpith Jacob, with whom I worked closely with for my entire graduate career. It has been quite the journey.

Finally, I want to thank the eternal patience of my wife, Sarah Katherine Lancaster, who supported me through the most challenging times as a graduate student.

Joseph Marion Lancaster

Washington University in Saint Louis
August 2011

To my lovely wife, Sarah Katherine Lancaster.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, we have seen the emergence of a number of specialized computing technologies available for use in commodity platforms. Single-processor systems have yielded the mainstream to homogeneous multi-core chips, and commodity computers with 64-cores (using 4 sockets) are scheduled to be released soon. Graphics processing units (GPUs) from both major manufacturers, Nvidia and AMD, now have reasonably mature language and runtime support for general purpose computation, opening up vast amounts of computational resources to a programmer. Field-programmable gate arrays (FPGAs) have continued to increase in capability and are particularly useful for some applications. Important problems have been accelerated by incorporating specialized architectures with multi-core processors. Recently described examples include computational biology [32, 50], computational chemistry [116], and high-performance signal processing [110].

Computer systems can be constructed from a variety of computational resources, such as the ones above, to form systems that have strengths in different application domains. A system builder can then tailor the mix of technologies to best meet the needs of the application domains of interest. These platforms enable a developer to exploit the strengths of each individual architecture to create higher-performing applications when compared to computer systems that use multi-core processors alone. We refer to computer systems constructed using a variety of the above technologies as "heterogeneous computing systems" or "heterogeneous systems" for short. Similarly, applications that are deployed on more than one technology are dubbed "heterogeneous applications."

Each of the above technologies has its own architecture, memory subsystem, language for authoring applications, performance capabilities and limitations, algorithmic strengths and weaknesses, and communities of proponents and detractors. The potential for higher performance in a heterogeneous application can be challenging to realize. An interesting open problem is how to effectively harness the power of such systems. Researchers currently rely

on ad hoc methods to build applications on such systems, and these methods lack many staples of traditional computing platforms such as robust debugging and performance analysis tools. The combination of increased complexity from using multiple programming paradigms and the lack of tools that support heterogeneous applications deployed onto them leads to unacceptably long design cycles.

The success of future designs for large, complex heterogeneous systems is contingent on improvements in developer tools. Tools that effectively measure the performance of an application as it is running are essential for creating high-performance applications, and tools that substantially degrade the performance of the applications they are monitoring are of marginal benefit. Low-impact performance monitoring is needed.

The domain of this dissertation is performance analysis of streaming heterogeneous applications. The techniques developed are embodied in our performance measurement system, *TimeTrial*, which enables an application developer to better understand the performance of streaming heterogeneous applications. Providing relevant performance feedback was a primary goal in the design of TimeTrial so minimizing the impact on the measured application was emphasized. TimeTrial constructs performance profiles by measuring the performance of salient aspects of the application during runtime, summarizing the measurements into statistics periodically during execution, and presenting the results to the developer in a concise manner that directly correspond to metrics of interest.

1.1 Utility of Heterogeneous Computing Systems

Heterogeneous systems have been successful in accelerating applications in a wide variety of computing domains. As the name suggests, each heterogeneous system may be constructed differently, choosing from a mix of accelerators such as traditional multi-core processors, graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and custom heterogeneous integrated circuits (ASICs). FPGAs are a popular component in many heterogeneous systems due to the high degree of parallelism available and flexibility to reconfigure the architecture for each application.

With the stalling of clock rate increases for processor cores, the high performance computing community has relied more and more on parallelism to achieve performance increases at the application level. This has recently expanded to the investigation of application accelerators, or non-traditional computing components, as part of the solution to the need for performance. Heterogeneous systems have received significant attention by the research

community. The promise of reconfigurable logic is described in [36]. The use of graphics engines is shown in [18], and both technologies are used together in [23] and [109].

Figure 1.1 shows an example of a heterogeneous system similar to the prototyping platform used for the work in this dissertation. It is constructed using multi-core AMD Opterons, an off-the-shelf graphics card connected via a PCIe bus, and an dual-chip FPGA card connected via a PCI-X bus.

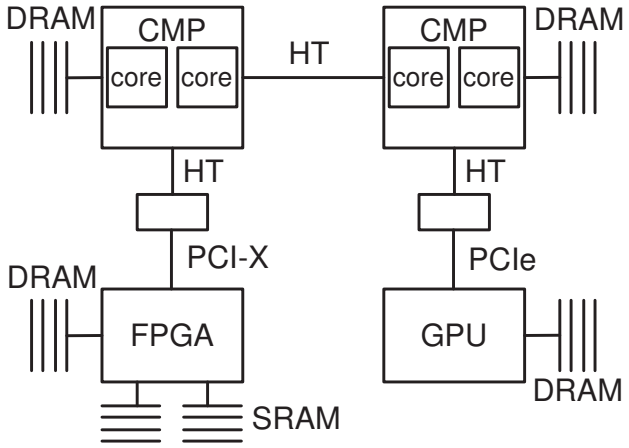


Figure 1.1: Example of a heterogeneous computing system.

While Figure 1.1 illustrates a multiple chip design, system-on-chip designs share many similarities, especially when considering the FPGA portion of the design. For instance, many SoC designs utilize FPGAs to prototype circuits which will eventually become an ASIC. Our techniques can be employed as part of the prototype implementation for a system-on-chip design as well, enabling performance characterization on large data sets.

1.2 Programming Heterogeneous Computing Systems

Multi-core processors alone enable a coarse-grain approach to improving application performance and the number of cores on a die is expected to increase proportionally with Moore's Law. While this trend helps with some tasks (e.g., the OS and user tasks can operate more independently), many applications see little to no improvement from this architectural trend since they were written with very little thread-level parallelism. It is likely that future applications will be designed to take advantage of multiple cores, however, it is an open research question as to whether a thread-based programming model is the best way to program parallelism.

Effectively utilizing heterogeneous systems to create a high-performance application is a challenging task. Application designers must contend with multiple programming styles, non-standard communication between resources, a high-dimensional design space, and a lack of developer tools. These challenges significantly increase the complexity inherent in the design of an heterogeneous system relative to that of a traditional multi-core platform. As heterogeneous system designs become ever larger and more complex, the challenges become ever more acute.

In order to ease the programmers' burden, both the research community and industry have been focusing on the design of *concurrency platforms*. A concurrency platform is an abstraction layer that coordinates, schedules and manages resources, and provides an interface for programmers to write parallel programs. A concurrency platform typically supports one or more parallel programming paradigms and may consist of a compiler, a runtime system, support tools (e.g., performance monitor), etc.

The streaming data computing paradigm, sometimes referred to as course-grained dataflow computing, has been touted as a clear improvement over traditional thread-based concurrency platforms and its associated locks, mutual exclusion, and race conditions [69]. While shared-memory programming is available for heterogeneous systems (e.g., see [4]) and message-passing has been implemented for some systems (e.g., see [3]), our focus is on stream computing in this work.

An important class of applications that can exploit the capabilities of heterogeneous systems can be expressed using a stream processing paradigm. Applications expressed in streaming semantics fit these systems particularly well because of the segmented memories and explicit expression of communication and both wide and deep parallelism. Generally, streaming applications can be thought of as coarse-grained dataflow computations in which computation blocks, or kernels, are interconnected by arcs over which data is communicated.

An example application topology is illustrated in Figure 1.2. The output data stream from block A is delivered as an input stream to block B, etc. Inside each block, data is consumed from the input arc(s), some computation is performed, and the results are sent along the outgoing arc(s).

In this paradigm, data to be processed are streamed into the computer system (typically from a disk, network or sensor), a variety of pipelined and parallel computations are performed on the data, and the results are streamed out of the system. Applications expressed in streaming semantics fit heterogeneous systems particularly well because of the segmented memories and explicit expression of both wide and deep parallelism. Wide parallelism refers

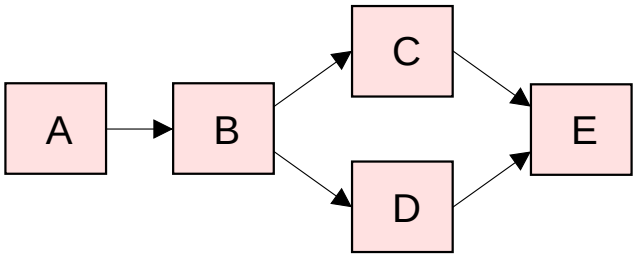


Figure 1.2: Sample streaming application.

to data-parallel tasks, similar to the MIMD model of computation. Deep parallelism refers to pipelining, either coarse or fine-grained. Examples of application domains that are easily mapped to a streaming model are sensor-based signal processing, audio and video processing, and many data-intensive scientific applications.

In most streaming languages (e.g., Brook [18], StreamIt [106]), the computation within the blocks and their interconnections are all expressed in a common language. In the Auto-Pipe environment [40], block computations are expressed in the native language of the computational resource(s) onto which the block is allocated. Examples of concurrency platforms that support the streaming data paradigm include Auto-Pipe [23], Brook [18], StreamIt [106], Wavescript [84], and Streamware [47] (see [101] for a survey of older streaming languages).

A more concrete streaming application is illustrated in Figure 1.3. This application performs a Monte Carlo simulation that computes a numerical solution to Laplace's equation. In this application, pseudo-random numbers are are generated in the `RNG` block. The `Split` block divides the random number stream into two data streams. The two streams are fed to the two `Walk` blocks that each perform the Monte Carlo based solution to Laplace's equation. The results of these walks are then passed downstream to the `Avg` block which averages the results. Finally, the `Print` block displays the results of the simulation.

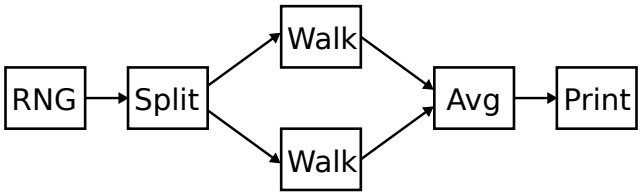


Figure 1.3: An example streaming implementation of a Monte Carlo solution to Laplace's equation. Computation is performed within each block, communication is one-way along edges.

Figure 1.4 shows an example of a signal-processing streaming application with three major stages. In stage 1, a sensor (e.g., a gamma ray telescope [117]) produces data that is split up

and sent to parallel processing pipelines in stage 2. Each pipeline performs some functions on the data, which is merged in stage 3. Finally, the results are all compared and stored to disk. Tyson et al. [110] describe the implementation of this application using FPGAs to execute the computationally expensive stage 2 and multi-core processors for stages 1 and 3.

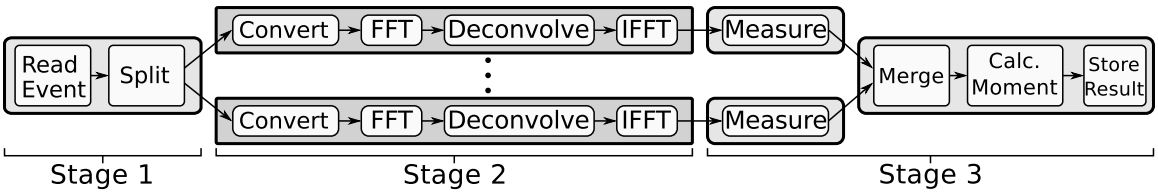


Figure 1.4: An example streaming application from computational astrophysics [110]. Computation is performed within each kernel, communication is one-way along edges.

There are several benefits to authoring applications using this approach [24]: (1) it is possible to build a library of blocks that can be re-used; (2) the concurrency platform provides for the data movement and associated synchronization; (3) the explicit knowledge of algorithm decomposition available to the system supports flexible mapping of blocks to compute resources; and (4) reasoning about the correctness of streaming data applications is fairly straightforward (approximately on a par with sequential codes).

We have experienced success with a particular streaming environment, Auto-Pipe, and language, X, that is tailored explicitly to these platforms [23, 25, 40]. Using X and Auto-Pipe allows one to describe both wide and deep parallelism by coordinating dataflow at the system level, while taking advantage of the efficiency that can be gained by authoring segments of applications in the component's native language.

1.3 Difficulties in Profiling Heterogeneous Applications

Authoring applications for heterogeneous systems is difficult for a number of reasons. First, the various co-processors have their own unique languages (e.g., Verilog or VHDL for FPGAs, CUDA or OpenCL for GPUs) and development environments. Second, significant developer effort is typically required to ensure that an algorithm deployed on a co-processor actually does perform well. Third, the application must be decomposed into components that execute across the heterogeneous platform. This requires a great deal of attention to address both correctness and performance issues. Finally, the tools available to developers are very limited at present. Speaking directly to the issue of FPGA co-processors, Underwood et al. [111] list 12 essential elements for acceptance in the high-performance computing

community. Developer tools of various forms (specifically including performance analysis tools) comprise two of their twelve elements.

Even basic performance information is challenging to obtain for heterogeneous applications with current tools. Without a profiler for the FPGA, a question like "Which block is the throughput limiter?" may be difficult to answer. Many designers result to manual instrumentation using ad hoc systems to get an answer to this specific question. For streaming applications that just use multi-core processors, answering the above question about a highly-concurrent application is also not straightforward to answer with current tools.

Obtaining peak performance from a heterogeneous system is essential to obtain an adequate return on the increased effort needed to design such a system. Unfortunately, because non-traditional architectures, particularly FPGAs, often offer limited visibility into the executing application, designers are often left without even the most basic performance information about these components of their designs. Traditional software tools such as `gprof` [45], `Valgrind` [83], and TAU [98] tend to be of marginal benefit because they are processor-centric and do not support meaningful metrics for architecturally diverse platforms (e.g., CPU $\rightarrow$ FPGA communication). FPGAs do not offer native support for performance assessment other than through simulation, which is too slow to be helpful for complex designs that process lots of data. Functional debugging tools like Xilinx's ChipScope, Altera's SignalTap, and Synopsys's Identify can support limited performance logging. However, the hardware designer must manually design in the evaluation logic for each performance metric. Adding such logic is frequently an afterthought at best, resulting in minimal performance visibility and brittle, error-prone solutions.

1.4 Research Questions

We address the following specific research questions in this dissertation:

1. How can the performance of a distributed, streaming application be measured? What aspects of the application should be measured?
2. How can the performance of a streaming application that is distributed across FPGA accelerators and processor cores be measured?
3. What data needs to be collected in order to answer the above questions? Can this data be collected in a low-impact manner and still satisfactorily answer those questions? What techniques should be used?

4. How does one build a profile of resources that have little inherent visibility (e.g. system I/O buses)?

5. How does one use the collected performance data to calibrate or validate performance models?

We will return to these questions Chapter 8.

1.5 Summary of our Approach

Performance measurement is the act of tracking the performance of a computer system or application during execution or simulation of one or more programs. A profile is a summary of those measurements which is typically presented to a developer for use in tuning the performance of an application.

A profile of a streaming data application is necessarily different than a profile of a sequential program (e.g., no universal program counter). Like profiles of parallel MPI programs, streaming application profiles should indicate the performance-limiting sections of the application in terms of both communication and computation (i.e., what is the performance bottleneck). For example, referring back to Figure 1.3, a stream profile should include answers to questions of the following type:

- At what rate is data moving across the link that connects the `RNG` block to the `Split` block?
- What is the occupancy of the queues between each block?
- What portion of the pipeline is limiting the throughput bottleneck?

We provide a broad set of measurements and enable the developer to ask performance questions directly. A simple, domain-specific language has been developed to provide stream profiles tailored to specific developer queries of application performance.

We present a method of performance monitoring that enables precise performance characterization and the flexibility to trade-off the frequency of reporting for higher precision performance metrics. These methods are implemented in a new performance measurement tool, *TimeTrial*, which enables wholesale collection of performance profiles of FPGA-accelerated

streaming data applications. TimeTrial is a runtime performance monitor that supports whole-application monitoring on heterogeneous systems comprised of processors and FPGAs while aggressively minimizing the impact that it has on the executing application.

With any runtime performance measurement system, there is a risk of interfering with the application being measured. To mitigate this potential interference, TimeTrial monitors applications by aggregating data online, supporting selective developer-directed profiles, and dedicating computational resources to the runtime monitoring tasks. These practices enable performance measurements over long executions and real-world data sets. In contrast, simulation permits observation of execution on only small data sets, increasing the likelihood that events of importance to performance will be missed or under-sampled.

This dissertation presents TimeTrial as incorporated with the Auto-Pipe [23] concurrency platform. TimeTrial fills a gap in available tools for performance monitoring, enabling users to profile their streaming data applications on a real heterogeneous platform.

1.6 Contributions

The following is a detailed list of my contributions:

- Developed techniques to measure heterogeneous streaming applications and designed and implemented these techniques into a measurement system named *TimeTrial*.
  - Enabled measurement of the performance of heterogeneous streaming applications, observing 100% of the execution time (not sampling) for both software and FPGA portions of an application.
  - Enabled wholesale communication profiling of complex streaming applications, measuring the performance independent of the implementation language of a kernel or block.
  - Designed and implemented a monitoring agent for an FPGA, explicitly focusing on low resource utilization, high clock frequency and minimal interference with the application being measured.
  - Designed and implemented a software performance monitoring agent which monitors the software portions of the application and logs FPGA performance results.
  - Designed a extensible domain-specific measurement specification language which allows developers to articulate a wide variety of measurements and aggregations.

Developer specification of what to measure and how to aggregate enables TimeTrial to discard the traces while still retaining the desired measurement.

  - Designed several lossy compression techniques that still capture important performance events while reducing communication overhead by several orders of magnitude.
  - Designed techniques to monitor and provide performance profiles of communication paths that cross the CPU to FPGA or FPGA to CPU boundary, referred to here as "virtual queues".

- Evaluated the effects of the measurement process on the performance of the application being measured.
- Integrated TimeTrial into the Auto-Pipe system to enable automated compiler-based instrumentation of heterogeneous applications deployed on traditional processors and FPGAs.
- We developed techniques and demonstrated these techniques to calibrate and validate performance models.
- We used TimeTrial to measure and tune the performance of several streaming applications.
  - We profiled the performance of a parallel implementation of a Laplace equation solver.
  - We debugged the performance of several iterations of a heterogeneous application, Mercury BLASTN, using TimeTrial.
  - We used TimeTrial to measure the overhead of deadlock avoidance algorithms implemented in Mercury BLASTN.

1.7 Outline

We now introduce the organization of the rest of the dissertation. Chapter 2 gives an overview of the information necessary to understand the motivation and direction of the work followed by descriptions of research relevant to the work in this dissertation. Chapter 3 describes the approach taken to enable performance measurements of streaming applications and techniques used to enable an optimized implementation. Chapter 4 describes the overall architecture of TimeTrial and how it was implemented and integrated into the

Auto-Pipe system. Chapter 5 details our strategy to provide visibility into opaque system buses, followed by the evaluation of this strategy for a PCI-X bus. Chapter 6 explores the use of TimeTrial to measure real streaming applications and presents the results and impact of those measurements. Chapter 7 describes our approach to using TimeTrial to support performance modeling. Finally, Chapter 8 summarizes the results presented in this dissertation and describes potential extensions of this work.

Chapter 2

Background and Related Work

In this chapter we introduce the background material that is useful for understanding the techniques and terminology employed in this dissertation. This is followed by a description of related work in the field.

2.1 Profiling Application Performance

Application profiling is the act of determining the aspects of an application that negatively impact the execution time or limit the throughput. Profiling has long been utilized to identify "hot" regions of a software application so that developers can appropriately focus their efforts on optimizing them. Profiling is critical to the understanding of many complex sequential applications, and is even more important when concurrent processing is used.

The term "profiler" is general and can refer to a variety of techniques to collect performance data. One can classify profilers into tools that simulate the instruction set to varying degrees of fidelity or those that monitor execution on deployed hardware. Instruction set simulation offers the most potential for detail and accuracy by trading off execution speed. Profiling through simulation is popular with computer architects to determine the benefits or drawbacks of particular architectural features on a set of benchmark applications. Unfortunately, it is not uncommon for instruction set simulations to be more than $10,000\times$ slower than native execution. A more common case for application development is the use of a runtime profiler to collect data by using hardware performance counters provided on a particular platform.

One challenge when building a runtime profiling system lies in gathering raw data, another with efficiently storing or summarizing this data while the program is executing. A runtime profiler that provides a trace of execution must deal with large volumes of performance

meta-data that needs to be stored with minimal impact. The resulting trace can then be used to drive a simulation. Typically these traces are compressed out of necessity due to the size of reasonable datasets. An ideal trace compression technique is lossless (i.e. does not lose information in the compression process), has a high compression ratio, and is easily decompressed for simulation [78]. The particular compression technique used depends on the type of information that is collected. There is a large body of trace compression techniques covered in the literature and a concise summary is presented in [78].

Runtime profilers may add instrumentation to the binary to count the execution frequencies of subroutines, loops, or even blocks of code. These counts provide a statistical summary (i.e. "lossy compression") of the application performance. The profiling techniques employed in TimeTrial fall in to this category. In addition, sampling may be used to lower the overhead of monitoring and the resulting impact on the application. For sequential code, edge profiling is a useful way to show a graph of the frequency of subroutine calls within a program. These techniques are employed in e.g. *gprof* [45]. Edge profiling can identify which subroutines are most frequently called but does not show the context from which that routine was called. Context is useful to determine the executed control flow of a program if dynamic program behavior causes an edge to be traversed from multiple sources. Extending edge profiling, Ball-Larus path profiling keeps track of each unique *sequence* of subroutine calls, counting the frequency that each acyclic, intraprocedural path executes [10]. Path profiling typically incurs a larger overhead (e.g. ranging from 31% on average for SPEC95 but as high as 97% for gcc) than edge profiling (e.g. 16% on average for SPEC95) but provides more specific information to the developer. Path and edge profiling have been extended to lower the overheads significantly by combining both instrumentation and sampling to provide efficient profiles of programs, reducing runtime impact to approximately 1.2% with 94% edge profile accuracy [16]. These techniques provide useful insight into single-threaded programs, however, they do not support multi-threaded programs.

The techniques used to profile parallel programs depends on the paradigm used to express parallelism. For the threaded model on a single machine, instrumenting compilers with support for specific languages and threading libraries can support profiling of threaded programs. The TAU Performance System [98] provides a compiler for limited automated monitoring of multi-threaded applications implemented in C++. In addition, TAU supports a measurement API that allows the developer to add more detailed measurements through manual instrumentation.

Larger-scale applications where threads in contexts in different nodes communicate using message passing libraries (e.g. MPI) need yet another set of techniques and tools. A popular

approach is to use the above techniques to profile within a particular node in an MPI system. This can be augmented with MPI communication profiling techniques to locate the source of bottlenecks, whether they be compute or communication limited [11, 87].

2.2 Stream Processing

We begin with a brief history to introduce the context of the stream processing approach we utilize in this dissertation much of which is given in more detail in [101]. Conceptually, stream processing has a long history, spanning at least five decades. The earliest reference to the term stream that the author knows of is attributed to P. J. Landin created during the development of operation semantics as part of his work on the relationship between ALGOL 60 and λ-calculus [20]. This was a very different use of the the word stream than is generally used today, referring to models of histories of loop variables. Next came the term "dataflow systems" in the late 1960s, referring to "data flow analysis" [2] used to evaluate the potential amount of concurrency inherent in computations. Afterwords in 1974, the first dataflow language, Lucid, was developed [114]. In the same year, G. Khan published his famous paper describing what is now known as Kahn Process Networks (KPNs) [60]. While not completely general, programs written as KPNs have formally provable properties regarding termination, non-termination, and composability.

The next big advance in stream processing came with the development of synchronous dataflow [68]. Synchronous dataflow was developed to directly answer some of the drawbacks of KPNs, mainly anomalous behavior under certain conditions and propensity to deadlock. By restricting the behavior of a block to a set of well-defined input/output relationships, deadlock can be avoided and scheduling can be optimized. Unfortunately, this constraint is fairly restrictive for large, important classes of applications (e.g. stochastic filters).

In this dissertation, we refer to stream processing applications as streaming applications. We use a less strict semantic model of stream applications than synchronous dataflow. Stream applications are course-grained dataflow computations that orchestrate communication between asynchronous, atomic computation units called blocks. These blocks are interconnected unidirectionally over edges, the only mechanism over which data is communicated. This is a generalization of the restrictions of synchronous dataflow above to allow any design unit within a block, even if the relationship of output to input is unknown. An example of an application topology is illustrated in Figure 2.1. Block A is a data source, the output data stream from block A is delivered as an input stream to block B, etc.

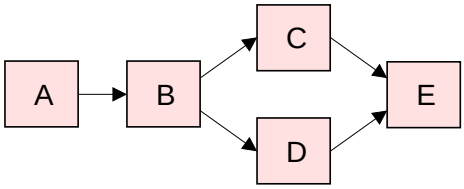


Figure 2.1: Sample application dataflow graph.

In most modern streaming languages (e.g., Brook [18], StreamIt [106]), the computation within the blocks and the interconnections of the blocks are all expressed in a common language. In the Auto-Pipe environment [23, 40], block computations are expressed in a language suited to the computational resource(s) onto which the block may be allocated. For example, blocks that may be allocated to traditional processor cores are coded in C/C++ while blocks that may be allocated to FPGAs are coded in VHDL or Verilog. The interconnections between blocks are expressed in a domain-specific language (DSL), X, which is independent of the block expression language(s) [40]. We refer to such languages as *coordination languages*. Another example is the S-Net language and runtime environment [90]. Separating the computation from communication is generally desirable but rarely achieved in many programming paradigms. Using a coordination language forces the designer to adhere to this paradigm and explicitly separates processing from communication which we believe to be beneficial especially for heterogeneous computing systems.

Streaming applications are well-suited to execution on heterogeneous architectures, in part because heterogeneous systems frequently have distributed memory infrastructures, and the explicit expression of required data movement inherent in streaming languages enables the memory to be effectively utilized.

2.3 The X Coordination Language

A coordination language is distinct from a traditional programming language in the sense that one does not express the entirety of the application within the coordination language. A coordination language is used to express the interactions between computing elements, but not the internal computations within those computing elements. A simple example application is shown in Figure 2.2. Here, a sensor of some form provides a stream of data that is filtered, accumulated, and written to a file.

The X language specification for the example application is given in Figure 2.3. In the terminology of X, the computational kernels are called *blocks*. The data into and out of

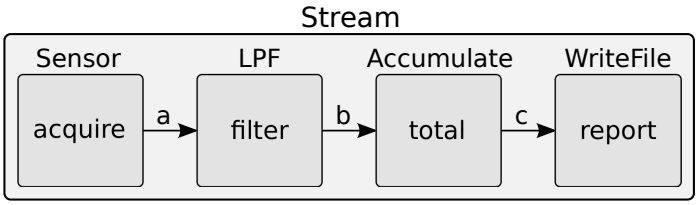


Figure 2.2: Example DSP streaming application.

each block is specified in X, but the functionality of the block itself is not. There can be one or more *implementations* of each block, each expressed in a different language, targeting a distinct type of computational resources. For example, a C/C++ implementation would target a processor core while a VHDL implementation would target reconfigurable logic.

```
block Sensor {
    output unsigned16 y; // port declaration
}
block LPF {
    input unsigned16 x;  // port declarations
    output unsigned16 y;
}
block Accumulate {
    input unsigned16 x;  // port declarations
    output unsigned32 y;
}
block WriteFile {
    input unsigned32 x;  // port declaration
    config string filename;
}
```

(a) `Sensor`, low-pass filter (`LPF`), `Accumulate`, and `WriteFile` block definitions. On all blocks, the input port is named `x` and the output port is named `y`.

```
block Stream {
    Sensor     acquire; // block declarations
    LPF        filter;
    Accumulate total;
    WriteFile  report (filename="out.txt");

    a: acquire -> filter; // topology
    b: filter -> total;
    c: total -> report;
}
```

(b) Application topology for `Stream`. As part of topology specification, edges are given labels `a`, `b`, and `c`.

Figure 2.3: X language specification for example application `Stream`. Note that block definitions only specify I/O and configuration, not function.

At the lowest level within X, the typed input and output ports of each block are specified. Blocks can then be composed into higher level blocks, which form the complete application. Also specified (but not shown in the figures) are: (1) the implementations that are available for each lowest level block, (2) the set of computation and communications resources used for application deployment, and (3) the desired mapping of blocks to resources. It is this separation between the application topology specification in a coordination language and the block function specification in other languages, point (1) above, that defines what it means to be a coordination language.

The benefits of separating the overall application into a coordination of lower-level components include the following:

- Re-use of code blocks. With a clear distinction between the language that specifies what individual blocks do and how they are composed, there is a greater tendency for the application developer to think first in terms of using blocks that already exist, rather than building new blocks. To be effective, this does require the existence of a rich block library, but once that library exists, the language separation will promote its use.

- Data movement is handled by the system, not the application developer. There are two benefits that accrue from the system handling the data movement. First, the application developer need not actually author the code required for data movement (e.g., DMA engines and the like). With less code to write, the application developer's time is saved. Second, there are fewer errors introduced since the application developer is working at a higher abstraction level.

- The application decomposition is known to the system. This facilitates the system being capable of parallel execution while being cognizant of the required data dependencies. It also facilitates the system specifying the mapping of compute kernels to computational resources.

- The application is less error-prone. Memory races don't exist. Locks are unnecessary. Reasoning about the correctness of a pipelined application is very similar to reasoning about a sequential application.

Prior to actual deployment on the heterogeneous system, the X-Sim federated simulation tool [42], which is part of the Auto-Pipe environment, provides a simulation model of the streaming application as mapped to the various devices in the heterogeneous system. Figure 2.4 shows the timestamp trace files collected by X-Sim for the simple application

in Figure 2.1. At an out-bound arc departing a device, the times that data elements are available are recorded in the file labeled T_{out}. Associated with in-bound arcs are timestamp files that record the time data elements are available, T_{avl}, and the time data elements are consumed, T_{in}.

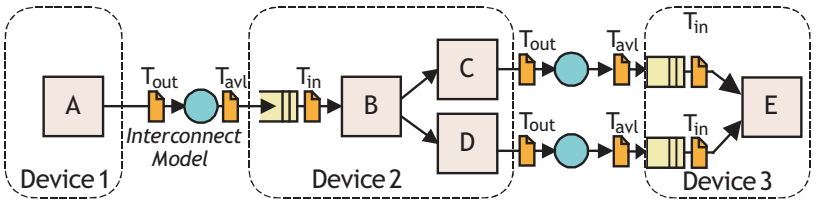


Figure 2.4: Data traces collected with X-Sim[42].

X-Sim can be used to verify both functional correctness and performance characteristics of the application by examining the trace files created as part of the simulation execution. Essentially, the simulator provides complete observability into the communication arcs of the streaming application. (Note that while the example illustration only shows traces being created for inter-device communication, there are mechanisms available for tracing interior arcs as well [42].) Once the application developer is happy with what can be learned from simulation, the task moves to actual deployment, executing the streaming application directly on the heterogeneous system. In the Auto-Pipe system, the developer specifies which application blocks are to be mapped to which devices in the heterogeneous system, and Auto-Pipe provides the appropriate communications infrastructure to effect the necessary data movement. In the example above, data from a processor core to the FPGA was moved across the PCI-X bus. Data delivery between two processor cores uses the native shared memory, data to/from the graphics engine uses PCIe, etc.

While the Auto-Pipe system ensures the data is delivered correctly, it is not uncommon for an actual deployment to have performance characteristics that differ in some way from the simulation model. This might be due to lack of complete fidelity in the models of the computing devices, the interconnection networks, or possibly their interactions. Whatever the reason, to support the developer's understanding of the actual performance realized on the deployed system, observability of runtime, dynamic performance characteristics is beneficial.

An ideal performance monitoring system would provide traces of the entire execution with full coverage of the application, similar to the trace files produced by X-Sim. However, these traces would be generated not by models of the system but by observing the application's runtime behavior through instrumentation. A significant challenge to developing such a system is how to generate and extract the trace data without perturbing the execution to

the point that the trace results are not representative of the original application. While in some cases full instrumentation may be feasible, the volume of data required and overhead to collect such data to monitor most interesting applications is prohibitive. Instead, this work focuses on extracting just enough information from an application running on a deployed system such that the user gets similar benefit to having the full trace.

2.4 Heterogeneous Computing Systems

We define an heterogeneous computing system as a set of heterogeneous computing components connected together to create a larger computational resource, typically (but not necessarily) packaged within a single enclosure. Heterogeneous systems are frequently used in high-performance embedded computing (HPEC) applications (e.g., medical instrumentation, military signal processing) where there are often stringent size, weight, or power constraints on the system. Also, as is common with traditional general-purpose processors, large collections of heterogeneous "nodes" can be networked together to form a cluster, thereby obtaining significant advantages in the high-performance computing (HPC) arena. For many compute-intensive applications, a heterogeneous system can perform significantly faster than a cluster of similar size that is constructed exclusively with general-purpose processors. Another approach to utilizing heterogeneous computing systems is to reduce the size of the cluster required to compute a given task, thereby reaping the benefits of greatly reduced power and maintainability obligations. Recent large-scale HPC clusters have demonstrated a trend towards heterogeneous computing systems as with *Roadrunner*[1], which is a Cell and Opteron based heterogeneous architecture. We will now survey the types of computing components used in heterogeneous systems.

Today, it is fairly straightforward to physically construct a heterogeneous system. Figure 2.5 shows an example of a heterogeneous system constructed using dual-core AMD Opterons, an off-the-shelf graphics card connected to HyperTransport (HT) via a PCIe bus, and an FPGA card connected via a PCI-X bus. Note that while the DRAM memory on the processors is generally accessible by all of the GPP cores, the memories associated with the GPU and the FPGA are segregated (both from the main memory and each other).

[1]See `http://www-03.ibm.com/press/us/en/pressrelease/20210.wss`.

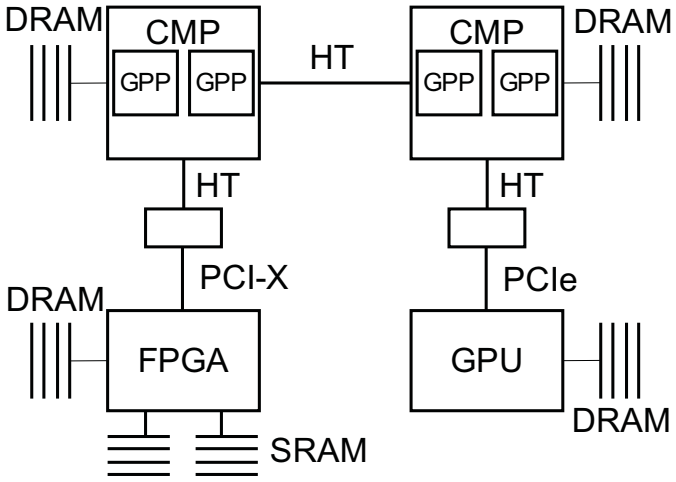


Figure 2.5: Example heterogeneous system that reflects the type we utilize in this dissertation. Two chip multiprocessors (CMP) are interconnected with a HyperTransport (HT) link. Additional HT links are used to connect to an FPGA through a PCI-X bus and a graphics processing unit (GPU) via a PCIe bus.

2.4.1 Traditional Architectures

Homogeneous multi-core processors are still the most common computing components used in both the HPEC and HPC worlds, and virtually all heterogeneous systems still maintain a healthy quantity as computing components. For a given generation of these processors, improved performance is provided via coarse-grained parallelism, and massively parallel HPC clusters have been constructed consisting of network-connected collections of single-processor nodes. As symmetric multiprocessor (SMP) systems became available, cluster nodes were expanded to include additional processors (scaling up until the local interconnect, typically a bus, was saturated). From one generation to the next, clock frequency increases and the resulting per processor performance gains had a dramatic impact sustaining this general approach to achieving high performance. Rather than invest in alternative architectural approaches, one could realize dramatic computational capacity increases simply by buying and installing the latest generation of GPPs. Recently, performance gains due to increased ILP and clock frequency have diminished and architects are exploring alternate paths to increased performance. Figure 2.6 shows the clock frequency trends of recent Intel Xeon server processors. After consistent significant increases in clock frequency for several years, the current trend is clearly toward larger numbers of on-chip processors, each running at similar (or even lower) clock rates.

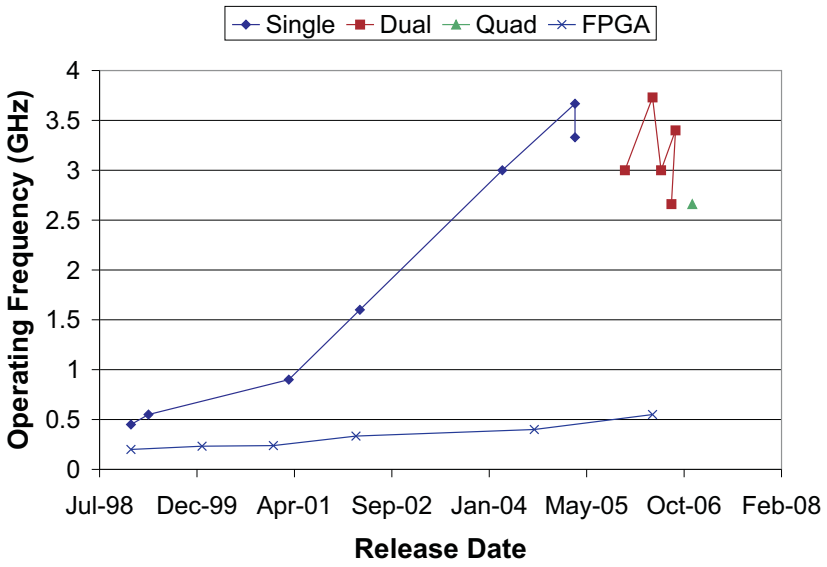


Figure 2.6: Operating frequencies of Intel Xeon processors and Xilinx Virtex series FPGAs over time. FPGA performance is for a 16-bit addition on the Virtex through Virtex-5.

HPC clusters still take similar forms today. With the arrival of homogeneous chip multiprocessors (CMPs), the core density within an SMP node is rapidly increasing. While GPPs offer many advantages in terms of flexibility and programmability, increases in per-core performance are dwindling while power requirements are soaring. The trend is therefore toward more cores on a chip, typically running at a slower clock frequency than single core processors.

While considerable research has been undertaken [12, 13, 26, 27, 54, 55, 61, 75, 97, 112, 119] into how best to construct these chips (e.g., determining an appropriate cache organization) as well as program these chips (current practice is to view them as similar to SMP systems and use memory-based synchronization), there is no clear consensus yet as to appropriate solutions to these issues. The issues are complicated further by the advent of heterogeneous CMPs (e.g., the Cell). Here, not only does one have to re-address the scheduling problem due to the heterogeneous nature of the chip, but in addition the memory model for the different processors within the chip is distinct.

2.4.2 Specialized Computing Architectures

When the performance of multi-core processors is not sufficient, further performance gains must come from alternative approaches. Special-purpose hardware is becoming an increasingly popular alternative to algorithm acceleration. By exploiting the parallelism available in algorithms, computer scientists have realized several fold speed-ups over general-purpose processors.

Reconfigurable logic, in the form of field-programmable gate arrays (FPGAs), is popular as a computing component. Historically relegated to the task of providing simple "glue logic" in custom hardware designs, FPGAs have recently enjoyed tremendous growth in capacity to the point that entire applications can effectively deployed on them. Manufacturers are also tailoring FPGAs to the needs of different application areas, including optional embedded processor cores, multiply-accumulate units, specialized I/O functionality, etc. FPGAs can achieve better performance than that of a GPP by exploiting the fine-grained parallelism present in an application to a large degree. To exploit this parallelism, an FPGA provides a large quantity of configurable logic with which calculations can be performed. By controlling this logic precisely, many calculations can be performed concurrently. In addition, FPGAs have extremely large amounts of on-chip memory bandwidth available to support the data needs inherent to this degree of parallelism. Carefully architecting an FPGA circuit can lead to a large speedup over a GPP in many non-trivial applications [1, 8, 9, 28, 48, 49, 59, 70, 95, 108, 115, 121]. El-Ghazawi et al. [36] recently published a comprehensive discussion of the applicability of FPGAs to high-performance computing applications. Even though FPGAs tend to run at much lower clock frequencies than GPPs, the clock frequency gap between the two is actually decreasing for the first time as modern GPPs scale back clock frequency in favor of more cores on a chip (see Figure 2.6). Unfortunately, not all applications are suitable for deployment to an FPGA. FPGAs generally run at lower clock speeds than GPPs, do not have native support for floating point units, rely on board designers to incorporate external components (external I/O, memory), and require significantly larger design cycles than that of software. In spite of these drawbacks, FPGAs are increasingly being used to accelerate computations in performance critical applications.

Graphics processing units (GPUs) are also a recent addition to the general computing component toolbox [18, 38, 41, 44, 52, 88]. Both major GPU manufacturers have provided the ability to utilize at least a subset of the processing elements for computations other than graphics applications. Most recently, there has been a move toward a unified stream processor architecture [86], greatly increasing their worth as a general computation resource.

Offering many programmable units running in parallel at high speeds, GPUs are well suited for deployment of data-parallel, floating-point applications.

Digital signal processors (DSPs) have long been, and still remain, the most popular choice for many signal processing applications. DSPs operate much like GPPs with the notable exception that their ISA and resulting micro-architecture have been carefully tailored to perform tasks in the signal processing domain (e.g., multiply-accumulate instructions, parallel address register computations).

Application specific instruction-set processors (ASIPs) are also a candidate for heterogeneous system deployment. Commercial examples of these systems, such as the PhysX from NVIDIA (implemented on a GPU), custom instruction-set processors from Tensilica [104], and the floating-point processor from ClearSpeed [29], promise performance and power benefits if the application can effectively exploit their capabilities.

2.4.3 The Viability of Heterogeneous Systems

There are a number of heterogeneous systems in existence, both in the research community and commercially available. The simplest to construct machines are based upon commercially available motherboards containing GPPs that are then expanded with one or more heterogeneous computing components. Several manufacturers make FPGA boards, including I/O bus-based boards from Annapolis Microsystems [7] and Nallatech [81]; and HyperTransport boards from XtremeData [120]. Nallatech and XtremeData have both recently announced boards that will connect via the Intel front side bus. GPUs are traditionally connected via PCIe slots on the motherboard. The ASIP from ClearSpeed can also attach via an I/O bus.

To avoid the limitations of standard motherboards, a number of companies have built heterogeneous systems that include higher performance interconnects between the computing components. Examples here include the SGI Altix line [96] (which uses their proprietary NUMAlink interconnect); a whole family of machines from Mercury [77] (which support GPPs, FPGAs, DSPs, Cells, etc.); and SRC systems [99]. Of these companies, SGI is focused more toward the HPC community while more of Mercury's and SRC's business is in the HPEC community.

Unfortunately, heterogeneous computing systems are not without drawbacks. Developing and deploying an application on a heterogeneous system is more challenging than traditional GPP clusters due to the heterogeneous nature of the system. Considerable research has

already been devoted to this task for GPP/FPGA systems [6, 14, 46, 63, 93, 103, 105]. The model of the computer taken when developing applications for GPP systems is often resource agnostic. Many applications are developed with a mostly unrestricted view of memory, which creates problems porting code to components with restricted available memory. The interconnect between the GPP and the non-traditional component(s) may be high latency or low bandwidth, which may cause bottlenecks not present on a homogeneous cluster of traditional processors. The number of computational units sharing an interconnect is greatly increased in some cases, which can also lead to link saturation. Finally, some computing components, such as FPGAs, require more explicit description of the fine- and coarse-grain parallelism. All of these issues transform a complex development task into a formidable Gordian knot of deployment. Understanding the runtime performance of a deployed application is an important component of the unraveling of this knot.

2.5 Related Work

In this section, we describe work that is related to TimeTrial.

2.5.1 Performance Profiling and Monitoring

Profiling tools are used to measure the performance of a computer application or system based on the time it takes execute certain tasks. They can help detect performance bottlenecks such as cache misses, pipeline stalls and measure other performance metrics in a software system. There are a large number of tools available for profiling software systems, a few that profile FPGA-based systems, and even fewer that profile applications deployed on both hardware and software.

Existing software profilers offer a wide variety of techniques to collect performance data of code running on a target processor and support different programming languages. Multiple strategies can be used to collect a diverse set of performance measurements as well as different techniques to invoke them during runtime. Since there is a wide set of tools available, we utilize the classification system proposed by Tong et al. [107].

Profilers can be broken down into software-based, hardware-counter based, and FPGA-based tools. Software-based profiling tools are the most common, measuring performance either through binary instrumentation or simulation. Hardware-counter based profiling tools utilize special counters that are exposed on modern processors. FPGA-based profilers

measure the performance of an application running on an FPGA, where that application can range from a soft-core microprocessor to an arbitrary FPGA design. We now describe some important profiling tools and how they relate to TimeTrial.

Software-Based Profilers

The most detailed profiles usually result from executing an application in an instruction set simulator. The trace of the simulation events can then be used to analyze the performance over time. The *SimpleScalar* tool set is one such tool which simulates application binaries executing on the SimpleScalar computer architecture [21]. An advantage of using simulation is that the designer has access to the entire set of data flow down to the microarchitecture registers and cache behavior. This level of detail is helpful for creating highly-optimized code without impacting the performance. Simulation is extremely slow compared to native execution or other profiling methods which makes profiling long-running programs time consuming. Simulation can lead to inaccurate profiling when the model of the architecture does not match reality, which is often the case when a simpler architecture model is used to get a faster simulation time.

GNU's *gprof* is a profiling tool that is used to profile C and C++ application code. gprof focuses on logging function execution time and frequency. The result can be presented as a flat profile or a call graph. A flat profile is a report that shows the execution time of the application broken down by the time spent in each function along with the number of times each function was called. The call graph is an edge profile described above that displays each function in the program, its parent function (i.e. the function that called it) and its children (i.e. the functions it calls). Instrumentation is added by calling the GNU compiler with appropriate flags. The instrumentation records execution times of each function by sampling the program counter periodically. The program counter determines which function the program was in when it was sampled the statistics are then updated. Sampling is a common technique used in software-based profilers to help reduce the run-time impact on the program being monitored. Regardless, this can lead to reduced accuracy in the resulting measurements.

Another approach to software profiling is memory profiling. Memory profiling focuses on detecting memory leaks, cache misses and memory allocation frequency per function. Probably the most well-known tool that uses this technique is *Valgrind* [83]. *Valgrind* can check function calls for read and writes to memory as well as for allocating and freeing memory for certain memory allocation methods (e.g. C++ new and delete). A major advantage

of *Valgrind* over tools such as gprof is the ability to determine cache behavior, showing cache hits and misses for each portion of the program being monitored. The cache profile is determined by simulating a virtual processor and the accuracy of the results depends on how well the virtual processor models the real processor. For large programs, this technique impacts the measured program significantly.

Hardware-Counter Based Software Profilers

Hardware-counter based software profilers utilize the performance counters exposed on modern microprocessors to measure performance. These hardware counters can be configured to measure specific performance events such as memory accesses, cache misses or spills, pipeline stalls, etc. Profiling in this manner requires very little instrumentation in the source code since the counters are dedicated to collecting this information. To access the values in these counters, the developer must issue specific machine instructions to the processor. Since most developers do not wish to operate at this level, nor do they want to change the source code for each type of microprocessor, a popular approach is to use the *Performance Advanced Programming Interface* (PAPI) [17]. PAPI provides high-level access to these counters that is accessed in a common way regardless of which microprocessor the applications is executed on. A commercial example from Intel is *VTune* which works on Pentium-based processors [56]. Other tools, such as TAU [98] can reduce the overhead of profiling by throttling and instrumenting a subset of functions. Manual instrumentation can also be used to measure custom performance events selectively during execution with low overhead. These results can be very accurate but require a developer to manually modify the source.

TimeTrial shares techniques with both software-based profilers and hardware-counter based profilers. However, none of the above tools natively support streaming programs. For the software portions of a heterogeneous application, TimeTrial provides the capability to profile the stream communication between blocks. The measurements are then summarized, according to the user's specification, during runtime. In this way, communication bottlenecks can be located by inspecting aggressively summarized profiles. TimeTrial uses the hardware performance counters to determine time between events, where the events are software-based.

FPGA-Based Profilers

While commercial CAD tools such as Synopsys Identify and Xilinx ChipScope and Altera SignalTap do provide visibility into FPGA designs, they are focused primarily on the debugging task (providing detailed views near a trigger event) rather than determining the overall performance properties of an application. Performance monitoring, in contrast to functional debugging, focuses less on individual occurrences of events and more on collecting and aggregating information to characterize the performance of an application on a particular system.

In [64], Koehler et al. discuss a pair of trade-offs associated with performance monitoring in reconfigurable systems. First, they describe the inherent trade off between impact and fidelity, and second, they also describe the trade off between adaptability and convenience. The impact vs. fidelity trade off positions a monitor in the space between trying to minimize the perturbation of the system being monitored and trying to maximize the total amount of performance data being collected. The adaptability vs. convenience trade off describes the ability of the user to observe a desired signal relative to the ease of use of the monitoring system as a whole.

Koehler et al. then continue with a description of a system they have constructed for monitoring of FPGA-based applications authored in VHDL. While both systems allow the user to adjust the fidelity of the data collection (and thereby implicitly adjust the potential impact), we compare TimeTrial with theirs by contrasting the approach each system has taken with respect to the second trade off, adaptability vs. convenience. The system described in [64] can monitor arbitrary interior details within an FPGA design. It instruments the design at the language level, and allows for monitoring (either via profiling or tracing) of any signals, variables, or component ports available in the HDL source files. Our system, on the other hand, constrains itself to the data streams between blocks in a streaming programming paradigm. As such, it is language-agnostic and does not require near the detailed setup as their system. In short, their system aims for greater adaptability at the expense of convenience, while our system aims for convenience at the expense of adaptability.

The system of Koehler et al. has recently been extended by Curreri et al. [31] to support FPGA designs that have been specified in high-level languages (e.g., Impulse C) rather than hardware description languages such as VHDL. Earlier work by DeVille et al.[33] explored the design of hardware probes for performance monitoring purposes in FPGA-deployed applications.

A performance profiler has also been developed for the FPGA-based TMD machine developed at the Univ. of Toronto [85]. This profiler focuses on logging both MPI communication calls as well as user-defined computation states. It is designed explicitly to profile MPI-style communication and computation, sampling events to reduce trace size. TimeTrial instead uses online metric computation to reduce performance meta-data volume.

Other FPGA monitors restrict measurement to soft-core processors. Shannon developed a performance profiler, SnoopP, to measure applications executing on a soft-core processor deployed on an FPGA. Using a MicroBlaze processor, the program counter is monitored and segment counters are loaded with ranges of PC values. For each cycle, the code segment counter is updated if the PC is within its range giving a cycle-accurate function profile. Hough et al. [53] provide a statistics module that counts performance events when the PC is in particular code regions of applications executing on an instrumented Leon 2 soft core. Performance events may observe taps into the processor architecture, measuring statistics that are not available to software.

Another approach is to provide a framework to monitor communication between processing cores by providing a standardized on-chip interconnect with instrumentation. Schumacher et al. do this within their system, the IMORC [94] framework, which consists of interconnects implemented as FIFOs or buses. In addition to bit-width conversion, IMORC also adds a set of simple performance counters for profiling communication. These counters are restricted to 'full' and 'empty' events on a FIFO which are periodically summed in a monitoring core. This is the most similar to the technique that TimeTrial uses in that is monitoring communication between cores. However, TimeTrial is more flexible in the range of measurements supported and uses the standard FPGA bus to communicate results between the FPGAs and the associated processors.

Analyzing and understanding the performance of a heterogeneous, streaming application requires a different approach than current tools use. Traditional software tools have no knowledge of accelerators. FPGA-based tools have no support for software. At best, tools such as *gprof* or *Valgrind* might be able to pinpoint the bottleneck to the accelerator by logging many samples of the software function that sends data to a particular accelerator board. There is a need for a performance tool with the capability to profile applications with support for accelerators. TimeTrial accomplishes this by monitoring focused, application-level events, aggressively reducing traces to aggregate performance events online and providing agents to monitor each accelerator platform.

2.5.2 Trace Compression

Trace compression is widely used in software simulations to reduce the data set size for quantifying evaluation of new architectural ideas and design prototypes. Compression is necessary to deal with large event streams resulting from both simulation and runtime monitoring. Consequently, much effort has been put into improving the methods for effective trace compression [79]. Simulations tend toward lossless compression techniques which do not lose any information in the compression process. Lossless compression techniques for runtime profilers tend to be highly impacting, clobbering the performance of the application being measured. Runtime profilers tend to employ "lossy" compression, summarizing the event stream by aggregating events into a set performance metrics. The software-based and hardware-counter based tools discussed above do precisely this, each tool focusing on particular aspects of the system and aggregating events into counts and time. The meaning of these counts depends on the tool and how it was configured.

None of these approaches adequately support streaming as a concurrency platform. As a result, the compression techniques used provide metrics of little interest for streaming programs. TimeTrial employs lossy compression; however, the types of measurements supported and its compression techniques result in metrics that directly support streaming semantics. Note that lossless compression could be added to TimeTrial but it would not have as low of an overhead and impact that lossy compression offers.

2.5.3 Performance Debugging Languages

TimeTrial provides a simple domain-specific language (DSL) to provide a framework for the developer to specify where, what and how to profile the communication of a streaming application. Performance related DSLs are helpful to reduce impact and overhead by empowering the developer with the capability of defining the scope and type of measurements. Relevant DSLs fall into two categories: query languages and constraint languages. The TimeTrial language supports both querying of the performance as well as verifying that constraints are met. Here we cover the relationship to other DSLs that deal with specifying some aspects of application performance.

Vetter et al. describe a performance assertion language for code segments written in C++ [113]. A function is called right before the segment of code to be instrumented. This function call has the assertion expression as a parameter to the call. The end function is similarly inserted at the end of the code segment to be measured. The expression is then evaluated at the end

of the measured segment and an error is thrown if it fails. The language supports a number of useful variables such as the number of instructions, cycles, and architecture-dependent constants such as peak IPC.

A user-driven query language, Metric Description Language (MDL), is presented by Hollingsworth et al. [51]. MDL is part of the Paradyn Parallel Performance Tools for performance debugging of HPC systems [80]. The purpose of this language is to specify what data to collect about a running program in an platform-independent manner. It also provides a method to constrain performance collection to particular resources such as objects, procedures, nodes or files. MDL provides two basic performance constructs: instances of counters and timers. Custom metrics can be defined to compute an online metric of choice. While MDL does help take some of the book keeping off the burden of the program, high-level analysis such as inter-process communication can not be measured.

SelfTalk, a performance query language for measurement of multi-tier systems is described by Ghanbari et al. [43]. SelfTalk is designed to allow a developer to express their understanding of what the performance should be at a high level. SelfTalk takes performance expectations in the form of performance hypotheses. The parameters for these hypotheses can be derived from manual measurements or models and contexts can be defined using qualifiers. SelfTalk is mostly useful for validating performance after detailed modeling or measurements have been made. SelfTalk hypothesis statements are a more general (and complex) form of the TimeTrial language `assert` statements.

Another language used to specify measurements is from the *CoSMoS* performance monitoring system [100]. Measurements are specified similar to a function call in C++ but are tagged with a special leading character for the instrumentation system to recognize them when parsing. The language itself is not described formally; measurements are meant to be specified using the CoSMoS source code browser GUI. This approach is less general than the previous two in that it is designed for instrumenting single lines of code with counters.

Chapter 3

Profiling the Performance of Streaming Applications

In this chapter we begin by describing our approach to building profiles of streaming applications. Next we discuss our strategy to reduce the volume of raw measurement data needed to build such a profile. We then describe framing, our strategy for capturing profiles through time. Finally, the TimeTrial measurement language is given, its capabilities are described, and examples of its use are given. Note that here we limit our description to the design of TimeTrial's approach, functions and capabilities. The architecture and implementation details are described in Chapter 4.

3.1 Profiling a Streaming Application

A profile of a streaming data application is somewhat different than a profile of a sequential program. Like profiles of parallel MPI programs, streaming application profiles should indicate the performance-limiting sections of the application in terms of both communication and computation (i.e., what is the performance bottleneck). Furthermore, measuring a reasonable global view of a distributed application's performance is not straightforward. Distributed applications do not typically share storage, nor is there a universal program counter that accurately reflects the state of the application. For example, given the application in Figure 3.1, a stream profile should answer questions of the type:

- At what rate is data moving across the link that connects "Convert" to "FFT"?
- How long does it take data to travel through stage 2?
- What is the utilization of the "FFT" kernel? The "Measure" kernel?

- What is the occupancy of the queues between each block?
- What fraction of the time is back-pressure being asserted from stage 3 to stage 2?
- What portion of the pipeline is limiting the achievable throughput?
- If that bottleneck were resolved, what would be the next bottleneck?

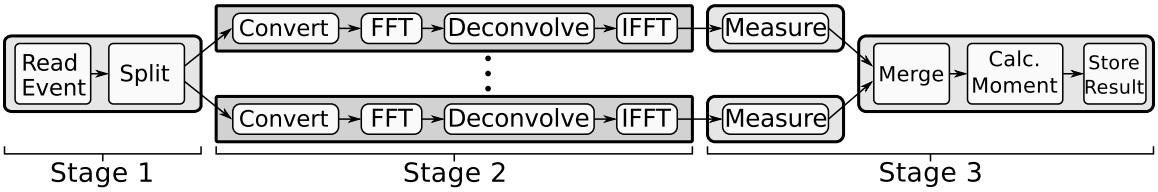


Figure 3.1: An example streaming application from computational astrophysics [110]. Computation is performed within each kernel, communication is one-way along edges.

An effective profile provides useful summaries of performance metrics. A performance metric is a measurement of a system used as a quantitative criterion for judging perturbations of an application. The most widely used metric is execution time of a program. By comparing the execution time of an instance of a program, the effects various optimization efforts can be quantified. Execution time is a useful metric to determine changes in runtime; however, it gives little insight into why the program is performing that way.

A common technique when debugging application performance issues is to locate the sections of the program that are contributing negatively to the overall performance. For parallel streaming programs, a developer is likely interested in metrics that enable her to narrow down the bottleneck to a block or edge. Then, the cause of this performance can further investigated. We provide performance metrics in TimeTrial that help with the first step: locating the bottleneck block or edge. Metrics that are helpful with this task include throughput of an edge, utilization of an edge or block, or latency from one block to another.

3.2 Online Aggregation of Performance Events

Complete performance profiles of streaming data applications would provide a full trace of timestamps for every event in the entire program, regardless of the computational resource, over the entire execution of the program. An event could be either data-movement from one block to another or aspects of execution within a stream block. This trace could then be used to analyze the execution and compute any metric the user desires offline. Practically, full trace collection is infeasible for most real-world applications.

One approach to gathering a profile is employed in X-Sim [42], a federated simulation environment associated with the Auto-Pipe system, targeted toward streaming applications. X-Sim logs traces for heterogeneous applications by limiting the scope of collection to communication events. The result of X-Sim logging is a sequence of data transfer timestamps for each edge. This approach allows for lower overhead than full trace collection while providing the necessary information to debug both functional and performance aspects of the program, deployed across processor cores and FPGAs. While an improvement over full trace collection, X-Sim style simulations are most useful for monitoring short executions since it still executes much slower than native execution.

TimeTrial's approach is to calculate performance metrics along side the execution of the program, aggressively reducing the storage and communication requirements compared to trace collection. The performance metrics supported are chosen to be both informative, low-impact, and for their ability to be calculated on both FPGAs and processors. Performance metrics are generated by monitoring performance events in the application and updating the values online. TimeTrial requires that there be available unused resources on each computational resource to implement the measurements. The computational resources used to implement the TimeTrial monitoring agents are dedicated resources (i.e., resources not shared with the application). On an FPGA, this is accomplished by allocating area for a monitor. In software, one or more processor cores are dedicated for monitoring. An advantage of using metrics that can be reduced to a small set of values is that the communication of these metrics off the FPGA for storage can use the same I/O paths as the application with minimal impact on the throughput. Sharing the I/O path allows TimeTrial to be easily ported to future heterogeneous systems.

3.3 Framing: Measuring Performance Through Time

Runtime monitors aim to measure performance during execution of the program. In an ideal scenario, the monitoring system should log timestamps and state information for every event, like full simulations systems offer but operating at native application speed. Additionally, this information should come at no overhead or interference cost to the application being monitored. Achieving this ideal is impossible on most real systems as the bandwidth and storage constraints are limiting factors. Our solution to this issue is data aggregation.

Aggregating performance measurements all the way down to a single result for the entire run may not be optimal for applications with long execution times. This runs the risk of

"averaging out" deviant performance situations (good or bad) that might be of interest to the user. Note, however, that as heterogeneous applications push the performance envelope of a system, it is increasingly more important to be able to correlate application performance with application behavior. Being overly aggressive with aggregations limits the developers ability to observe this cause and effect to find the root cause of a performance anomaly. Hence, an effective performance measurement system needs to have the ability to throw away a multiplicity of information while capturing application-dependent aspects which have high value for performance debugging.

To enable the user to control this trade off, TimeTrial automatically breaks up the execution into non-overlapping segments called *frames*, then computes and stores the requested metrics over each frame. We support two types of frames: time frames and data frames. If time frames are requested, TimeTrial will measure for the given time period and report a result. For example, assume that the developer is interested in the mean ingest rate of a block and the application under test executes for 1000 seconds. If he or she chooses a frame period of 100 seconds, TimeTrial will report 10 mean rates measured on that run. In the second mode, the user can specify the size of data frames in data elements (or bytes) instead of time. Data frames are useful for measuring variability in applications where performance is dependent on the content of the data, allowing for a dynamic frame period.

Statistics are initialized each time a new frame is encountered and the results are reported at the end of a frame. In this way, the system provides an ordered *set* of frames, each containing its measured performance metrics, from which he/she can reason about the behavior of the application. The cardinality of the set is determined by the frame size. The set of frames, or a subset of frames, can optionally be further aggregated to provide performance information about the entire run if the user so desires.

In TimeTrial, setting the frame size smaller than the execution time will split the execution into more than one non-overlapping segment retaining 100% coverage of the run. In an ideal environment (without resource constraints), the user could specify a frame size of one datum, since this provides the most information about the execution of the application. Realistically, the frame size should be chosen to provide a sampling frequency that exceeds the Nyquist frequency of the signal of interest in order to capture high-frequency or rare performance events. Note that in many cases, even this basic condition may not be feasible. Choosing too low a frame size may have a large impact with the execution of the application itself. Collecting performance information in frames aggregates only the data within the frame instead of the data over the entire run.

Figure 3.2 shows an example execution with different frame periods. Choosing the frame period enables the developer to explicitly control how much time resolution remains in the aggregated performance meta-data versus the overhead TimeTrial will incur making measurements. A large frame period will be low overhead but likely to 'average-out' any rare performance events. In this example, a mean rate of 30 MB/s is recorded. Successively smaller frame sizes reveal that there are portions of the execution that perform well and other portions that perform poorly. This might be cause to investigate the circumstances around the poorly executing region(s). Note that reducing the frame period results in a larger number of frames that must be handled by TimeTrial and logged to disk. Setting the frame period based on event count (instead of time) is helpful for correlating data-dependent performance events as they flow through the application.

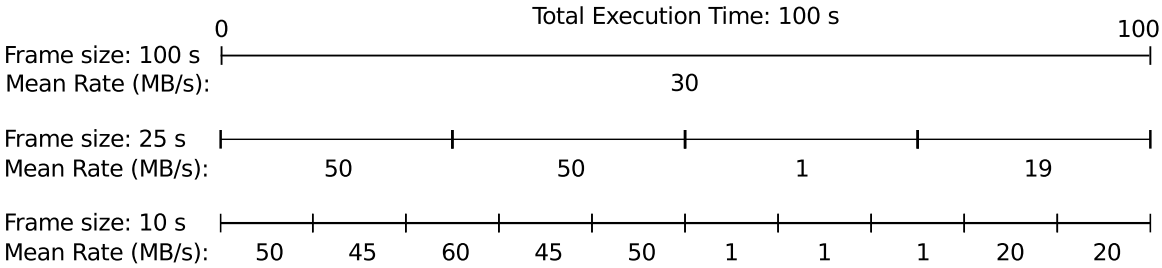


Figure 3.2: Illustration of measuring with varying frame periods. Each frame results in one aggregated metric.

Even small time or data periods will yield some benefit over storing raw traces. Consider an execution where the developer simply wants to know the occupancy of a FIFO presented as a histogram. A trace-based approach would record a stream of enqueue and dequeue events over the run and post-process them into a histogram. Let's assume that there were 10^9 elements that flowed through the FIFO, and that each time stamp could be stored in four bytes. Computing the histogram offline would require eight gigabytes of performance meta-data to be communicated and stored for post-processing. Computing the same metric online using 512 bins with eight bytes for the count in each bin requires only 4096 bytes of storage. Note that this is the case for a frame period equal to the entire execution time. Each additional frame adds another 4096 byte storage requirement for our system.

3.4 The TimeTrial Performance Query Language

TimeTrial implements performance profiling by first analyzing the X language source code for performance query expressions and then instrumenting the resulting binaries with measurement code to collect runtime statistics. These performance query expressions, in the

form of TimeTrial statements, are designed to be user friendly (i.e., straightforward in their meaning) while enabling the compiler to automatically implement low-impact, full-execution runtime instrumentation.

In order to minimize the measurement impact, TimeTrial selectively profiles only the portions of the application that the developer is interested in. TimeTrial constrains its view of the application to the communication edges as a way to provide a simple mechanism for reasoning about performance and staying agnostic to the underlying implementation language used to build blocks. Targeting communication allows TimeTrial to identify bottlenecks at the block level. That is, TimeTrial is designed to efficiently identify one or more blocks that are performance-limiting during a run. The developer can then focus on optimizing those blocks and then re-execute the application.

To introduce the TimeTrial language, we begin with two example performance queries of the application of Figure 3.3, "What is the throughput of edge `a`?" and "How long does each datum spend in edge `b`?" These queries can be stated as measurements in the TimeTrial language as:

```
m1: measure rate at Stream.a

m2: measure mean latency at Stream.b
```

The first performance expression begins with a label, `m1` followed by the keyword `measure`. The keyword `rate` specifies the type of measurement, followed by `at` and the edge `Stream.a`, the target of the measurement. The second performance expression, `m2`, again uses the keyword `measure`. `rate` and `occupancy` specify the "metric type," which tells the compiler which performance metric to implement. Following the `at` keyword is the "edge label," the target communication edge to observe. The second statement has the optional "aggregation function" `hist` which tells the compiler the type of statistical aggregation to perform on the trace data during runtime. In this case, `hist` implements a histogram function on the data, resulting in a histogram of queue occupancy for that edge. Figure 3.3 depicts the logical insertion of these measurements into the application `Stream`.

The target of a measurement is restricted to an edge, as defined in the streaming application language. Each edge declares the location(s) within the application that the performance expressions observe.

Currently, the TimeTrial language supports these metric types (units are given in parenthesis):

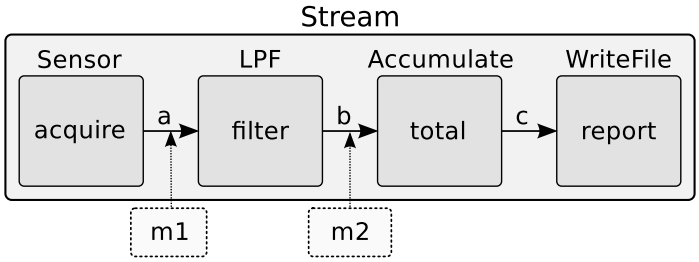


Figure 3.3: Stream application instrumented with measurements `m1` and `m2`. Dotted lines and blocks represent the performance monitoring instrumentation.

- `rate`: The throughput at which data elements are transiting an edge queue. (transfers per second)
- `util`: The utilization of a communication edge (a normalized rate). (fraction)
- `backpressure`: The amount of time spent blocking on an insertion into a queue. (fraction)
- `latency`: The amount of time a data element spends in a queue. (time)
- `occupancy`: The number of elements in a queue. (count)
- `value`: The value of the data elements inserted in a queue. This is enables functional stream debugging when combined with the trace aggregation function. (varies, depending on target edge data type)

The aggregation function is important to the efficiency of the TimeTrial system since it specifies the degree of data compression that is to be performed. Instead of the traditional method of generating a trace for each event on a target and aggregating data offline, TimeTrial compiles the aggregation function into runtime code that compresses (in a lossy manner) the event stream it is observing. This is effective since performance monitoring, when contrasted with functional debugging, is generally much more concerned with aggregated metrics as opposed to individual element values. Since the developer is able to specify the precise metric and the mechanism to summarize that metric, the compiler can implement highly optimized, low-impact monitoring.

The following aggregation functions are supported on metric types:

- `min`, `max`: minimum, maximum of metric values
- `mean`: arithmetic mean of metric values

- `hist`: histogram of metric values
- `sum`: sum of metric values
- `trace`: a log of each metric value

The aggregation function combined with the metric type and target completely specify where the compiler needs to place instrumentation into the system.

The TimeTrial language is intended to augment streaming languages with the ability to articulate performance statements about the application *at the level of block interactions*, regardless of which computations resource is targeted (e.g. FPGA vs. CPU). Understanding the performance at the block level is useful for locating bottlenecks to particular blocks, or a set of blocks, in a streaming application. By not attempting to query internal to a block, TimeTrial can stay agnostic with respect to what language is used to specify the functionality within a block. It can stay focused on the block interfaces and interactions.

Performance measurements operate on a multiplicity of events on taps and aggregate these into one or more statistical measures per execution. As each segment flows across a tap on a streaming application, the performance is summarized within that frame. TimeTrial allows the developer to choose the frame size at runtime to suit.

Some of these measurements have only one logical value per data frame (e.g., rate, utilization), whereas others are multi-valued during the frame (e.g., occupancy, value). Hence, the aggregation performed can depend on both the type of measurement and the developer's desire for detail.

In addition to performance measurements, the TimeTrial language also provides a construct for specifying performance assertions through the `assert` statement. Assert statements let a developer specify high-level performance requirements of the application, such as, "The throughput of edge `a` should always be at least 100 mega-transfers per second." Assertions are useful for designs that are nearing the final stage of development to ensure performance hypotheses are not broken on a wide variety of data sets execute within the performance specifications. Writing this example in TimeTrial looks like:

```
a1: assert m1 >= 100 Mtps
```

Thus far, we have introduced the use of two TimeTrial performance expressions, `measure` and `assert`. These expressions form the basis for TimeTrial language statements. However,

the TimeTrial language provides the ability to articulate more powerful expressions as well. Here we introduce the notion of conditionals and Boolean expressions followed by their use within the context of `measure` and `assert` expressions.

A conditional in TimeTrial is used to express a propositional logic predicate. Conditionals are specified as a measurement, a relational operator (e.g. `>`, `<=`), a value, and a units specification or another measurement. This should look familiar, since we used a conditional (albeit a simple one) when writing the assert `a1`. The conditional is everything that follows the keyword `assert`. The units can be chosen from a list of time units (`s, ms, us, ns`) or rate units (`tps, ktps, Mtps, Gtps`), which represent data transfers per second. Unit specification is optional and appropriate default units are inferred from the measurement type (e.g., time for a latency measurement, unit-less for a utilization measurement). Note that in the case of a measurement type that returns a Boolean value (e.g. backpressure), the relational operator and what follows is omitted.

Conditionals are evaluated for each instance under which the underlying measurement can hold a distinct value. As such, if a data aggregation is specified the conditional is only evaluated once per data frame. The default aggregation for a conditional is trace aggregation (i.e., the conditional is evaluated for each instance).

Complex Boolean logic expressions can be formed. These Boolean expressions are formed by combining any number of conditionals with and, or, and not (`&, |, !`) operators. The simplest Boolean expression is a single conditional statement, as in `a1`. Boolean expressions are used to qualify an `assert` statement, or are used to restrict the scope of a `measure` statement by using the `when` keyword. The `when` keyword is added to the end of a measurement statement to specify that a measurement statistics are to be collected *only* when the Boolean expression evaluates to true.

To illustrate the expressiveness of Boolean expressions, we provide two example TimeTrial performance statements shown below that utilize Boolean expressions to qualify the scope of the statement.

```
m3: measure mean occupancy at Stream.b

m4: measure m1 when (m3 >= 1 & !backpressure at acc_out)

a2: assert (m1 < m4) | (util at acc_in < 0.5)
```

The measure statement `m4` above directs TimeTrial to measure the throughput into `total` when there is data available in its input queue and no backpressure from it output. This enables the developer to measure the achievable throughput of `total` (i.e., when it is not hindered by its environment). The assertion instructs the compiler to test whether the unqualified rate measure `m1` is lower than `m4` or whether the utilization of `total` is fairly low (below 0.5). If both of these expressions are false, the assertion fails. Expressions such as these allow for articulation of very specific performance statements that enable reasoning about the performance of an application in different contexts.

Currently, the TimeTrial compiler can generate software and hardware instrumentation for measure statements without Boolean expressions. All of the above statistic types and measure types described above can be automatically generated as well. Some assert statements are supported by checking them offline as a post-processing step.

The decision to develop a new domain-specific language is not a something that should be done without extensive analysis. For TimeTrial, the decision was made because of my desire to enable a developer to direct performance analysis in the most straightforward way possible with high correlation to the application under test. The set of features available in the TimeTrial language were targeted to support a wide variety of performance queries while gently steering the user toward light weight measurements. As heterogeneous systems continue to evolve, new measurements and statistics can easily be added to TimeTrial by adding the corresponding keywords to the language grammar and implementing the resulting statements in the TimeTrial compiler.

3.4.1 Formal Language Definition

Here we articulate the formal grammar of the TimeTrial language. The language grammar is described in Extended Backus Naur Form (EBNF) [118]. To be clear, here are notes on the EBNF syntax we use. Figure 3.4 gives the formal syntax of TimeTrial.

- A symbol on the left-hand side of ::= is defined by its substitution on the right.
- Symbols in **boldface** are non-terminal symbols, and begin with upper-case.
- Symbols in `plainface` and those found in single-quotes (`‘’`) are terminal symbols.
- The pipe character (`|`) delineates substitution choices.
- Parenthesis (`()`) group a set of symbols into one logical symbol.

PerfStmts	::=	(**AssertStmt** | **MeasureStmt**)+
AssertStmt	::=	[**Identifier**:] `assert` **BooleanExp**
MeasureStmt	::=	[**Identifier**:] `measure` [**StatType**] **MeasureType** `at` **TargetType** [`when` **BooleanExp**]
BooleanExp	::=	`!`**BooleanExp** | **Conditional** `&` **BooleanExp** | **Conditional** `'|'` **BooleanExp** | **Conditional**
Conditional	::=	**CondOperand** [**RelOp CondOperand**]
CondOperand	::=	**MeasureType** `at` **TargetType** | **Number** [**Unit**]
StatType	::=	`(min | max | mean | sum | hist | trace) ['('`**ParamList**`')']`
MeasureType	::=	`rate | util | occupancy | latency | backpressure | value`
RelOp	::=	`> | >= | < | <= | == | !=`
Unit	::=	`s | ms | us | ns | tps | ktps | Mtps | Gtps`
TargetType	::=	**PortLabel** | **EdgeLabel**

Figure 3.4: EBNF for TimeTrial. Non-terminals that are not explicitly defined in the grammar either come from the target streaming language (e.g., **PortLabel** and **EdgeLabel** are identifiers of ports and edges, respectively) or follow common usage (e.g., **Number**, **Identifier** and **ParamList**). While not explicitly included above, parentheses are also supported in Boolean expressions for the purpose of describing operator precedence.

- Square brackets (`[]`) group a set of *optional* symbols into one logical symbol.
- An asterisk (`*`) follows a symbol that may be replicated *zero* or more times.
- A plus sign (`+`) follows a symbol that may be replicated *one* or more times.
- White space is ignored.

3.4.2 Examples of Use

We now illustrate the language by expressing the questions we posed above as TimeTrial language statements. These questions are of the type we want a good profile of a streaming application to be able to answer. The questions are repeated, equivalent TimeTrial language statements given and discussed where explanation is required.

- At what rate is data moving across the link that connects "Convert" to "FFT"?

  ```
  measure rate at Convert.out -> FFT.in
  ```

- How long does it take data to travel through stage 2?

  Currently the TimeTrial language does not support this measurement directly. Once support for specifying a path through the application has been added, the following could express this query:

  ```
  measure mean latency at <Path Spec>
  ```

- What is the utilization of the "FFT" kernel? The "Measure" kernel?

  ```
  measure util at FFT.in
  measure util at Measure.in
  ```

- What is the occupancy of the queues between each block?

  ```
  measure hist occupancy at ReadEvent.out -> Split.in
  measure hist occupancy at Split.out =< Convert.in
  measure hist occupancy at Convert.out -> FFI.in
  etc.
  ```

- What fraction of the time is back-pressure being asserted from stage 3 to stage 2?

```
measure mean backpressure at Measure.in
```

- What portion of the pipeline is limiting the achievable throughput?

  This query requires profiling edges from the end of the application back towards the front until the bottleneck is found. The most straightforward way to express this is by asking for queue occupancies.

  ```
  measure hist occupancy at CalcMoment.out -> StoreResult.in
  measure hist occupancy at Merge.out -> CalcMoment.in
  measure hist occupancy at Measure.out => Merge.in
  etc.
  ```

- If that bottleneck were resolved, what would be the next bottleneck? In some situations, it might be clear where the next bottleneck will lie. This could be the case if the occupancies of two queues in tandem were both high. Resolving one would likely throw the bottleneck onto the other. In other situations the next bottleneck might not be easily determined. Utilization measurements are a lightweight way to get some insight into the remaining capacity of the links:

  ```
  measure util of Measure.in
  measure util of IFFT.in
  measure util of Deconvolve.in
  ```

The TimeTrial language was designed to be a simple way to query the performance of an application and as the above examples show, the statements lend toward performance questions a developer would like to ask about an application. The details of how the compiler interprets the language and implements the measurements are described in Section 4.5.

3.5 Chapter Summary

In this chapter, we described the design approach used in TimeTrial to measure the performance of a streaming application. We articulated the appropriate types of measurements we feel are useful to a developer. We then introduced the TimeTrial query language as the interface for a developer to choose what portions of the to application to profile. The chapter concluded with example performance queries stated in our query language.

Chapter 4

Architecture and Implementation of TimeTrial

This chapter begins with an overview of the design goals when implementing TimeTrial and is followed by a description of the architecture of the TimeTrial measurement system. The instrumenting compiler is then described as integrated with the Auto-Pipe development environment and the X language. The chapter concludes with descriptions of the runtime measurement agents along with an evaluation of the limits of their performance.

4.1 Overview

TimeTrial was designed to expose performance characteristics at the level of block interactions; that is, it treats blocks as black box objects and instruments their ports and the communication links between them. While many streaming languages (e.g., Brook [18], StreamIt [106]) express block functionality in the same language as the application topology, it is frequently advantageous to distinguish between block implementation languages (chosen to be appropriate for the target computational resource) and the coordination language that expresses the application's streaming topology (e.g., X/Auto-Pipe [40]). By focusing on the application topology rather than block internals, the TimeTrial compiler is able to automatically instrument the communication edges as they are deployed. This provides a mechanism for TimeTrial to instrument the application at a reasonable level of detail without needing support for a large menu of languages. Note that, focusing on monitoring the edges connecting blocks does not preclude the use of TimeTrial for intra-block measurements. Monitoring can occur within a block if the block developer sends the information to be monitored to an (potentially newly created) output port just for measurement.

TimeTrial measures the performance of a streaming application by implementing directives from the developer specified in the TimeTrial language that are combined with the specification of the topology and mapping. These performance directives specify the what and where of the measurements. TimeTrial then analyzes these directives and instruments the application to collect runtime performance measurements. This approach has the advantage of very high specificity, measuring only the behaviors that interest the developer. Also, knowing up front what is important to the developer exposes more opportunities to optimize the runtime instrumentation for minimal perturbation of the application.

The TimeTrial system is made up of three major parts: the TimeTrial language, instrumenting compiler, and the measurement agents. Statements in the TimeTrial language instruct the compiler "what and where" to measure, the compiler instruments the streaming application (including processor cores and FPGAs, if necessary) with measurement taps, and the agents collect runtime statistics, logging them to disk.

One major challenge when monitoring FPGAs is that the memory capacity available to hold performance meta-data is more constrained than in software. Additionally, the communication link(s) between the accelerator(s) and CPUs might already be saturated by the application. Simply logging event traces in available memory leads to unacceptable interference due to the overhead of communicating these traces off the accelerator. To mitigate these effects, TimeTrial performs aggressive data reduction operations, chosen based on the developer's desired measurements. TimeTrial uses developer input to target exactly the portions of the application of interest (e.g. potential hot spots), regardless of what computational resource that portion executes on, in a minimally invasive manner. Here we describe TimeTrial in the context of heterogeneous nodes comprised of multi-core CPUs and FPGAs.

Consider the following generic streaming application, a tandem pipeline shown in Figure 4.1. The blocks in the application are labeled "`A`" through "`E`". The X code fragment that describes this simple topology is below.

```
A -> B -> C -> D -> E;
```

Blocks denoted with a circle have been mapped to a processor core, and blocks denoted with a square have been mapped to an FPGA. Associated with each edge is a queue, which buffers data generated by the upstream block for the downstream block. These queues, and the associated runtime infrastructure that performs the data movement between blocks, are the responsibility of Auto-Pipe. For example, from block `A` to block `B` Auto-Pipe will

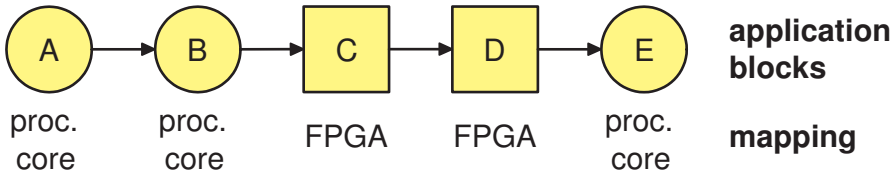


Figure 4.1: Example application topology (`A` → `B` → `C` → `D` → `E`) and its mapping to processor cores and FPGA.

instantiate a shared-memory buffer, while from block `B` to block `C` Auto-Pipe will invoke the appropriate low-level mechanisms (DMA engine, kernel buffers, etc.) to move data from software to the FPGA hardware.

On each computational resource, a TimeTrial agent is deployed that monitors the portions of the application deployed on that resource. The TimeTrial compiler inserts *taps* into the communication segments to be monitored, and event streams are sent to the agent local to that resource to aggregate the results during the run. The FPGA agent sends its aggregated data to the software agent, which is responsible for combining results from the software taps as well as the FPGA monitor.

Figure 4.2 shows TimeTrial instrumenting the streaming application of Figure 4.1, portions of which are deployed in software and other portions on an FPGA. For each accelerator, a TimeTrial monitoring agent is added to the application to collect and communicate performance meta-data to the TimeTrial server thread. *Taps* are inserted into the communication links, which contain FIFOs. When data flows in and out of a link, and event occurs on the respective tap and the event is processed by the FPGA performance monitor agent. In Figure 4.2, note the multiplexer that supports the sharing of the link from the FPGA between application data and performance meta-data. The use of aggressive data reduction within the FPGA performance monitor agent helps mitigate the impact of this resource sharing. The performance meta-data from the FPGA agent is communicated to the software performance monitor periodically during the execution at frame boundaries. In software, similar instrumentation is inserted in the form of software taps that send data events to the software agent. On a multi-core processor, the software agent is implemented as a separate process that gets schedule to an un- or under-utilized core. Software events are combined with other agent's results to perform further aggregation of the performance data.

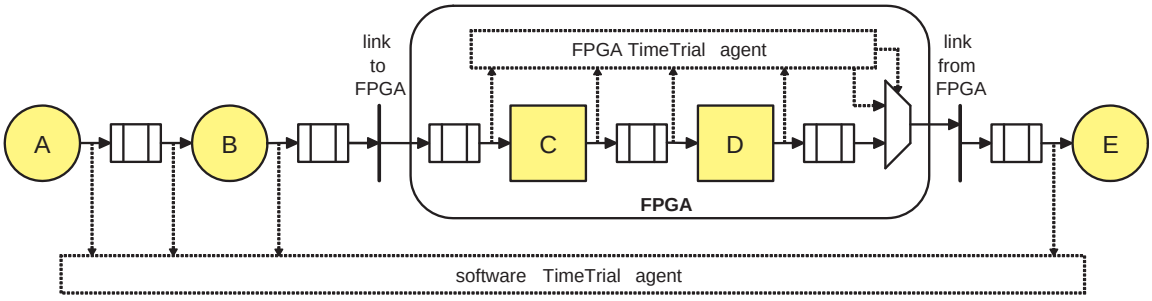


Figure 4.2: Example application from Figure 4.1 deployed on an heterogeneous platform comprised of processor cores and an FPGA. The dotted lines and boxes illustrate the runtime instrumentation of the application via TimeTrial.

4.2 TimeTrial Compiler

The TimeTrial compiler analyzes `measure` statements to determine where to insert taps on the queues which make up the edges between blocks of the streaming application. The compiler is capable of instrumenting three types of edges: edges contained entirely in software, edges contained entirely on an FPGA, and edges from an FPGA to software or from software to an FPGA. To facilitate data collection and aggregation, the compiler generates code to send the information gathered from the taps to the software TimeTrial agent.

For edges between software blocks, the compiler generates code to tap the edges and send the necessary signals to the software TimeTrial agent. To allow communication between the software process running the block and the agent, the compiler inserts initialization code that opens a communication channel to the agent and sends the agent start up messages to inform it which metrics to track and the data aggregation function to use. Depending on the metric type, the compiler will instrument the enqueue signal, the dequeue signal, the full signal, and/or the value enqueued. For example, to measure the rate of an edge, the enqueue event is tapped. Since the compiler emits C++ code to instantiate the blocks, the tap takes the form of a function call immediately before the monitored operation. This function call sends a message containing the type of event and a time stamp to the software TimeTrial agent.

Software blocks are often mapped to separate processes. The queues for edges between these processes are implemented using shared memory or network sockets. For some metric types, only one of the processes need communicate with the software TimeTrial agent. However, for metrics such as queue occupancy, where both the enqueue and dequeue signals are required, both processes must communicate with the agent. Thus, the software TimeTrial

agent is contained in a separate process and the Auto-Pipe processes communicate with the agent using shared memory queues.

The compiler instruments queues contained on an FPGA using the FPGA TimeTrial agent. To use the FPGA TimeTrial agent, the TimeTrial compiler generates VHDL code which instantiates the agent and taps the necessary queue signals. The compiler then generates code for the C++ process that controls the FPGA device to communicate the aggregated statistics from the FPGA TimeTrial agent to the software agent.

Data delivery between the software subsystem and the FPGA subsystem deserves additional explanation. In Figure 4.2, the communication between application blocks B and C is shown to include a software FIFO, the physical link to the FPGA, and a second FIFO on the FPGA. Note that there are taps at the beginning of the path (in software) and at the end of the path (on the FPGA), but no instrumentation between these two paths. As is often the case, the low-level DMA engines, etc., that deliver the data across the physical link are opaque and inaccessible for monitoring. We address this issue by considering the entire path to be a *virtual queue*. Similarly, a virtual queue is present in the application edge connecting block D to block E, this time from the FPGA to software. TimeTrial's approach to monitoring virtual queues is described in Chapter 5.

4.3 Architecture of the FPGA Agent

The TimeTrial compiler instantiates an FPGA agent on each FPGA device where measurements are to be taken. Figure 4.3 shows an overview of the architecture of the FPGA agent. The operation of the monitor is controlled by a set of commands under processor control. Commands trigger operations such as resetting the monitor, enabling and disabling individual tap monitors, setting the reporting frame period, and triggering an additional report from one or more tap monitors.

FPGAs only have a small amount (∼1-2 MBytes) of on-chip memory, making it impractical to save full traces of monitor meta-data. Traces could be stored if external memory is dedicated to monitoring, but such memory is not available for all FPGA boards and applications. To mitigate this constraint, runtime data aggregation is used to reduce the volume of meta-data. TimeTrial instruments the design with runtime logic that monitors a set of locations in the application and computes a reduction function on the values. The intermediate results for each run are then stored in limited temporary storage and communicated to the software performance monitor opportunistically.

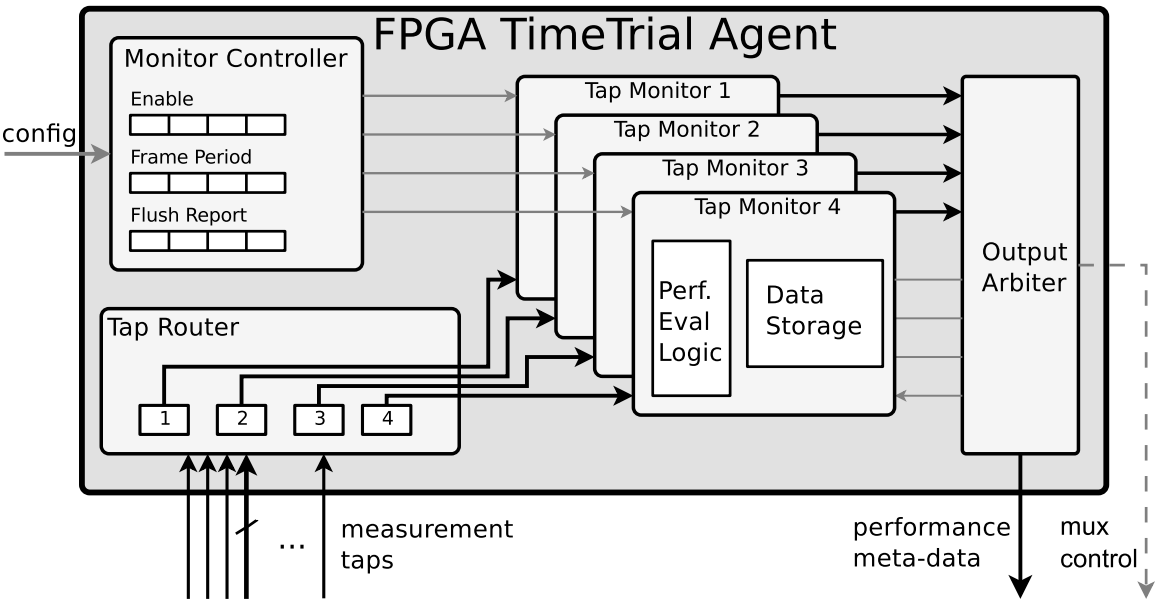


Figure 4.3: Detailed view of the TimeTrial agent for the FPGA shown with for tap monitors. The FPGA agent is a high-speed, parametrized circuit designed to aggregate measurements on the FPGA. Data paths are shown as dark lines, control paths are grey lines.

The tap monitors, on a cycle-by-cycle basis, observe the event streams on taps and perform the requested aggregation function over each frame. Currently, there are four different types of tap monitors supported: activity monitors, latency monitors, queue monitors and histogram monitors. Activity monitors count the number of cycles a signal or wire is active combined with period or size of the frame (whether clock cycles or event occurrences). Activity monitors are typically used to measure control signals for the communication channel, giving utilization, rate, and backpressure on a link. Latency monitors take a time stamp of two events and aggregate the differences over a frame. Queue monitors measure a queue occupancy by observing the stream of enqueue and dequeue events on a target FPGA communication FIFO. This monitor is invoked when queue occupancy is requested. Histogram monitors aggregate target data into a histogram of those data values. This is used for both queue monitors and when a data value histogram is requested. The architecture is designed to be easily extensible for additional tap monitors as new language features are added.

Since each tap monitor may, in general, operate concurrently, there are two levels of multiplexing necessary to send data out. First, the output arbiter chooses which tap monitor is allowed to send a report. After each frame, the FPGA agent returns the data it has collected over the past frame to software where it is forwarded to the software TimeTrial agent. Then the report is inserted into the data stream via the multiplexer in Figure 4.2.

The FPGA agent is a full duplex module; it can monitor the application while simultaneously reporting the summary of the previous frame back to the software agent. This is implemented by double-buffering the meta-data storage blocks. Currently, on-chip memories (either Block RAMs or distributed memories) are used to implement the meta-data storage system, however, other storage options could easily be supported by providing an interface to the external memory. Some boards have SRAMs or SDRAMs that might be utilized by this TimeTrial agent.

Large FPGA designs often utilize more than one clock domain. The performance monitor supports monitoring these designs by operating each tap monitor at the clock speed of the application being monitored. The rest of the performance monitor is operating at a nominal clock frequency appropriate for reporting results. The clock for the storage module is multiplexed back into the domain of the performance monitor for reporting. If the user wishes to measure signals from different clock domains within a single tap, either a simple double D flip-flop synchronizer is used for a single-bit signal or an asynchronous FIFO is provided for buses. The monitor operates at the fastest clock frequency of all the tapped signals' clocks.

4.4 Architecture of the Software Agent

The software TimeTrial agent is a separate process with which the application processes communicate using shared memory queues. Figure 4.4 shows an overview of the architecture of the software agent. When the software agent starts, it creates a queue in shared memory for each application process. The agent then polls from these queues in a round-robin fashion. When each application process starts, the application process places start up messages in the shared memory queue for each measurement at an edge originating from that process. These start up messages contain the metric types and aggregation functions to use as well as a unique index for the measurement. After sending the start up messages, the application process sends events to the agent associated with Auto-Pipe edges. These events contain the index of the measurement, a time stamp, and the event type.

Using the metric type and aggregation function, the software TimeTrial agent is able to derive the requested information about edge queue events. The metric type informs the software agent how to interpret the events. For example, the queue occupancy metric type tells the agent to sort enqueue and dequeue events by time to determine the queue occupancy after each event. For queue occupancy, the sort is necessary since the enqueue and dequeue events may come from different processes. However, once an enqueue event is

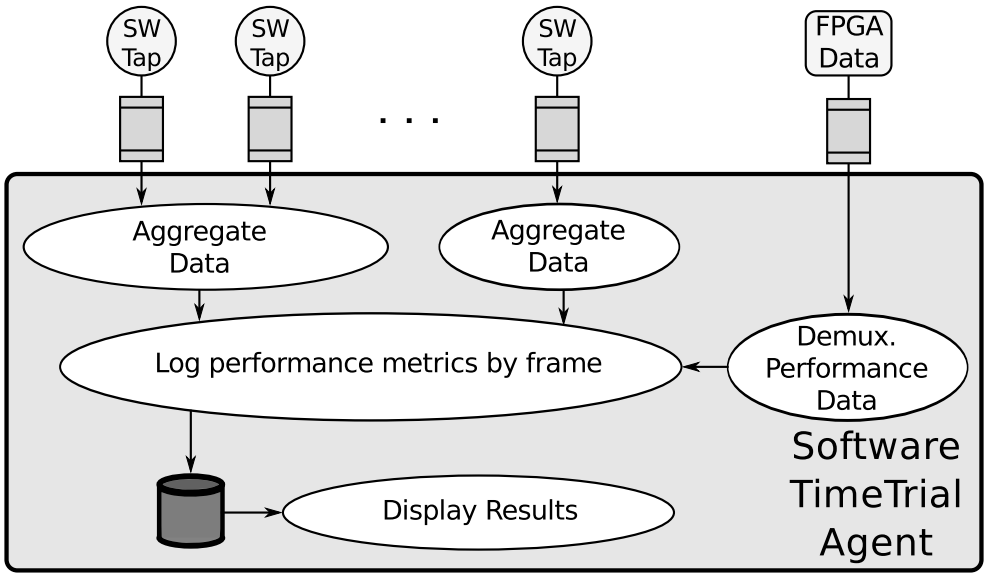


Figure 4.4: Overview of the software TimeTrial agent. In addition to aggregating event streams online, the software agent is responsible for logging results from the FPGA agents to disk.

matched with a dequeue event, the agent will discard the events keeping only the updated queue occupancy. Metric types which only tap one side of the queue (e.g., rates) do not need to sort incoming events.

The output of the metric function is then fed to the necessary aggregation functions. For example, if the "mean" aggregation function were requested for the occupancy of a queue, the agent would compute the mean of the stream of numbers returned from the metric function. The mean is tracked for each frame and reported at the end of the frame. The histogram aggregation function, on the other hand, uses the stream of numbers from the metric function as the bin number and accumulates time duration in the bin. For occupancy, this gives the amount of time that was spent at each queue occupancy. The histogram is reported at the end of a frame.

The software TimeTrial agent is also responsible for recording measurements from the FPGA agent. To do this, the process controlling the FPGA device communicates start messages to the software agent as is done for software queues. However, rather than communicating individual events as is done for software, the FPGA agent sends a message to the controlling process at the end of each frame. The controlling process then forwards the message to the software agent. Since the FPGA agent handles the data aggregation, the software agent simply records the data returned from the FPGA agent. The output of the software TimeTrial agent is a file containing the aggregated data. Once the software TimeTrial agent

has recorded the data to disk, the results can be read into the user's graphing software of choice.

Software taps are inserted at places in the application topology where a developer wishes to monitor performance events. As in the FPGA agent, taps can be inserted to measure general performance statistics such as FIFO enqueue/dequeue events, blocking probability, and/or communication throughput. Software taps are implemented as a standardized function call which generates a time stamp, event type, and relevant meta-data values that are used to calculate a specific performance metric. The function is written to be as lightweight as possible so as to lessen interference with the block where the tap originates. The output of a tap is written into a shared-memory FIFO, the other side of which is read by the software performance monitor agent.

4.5 Implementing the TimeTrial Language

The compiler has the task of interpreting TimeTrial Language statements, building the appropriate hooks into the underlying application (whether that be HDL or software) and instantiating the agents to monitor the performance accordingly. Given the complexity of the underlying architecture, this is a non-trivial task. Here we describe the mechanism that TimeTrial employs to translate from developer queries to a functioning runtime measurement system. Recall that each measurement statement is made up of three major semantic parts: statistic type, measure type, and target of the measurement.

The target of the measurement is any edge in the application. These edges may reside between two or more software blocks, two or more hardware blocks, or between hardware and software. In the case of a software to software edge, the compiler instruments the edge by generating function calls (i.e. taps) directly in the IPC channel that implements the edge. These function calls use an additional TimeTrial IPC channel to communicate events to the software agent. For fully FPGA-based edges, additional HDL signals are pulled from the data channel between blocks to monitor edge events. These signals are connected directly to the FPGA agent for processing into metrics. The number of taps and constitution of an event depends on the measurement type. Edges that cross architecture boundaries are discussed in Chapter 5.

The measurement type dictates what metric the agent is to compute during execution. The allowed measurement types generate one tap per edge with the exception of occupancy and latency. An `occupancy` measure taps the head and the tail of a queue and the agent,

whether software or hardware, computes the current state of the queue occupancy based on ingress and egress events. Currently, since only latency through a queue is implemented, the latency measure uses the same taps with a different computation. The remaining measure types, `rate`, `util`, `backpressure`, `value`, all look at one event type on the insertion side of the queue. The actual meaning of the event depends on the measure type and have subtle differences depending on whether the underlying edge is software or FPGA based. For instance, `util` tends to have a very well defined meaning for FPGA designs since the bandwidth of most buses is explicitly defined. However, in software, the capacity of an IPC channel may not be known as concretely. In this case we use an estimate of the upper bound for the channel throughput when calculating utilization. In contrast, `rate` has a very concrete definition for both software and FPGA portions of the application: amount of data transferred per frame. `latency` looks at insertion and removal events on an edge and calculates the time each element spent in the queue. For software, `backpressure` is the time an edge is unable to accept incoming data when there is data to be written. In the FPGA, this is the total number of cycles a channel sets a control signal high indicating it is unable to accept more data. `value` is simply each data value written into the queue. Note that without aggregation, many of these measurements would significantly impact the performance of the application being measured. The compiler also optimizes the measurements by not duplicating communication channels if multiple measure statements ask for the same tap. In this case only one tap is created and is read by multiple aggregation functions in the agent.

The statistic type indicates what to do with all the events for each measure type. The TimeTrial compiler generates the appropriate channels to communicate events to the various agents. For software, there is are additional control signals to indicate the type of measure for a set of taps and what statistic to perform. Each software measure statement gets its own aggregation function. The compiler also adds parameters into the tap instrumentation to further instruct the agents aggregation. For the FPGA portions, the compiler directly generates the parametrized FPGA agent HDL code tailored to the desired measurements. The FPGA agent is then synthesized at the top level with the rest of the FPGA design. Each measure statement gets its own tap monitor block as well as a aggregation block to perform online aggregation. Note that some combinations of statistics with measurement types have questionable semantic meaning. Consider the following TimeTrial statement:

```
measure sum util at B.out -> C.in
```

Since `util` is defined as a single-valued metric over a frame, the `min, max, mean, and sum` would all be identical and probably not what a developer would expect. Table 4.1 shows

the meaningful combinations of statistic types and measurement types. The measurements fall into two types: those that are single-valued within a frame and those that have multiple values. For the former, the optional statistic type is left from the statement and this defaults to trace.

Table 4.1: Compatibility between statistic types and measurement types in a TimeTrial language statement.

	min	max	mean	sum	hist	trace
rate	✗	✗	✗	✗	✗	✓
util	✗	✗	✗	✗	✗	✓
backpressure	✗	✗	✗	✗	✗	✓
occupancy	✓	✓	✓	✓	✓	✓
latency	✓	✓	✓	✓	✓	✓
value	✓	✓	✓	✓	✓	✓

4.6 Cross-platform issues: Timezones and Virtual Time

Given that the various computational resources in a heterogeneous system may not all run from a common clock, it is often required to transform time stamps recorded on distinct compute resources (known in TimeTrial as distinct timezones) into a common frame of reference, called *virtual time*. Using an appropriate system call, processor time (denoted t_P) is available to executing software with a known resolution. TimeTrial provides the FPGA time (denoted t_F) using a cycle counter in the reconfigurable logic design. TimeTrial's ability to convert both processor and FPGA time stamps into virtual time stamps enables reasoning about the performance of application events independently of the resource on which these events occur.

When an application is deployed on multiple computing resources, it is frequently the case that time is measured differently on one or more of the resources. A clear example of this is a processor core versus an FPGA. On a processor core, system calls return the current time at nanosecond resolution. On an FPGA, a counter can measure relative time with the resolution of the clock period, which is often application specific. Here, we describe our approach to resolving time across multiple distinct computing resources, each in their own time domain (which we call a timezone), in the TimeTrial performance monitor.

While the approach below generalizes to more than two computing resources, for brevity we will constrain the discussion to a particular circumstance containing only two resources,

a processor core and an FPGA. Using an appropriate system call, processor time (denoted t_P) is available to executing software with a known resolution. TimeTrial provides the FPGA time (denoted t_F) using a cycle counter in the reconfigurable logic design with the resolution of the FPGA clock period.

With knowledge of the relative rates of the two available timestamps, t_P and t_F, one can define a global timestamp (which we call virtual time and denote t_V) and relate the various resource timestamps to virtual time via a set of linear transformations, i.e.,

$$t_V = s_P \cdot t_P + b_P \tag{4.1}$$

and

$$t_V = s_F \cdot t_F + b_F. \tag{4.2}$$

where s_P and s_F encode the clock rate differences between the two platforms, and b_P and b_F represent the different time offsets. As an example, consider a processor clocked at 1 GHz, an FPGA clock running at 250 MHz (4 ns resolution), and a desired virtual time with 1 ns resolution. In this case, $s_P = 1$, $s_F = 4$, one of the offsets (say, b_P) can be set to an arbitrary value, and we need to discover (experimentally) the last remaining offset, b_F.

TimeTrial estimates the unknown offset b_F by performing a timezone calibration that records sets of three values: $t_{P,1}$, $t_{F,2}$, and $t_{P,3}$. Short messages are sent from the processor to the TimeTrial agent on the FPGA to retrieve the FPGA cycle counter. The value of $t_{P,1}$ is recorded immediately before sending the message by querying the processor cycle counter. Similarly, $t_{P,3}$ is recorded on the receipt of the response from the TimeTrial FPGA agent. The response message contains the FPGA cycle counter, $t_{F,2}$, as its payload. A typical calibration task sends one thousand of these messages and a suitable subset of those with minimum round trip times are used for calibration.

Given experimental values for $t_{P,1}$, $t_{F,2}$, and $t_{P,3}$ and the known values of s_P, s_F we can now reason about the value of b_F. Given causality, we know that

$$t_{V,1} < t_{V,2} < t_{V,3} \tag{4.3}$$

substituting for t_V,

$$s_P t_{P,1} + b_P < s_F t_{F,2} + b_F < s_P t_{P,3} + b_P \tag{4.4}$$

yields the following:

$$b_F > s_P t_{P,1} + b_P - s_F t_{F,2} \tag{4.5}$$

and

$$b_F < s_P t_{P,3} + b_P - s_F t_{F,2}. \tag{4.6}$$

These are the bounds on the true value of b_F. Setting b_P to zero, this further simplifies to:

$$b_F > s_p t_{P,1} - s_F t_{F,2} \tag{4.7}$$

and

$$b_F < s_p t_{P,3} - s_F t_{F,2}. \tag{4.8}$$

It is convenient to assume that the virtual timestamp $t_{V,2}$ is midway between $t_{V,1}$ and $t_{V,3}$ (i.e., this makes the assumption that the communication between the processor and FPGA is symmetric in delay). Under this assumption, $t_{V,2}$ is the midpoint between $t_{V,1}$ and $t_{V,3}$:

$$t_{V,2} = (t_{V,1} + t_{V,3})/2 \tag{4.9}$$

and b_F can be computed from (2):

$$b_F = t_{V,2} - s_F t_{F,2}. \tag{4.10}$$

In practice, our knowledge of s_F is not perfect, so a least mean squares curve fit is performed on the set of calibration points to estimate both s_F and b_F.

Given the above, TimeTrial is able to convert both processor timestamps and FPGA timestamps into virtual timestamps. This enables reasoning about performance of an application as a whole independent of the resource on which individual events occur.

4.7 TimeTrial Performance: Overhead and Impact

In this dissertation, we define overhead as the additional resources that were utilized by the TimeTrial system to profile the performance. We define impact as the change in runtime or throughput that is caused by adding TimeTrial instrumentation. This is the most important metric of a profiler since it measures the change in application performance due to monitoring. Overhead gives the developer insight into the extra resources that are necessary to measure in a low-impact manner. In this section we assess the performance limits of both the FPGA agent and the software agent.

4.7.1 Performance of the FPGA Agent

The performance of the monitoring circuit can be characterized by the maximum operating frequency of the circuit, the fraction of the total execution that can be monitored, as well as the resource overhead required to implement it. To determine the operating frequency of the performance monitor, the monitor circuit was instantiated on a simple adder block as the user application. The number and type of monitoring operations was varied to quantify the effects on the operating frequency. The circuit was synthesized using Synplify Premier DP v9.6.2 and placed and routed using Xilinx ISE v10.1i, targeting a Virtex 4 LX100 FPGA. Table 4.2 shows the results. The maximum operating frequency stays almost constant throughout the range implying that the monitor is scalable. We anticipate that these numbers will be hard to achieve on FPGA designs that utilize near 100% of the chip. In these cases manual placement might be necessary if high clock frequencies are required. Currently, the critical path is limited by an addition to update the value in the histogram table. If 18 bits are used instead of the current 36, a single queue monitor can run faster than 330 MHz.

Table 4.2: Maximum achievable clock frequencies (rounded to the nearest integer) for three configurations of the performance monitor. The results are shown for a Xilinx Virtex 4 LX100 speed grade 12 FPGA.

Configuration	f_{max} (MHz)
1: 1 queue monitor	256
2: 4 queue monitors	255
3: 8 queue monitors and 16 averagers	252

The FPGA agent is capable of 100% duty cycle operation, able to capture performance information across the entire execution of a program. In addition, the performance monitor is capable of monitoring while reporting results from the previous frame.

In order to assess the overhead of utilizing the FPGA agent, several designs were built using the same monitoring set up as in Table 4.2. Table 4.3 shows the available resources required to implement the different monitor configurations. The resource utilization increases mostly linearly as more monitoring components are added. Even in the most aggressive case (configuration #3), our total resource utilization is less than 10% in all categories which we deem acceptable for development purposes. Also note that Block RAM usage can be traded for LUTs if a design is resource constrained in that category.

Table 4.3: Performance monitor resource overhead for the same three configurations in Table 4.2. Numbers in parenthesis below the resource type show the total number of each resource available on the LX100 FPGA.

Config. #	LUTs (98,304)	FFs (98,304)	RAMB16s (240)
1	1655 (1.7%)	1260 (1.3%)	2 (0.83%)
2	3732 (3.8%)	2466 (2.5%)	8 (3.3%)
3	8584 (8.7%)	4776 (4.9%)	16 (6.6%)

4.7.2 Performance of the Software Agent

To assess the overhead and impact that the software agent has on the software portion of an application, a micro-benchmark was developed in the form of a chained Auto-Pipe application. In this benchmark application, the `Src` block generates data, forwards it through a chain of 9 blocks, and the `Sink` block discards the data. The purpose of this application is to benchmark TimeTrial's ability to measure multiple edges that communicate very rapidly. Figure 4.5 shows the topology of the application. Each block gets mapped to a single processor core. The application uses 11 cores of a 12-core AMD Opteron machine. The 12th core is reserved for the TimeTrial SW agent, shown as `TTA` in the diagram. Each edge is tapped (shown as dashed lines in Figure 4.5) and two different measurements were calculated on all 10 taps.

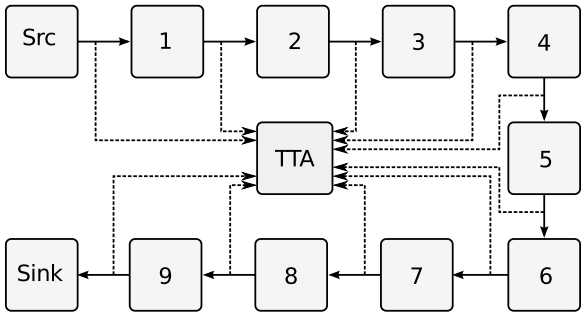


Figure 4.5: Software agent micro-benchmark application. The source generates data and each block consumes and forwards the data as fast as it is able. Each block has its affinity set to a unique processor core.

To measure both the impact and the overhead, the micro-benchmark was run for 30 iterations with two different transfer sizes. Each transfer is a bulk transfer of either a 2048 element 8-byte array or an 8192 element 8-byte array. These sizes were chosen since they are efficient array sizes for transferring data in the Auto-Pipe system. The frame size was set to 1 second. Four experiments were performed for each. The first was a baseline with

no instrumentation added. The second measured the mean rate on each edge. The third calculated a histogram of the queue occupancy on each edge. The final experiment calculates both the rate and the occupancy (20 total measurements). Figure 4.6 shows the overhead measurement results with respect to utilization of the software TimeTrial agent. For the array size of 2048, just measuring the rate is a relatively low burden on the agent. Gathering a histogram of queue occupancy is a much more compute intensive task. Both measurements push the utilization up around 70%. For an array size of 8192, the utilization is much lower, topping out around 20%. We consider these acceptable overheads for detailed measurements.

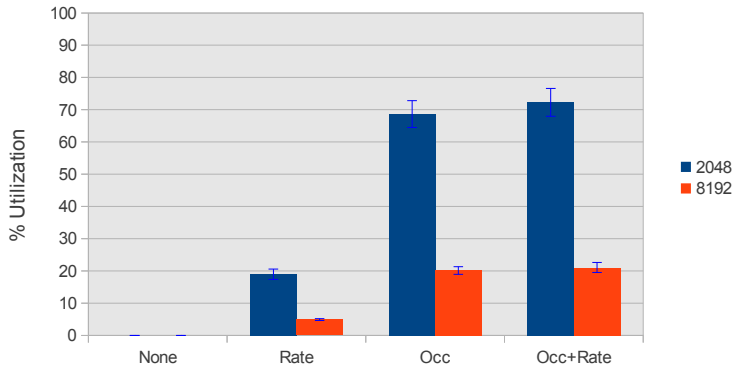


Figure 4.6: Overhead of the measurements for the software agent measured by utilization of one processor core for two array transfer sizes. Error bars show one standard deviation.

To measure the impact, the same set of experiments were run and the throughput was logged for both transfer sizes. The results are shown in Figure 4.7. For a transfer size of 8192, there is almost no impact on the application performance. The smaller transfer size does have some impact, a maximum 3.7% reduction in throughput when measuring both rates and occupancy of all 10 edges in the application.

4.8 Chapter Summary

In this chapter we described ways that TimeTrial is able to measure the performance of a distributed application with low impact. This is accomplished by deploying agents on each resource to measure the application and aggregation of event streams to performance metrics online.

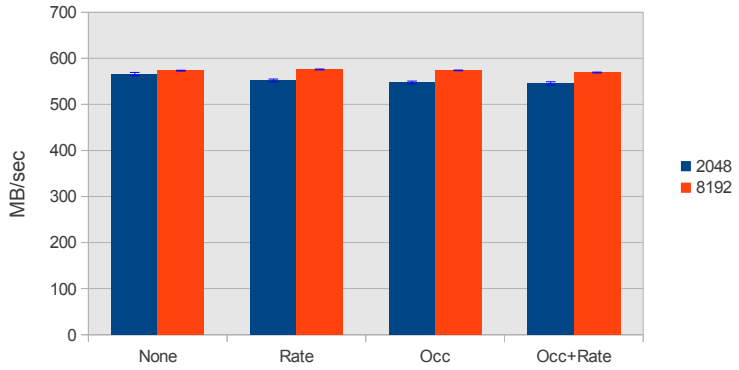


Figure 4.7: Impact on the throughput of the micro-benchmark by the software agent. Error bars show one standard deviation.

Taps were described as a way to gather information about the performance. The TimeTrial language was integrated with the Auto-Pipe compiler to enable automated insertion of taps and connection of these taps to the measurements agents. The software agent is responsible for collecting results from all the FPGA agents and combining them into a profile per frame.

Both the software and FPGA agents are able to measure substantial applications with very little impact on the execution. Dedicating resources is essential the measurement task.

Chapter 5

Monitoring Virtual Queues

In this chapter, we describe TimeTrial's approach to measuring virtual queues, while attempting to preserve the low-impact nature of the performance monitoring. This approach involves using a simple discrete event simulation model for measurements on a single resource. Then, a more sophisticated model is presented that handles crossing a system I/O bus. Next, we show example measurements of virtual queues using our approach and compare them to ground truth (precise knowledge of the quantity being measured) collected from a micro-benchmark application. Finally, we discuss the circumstances where the approach described here is inappropriate, and how users of TimeTrial might understand when the approach is or is not applicable.

The previous chapter described the approach to automated monitoring of regular queues, that is communication channels that begin and end within a single compute resource. More challenges arise when the communication channel crosses resource boundaries, such as the link from an FPGA to the a processor. In this case, a portion of the queue is on the FPGA, a portion of the queue is comprised of buffers associated with whatever low-level communication mechanism(s) are used to move data between the FPGA and the processor (e.g., a PCIe bus or something similar), and a portion of the queue is in the processor's memory.

We refer to the queues on such edges as *virtual queues*. Measuring the occupancy of virtual queues is not simply a matter of instrumenting the enqueue and dequeue operations. To perform the aggregation function(s), both enqueue and dequeue *events* must be known on a common platform, which necessitates moving either the enqueue events from the FPGA to the processor or the dequeue events from the processor to the FPGA. Neither of these options is attractive, since a large number of events implies a large volume of performance meta-data must share the processor-FPGA interconnect (such as the PCIe bus example above).

As stated above, the direct measurement of communication channels (and their associated queues) that cross platform boundaries is incompatible with the notion of low-impact monitoring. While some metrics of interest, such as rate, can be effectively measured at one end of the channel or the other, other metrics, such as queue occupancy, require information from both the head and the tail of the queue.

TimeTrial's approach to virtual queue monitoring is to instrument what it can and use a performance model of the underlying system to infer what it cannot directly measure, estimating the information that is missing. As such, it is important not only to provide the performance quantities that were estimated, but also to provide guidance as to the quality of the estimates.

Here we focus on querying the occupancy of virtual queues. It is anticipated that the same techniques will be similarly effective for other metrics that require detailed event information across platform boundaries (e.g., latency through virtual queues); however, this is left for confirmation in future work. It is clearly true for some aggregated metrics such as mean latency, which is directly related to mean occupancy via Little's Law[74].

As defined here, virtual queues are comprised of several constituent components, all chained together to comprise the communications channel between two compute blocks. For a channel that moves data between a multicore processor and an FPGA, the components will include buffers in user space on the processor, kernel buffers on the processor, a physical bus that transfers the data to the FPGA (e.g., PCIe), buffers in the DMA engine on the FPGA card, and application buffers deployed (by the Auto-Pipe runtime system) on the FPGA. While many of these constituent components of the communications channel are opaque (i.e., they cannot be directly monitored by TimeTrial), the two components at either end of the chain (comprising the head of the queue and the tail of the queue) are visible to TimeTrial and can be therefore be monitored. Whenever a user requests the occupancy of a virtual queue, TimeTrial deploys monitors for the head and tail sub-queues for which it has visibility.

5.1 Approach to Virtual Queue Occupancy

To measure the occupancy of a queue over time, one can record a series of insertion and removal time stamps at the head and tail of a queue, respectively. The queue can then be "replayed" using a simple simulation based on this event trace. Practically, this approach is useful for queues that have small volumes of data moving through them. However, the

amount of trace data that needs to be stored quickly becomes burdensome as the data volume approaches TimeTrial's ability to store the event trace. This problem becomes particularly acute on FPGA platforms, since there is very limited buffering on the FPGA to store events.

For regular queues (i.e., those that reside entirely on a single compute resource), TimeTrial monitors the insertion and removal events and aggregates these to a histogram or mean value of queue occupancy on-line to lower the impact of monitoring. For virtual queues, access to the insertion or removal event trace is impractical since they must be communicated across platform boundaries. This would likely overwhelm the communication bus between these resources for many applications causing unwanted performance impact.

Instead, TimeTrial recreates the queue occupancy through an empirically-driven, stochastic, discrete-event simulation [39]. TimeTrial measures the time between each insertion into the queue and removal from the queue. Both streams of delta time values are then aggregated on-line into an inter-insertion time histogram and an inter-removal time histogram. These histograms are then used to model an arrival process and a departure process from the virtual queue. By sampling from the histograms, a stochastic discrete-event simulation is performed which samples from these distributions to "replay" the virtual queues over time.

TimeTrial performs aggregations over a frame. This produces one set of inter-transfer time histograms per frame. Having more than one histogram allows for the stochastic simulation to be non-stationary, that is, the shape of the distributions change over time. Hence, the measurements enable the stochastic simulation to reflect changes in the application performance over the course of its execution.

There are two modeling assumptions that are present in the stochastic simulation that warrant consideration. First, when replaying the virtual queue dynamics the stochastic simulator is assuming that the virtual queue acts as an ideal queue (e.g., insertions at some time t are immediately available for removal at time t). This assumption presumes there is no inherent latency present in the underlying implementation of the communications channel.

Second, the stochastic simulation makes the assumption that the insertion and removal processes are independent and identically distributed (iid). Significant auto-correlation in either process, or cross-correlation between the insertion and removal processes are another potential source for error.

If one or more of these modeling assumptions is untrue for the application being executed, there is a reasonable chance that the stochastic simulation predictions for the occupancy of the virtual queue will diverge from the measured occupancy of the sub-queues at the head and the tail of the virtual queue. This suggests an approach for performing a sanity check on the model's predictions. If the predicted queue occupancies do not align with the measured occupancies of the visible sub-queues, the model is not to be trusted. Note that if the model predictions and the measured sub-queue results do match, that is no proof that the model predictions are correct across the board. When there is not a match, it is clear that the models predictions cannot be trusted. However, TimeTrial can sill provide useful information about the performance of the virtual queue. When the model fails, there is still direct measurement results for the individual sub-queues on either end of the virtual queue.

In summary, the procedure for responding to a query that asks for the occupancy of a virtual queue is as follows:

1. Instrument the ingress point (tail) of the queue and its associated sub-queue (that is visible to TimeTrial) on the source computational resource and the egress point (head) of the queue and its associated sub-queue (that is visible) on the destination computational resource.

2. Execute the application, recording a histogram of the inter-insertion times at the tail of the queue and a histogram of the inter-departure times at the head of the queue. Also record occupancy histograms of the sub-queues at the head and tail.

3. Using the inter-insertion time and inter-departure time histograms to drive the pseudo-random number generator distribution, perform a stochastic discrete-event simulation, predicting the occupancy of the entire virtual queue.

4. Compare the stochastic simulation predictions with the measured sub-queue occupancy distributions.

5. If the comparison of step 4 succeeds, report the occupancy of the virtual queue to the user.

6. If the comparison of step 4 fails, one or more of the stochastic simulation model assumptions are not true for the application, and the stochastic simulation's predictions should not be trusted. In this case, rely on sub-queue measurements alone for performance debugging.

5.2 Assessment of Modeling Technique

We assess the above described approach to understanding the dynamics of a virtual queue through the use of several micro-benchmark applications. These applications are designed to either: (1) have a known queue occupancy by construction; or (2) effectively record the queue occupancy during execution. Either way, we know ground truth and can therefore effectively assess the model.

The family of micro-benchmarks that are used are designed to emulate the queue activity in a traditional queue-server combination (i.e., queueing station) from classic queueing theory [62]. Figure 5.1 illustrates a single-stage queueing station that includes an arrival process from the left (that delivers "jobs" into the system with mean arrival rate λ), a FIFO queue, and a service process on the right (that services "jobs" from the queue with mean service rate μ).

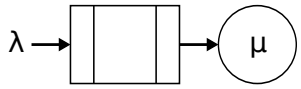


Figure 5.1: Single-stage queueing station.

For our purposes, a "job" will be represented by a single data element (8 bytes in size), and the distribution of the arrival process and the service process will either be Markovian (i.e., exponential inter-arrival and/or service times) or deterministic (i.e., constant inter-arrival and/or service times).

The first micro-benchmark emulates the queue activity in a classic $M/M/1$ queueing station [62]. In this notation, the first M denotes a Markovian arrival process, the second M denotes a Markovian service process, and the 1 denotes a single server. Figure 5.2 shows the Auto-Pipe micro-benchmark application, where the `source` block is acting as the arrival process (with pseudo-random numbers drawn from an exponential distribution, mean $1/\lambda$, provided by the left-most `PRNG` block). The `server` block is acting as the server (with pseudo-random numbers drawn from an exponential distribution, mean $1/\mu$, provided by the second `PRNG` block). The queue shown on the edge between `source` and `server` is the queue of Figure 5.1. Note that there do exist queues on the other application edges as well, they are simply not drawn in the figure.

To enable playback of true queue occupancy, the data value delivered from `source` to `server` contains the inter-insertion time between two insertion events on the queue. The data values delivered from the `server` block to the `record` block include both the above inter-insertion

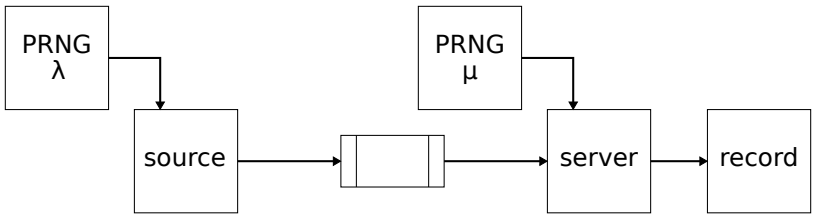


Figure 5.2: Micro-benchmark application that mimics queue activity of the single-stage queueing station.

time and the inter-dequeue time experienced at the `server` block. This stream of data enables us to recreate the actual dynamics of the queue via a trace-driven simulation after the run has completed. This trace-driven simulation provides our best understanding of ground truth (i.e., what really happened in the queue).

Due to potential confusion that might result from our use of two different simulations, each for a distinct purpose, we will use the following terms in the discussion below.

- *Stochastic simulation* refers to the simulation model introduced in Section 5.1 that forms the basis for TimeTrial's predictions of queue occupancy. This simulation will be utilized each time a user requests TimeTrial to measure the occupancy of a virtual queue.
- *Trace-driven simulation* refers to the simulation introduced in the previous paragraph that uses trace information recorded by a micro-benchmark application which represents ground truth for the ingress and egress times of the virtual queue. This simulation is only available if the application records these time traces.

The $M/M/1$ micro-benchmark was deployed on 2 processors (the `source` and its random number generator on one processor, the `server` and its random number generator on the other) and the queue in question was instrumented using TimeTrial, asking for a histogram of the queue occupancy. The utilization of the queueing server for this run, $\rho = \lambda/\mu$, was set to 0.9. Figure 5.3 shows two versions of the queue occupancy in this micro-benchmark execution: (1) the trace-driven simulation result from the micro-benchmark output; and (2) the measured value from TimeTrial. The obvious alignment of these results simply indicates that the micro-benchmark is doing what we expect it to do and TimeTrial is readily capable of instrumenting queues that reside entirely on one compute resource (in this case a multicore processor).

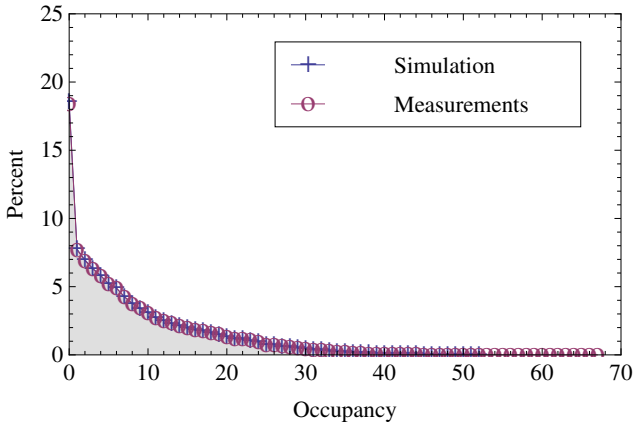


Figure 5.3: Queue occupancy in $M/M/1$ micro-benchmark deployed in software.

The above micro-benchmark was redeployed on an FPGA (all blocks except `record`) and the experiment repeated (this time with $\rho = \lambda/\mu = 0.8$) giving the results in Figure 5.4. Again, the trace-driven simulation results and the measured results from TimeTrial and in close agreement.

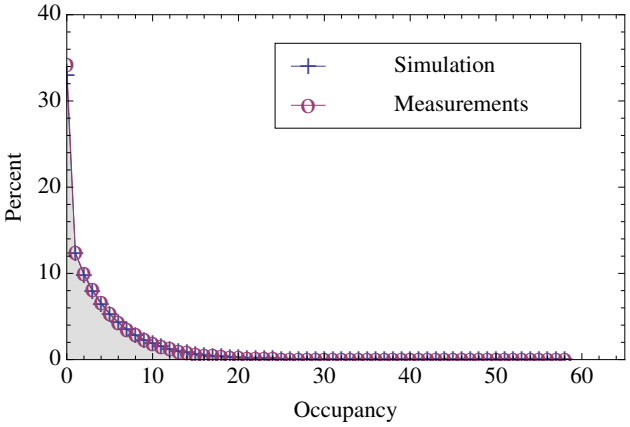


Figure 5.4: Queue occupancy in $M/M/1$ micro-benchmark deployed in hardware.

Next we will illustrate the approach using an alternative mapping, one that includes a virtual queue. The `server` block is mapped to an FPGA and the remaining blocks are all mapped to processor cores. Figure 5.5 shows the queue occupancy of the virtual queue that crosses the processor-to-FPGA boundary based upon the trace-driven simulation that represents ground truth. Figure 5.6 shows TimeTrial's estimate of the queue occupancy based on the stochastic simulation described above. It is obvious by comparison with Figure 5.5 that the actual queue occupancy is significantly different from TimeTrial's estimate. Clearly, TimeTrial needs a better model.

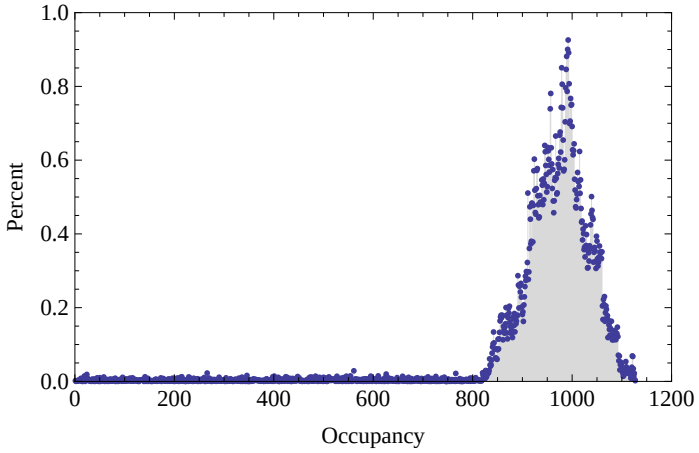


Figure 5.5: True queue occupancy in $M/M/1$ micro-benchmark with software-to-hardware virtual queue.

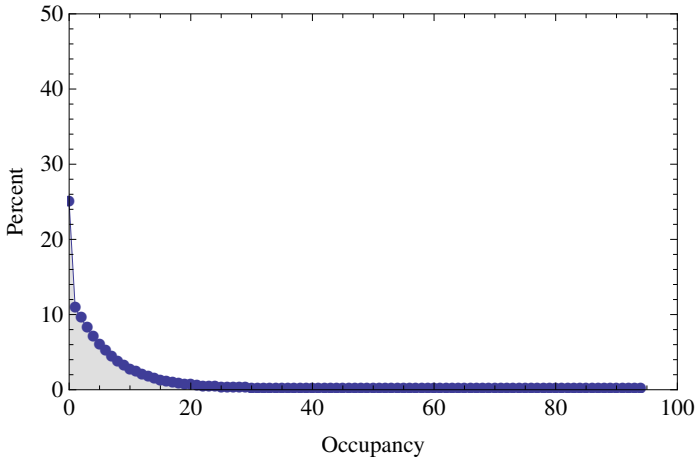


Figure 5.6: Modeled queue occupancy in $M/M/1$ micro-benchmark with software-to-hardware virtual queue.

5.3 Modeling Virtual Queue Occupancy

Guiding the consideration of a more robust model are the two modeling assumptions described in Section 5.1: (1) the existence of an ideal queue, and (2) iid insertion and removal processes. Since the chosen micro-benchmark matches the second assumption (i.e., its inter-arrival time distribution and its service distribution are both iid), we next investigate a more robust model of the underlying communications channel that implements the virtual queue.

The physical system that we are using consists of a two-socket motherboard that contains a pair of AMD multicore chips and a Xilinx Virtex-4 FPGA on a PCI-X bus. If we treat the bus as a "server," in the sense of a queueing model, we can extend the model of the communications channel into the two-stage tandem queueing network illustrated in Figure 5.7. Here, the first server (with constant service rate μ_1 and a deterministic service time distribution) models the bus and the second server (with mean service rate μ_2) models the downstream compute block. The stochastic simulation can then "replay" the activity in the virtual queue by simulating the tandem queueing network.

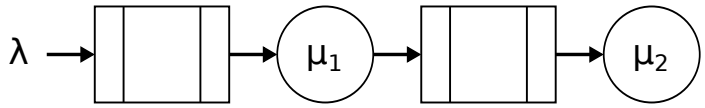


Figure 5.7: Two-stage tandem queue.

In this system, data to be moved across the PCI-X bus is buffered (by the low-level system software) until a minimum bulk transfer size is reached, at which point a bulk DMA transfer moves the data across. This property of the communication channel can be readily reflected in the tandem queueing network model by ensuring that the first server (representing the PCI-X bus) performs bulk transfers. In the stochastic simulation model, it allows data to buffer in the upstream queue until `BulkSize` data elements are queued, at which point it "services" them all together, removing them from the upstream queue and inserting them into the downstream queue. In this model, the instantaneous occupancy of the virtual queue consists of the occupancy of the upstream queue of the bus server plus the occupancy of the downstream queue of the bus server plus the number in service in the bus.

Returning to the $M/M/1$ micro-benchmark whose true virtual queue occupancy histogram is shown in Figure 5.5, the queue occupancy histogram that TimeTrial predicts using the stochastic simulation based on the tandem-queue model is shown in Figure 5.8. These two figures are in reasonable agreement with one another. In addition, the sub-queue histograms directly measured by TimeTrial of the head and tail of the virtual queue (not shown) are also in agreement with Figures 5.5 and 5.8.

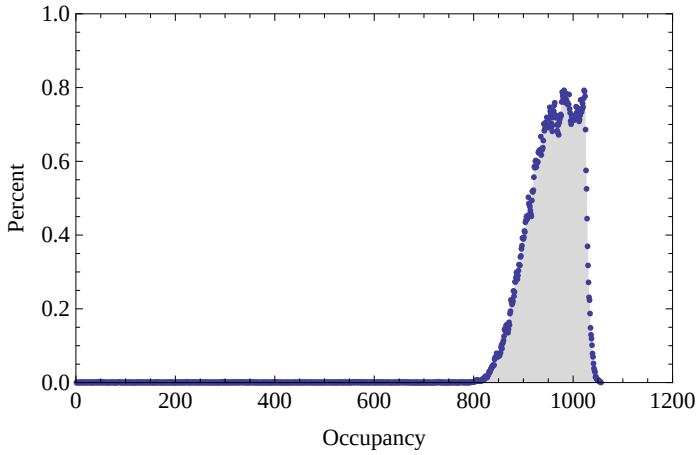


Figure 5.8: Modeled queue occupancy in $M/M/1$ micro-benchmark with software-to-hardware virtual queue, using the tandem queue model.

We next assess the appropriateness of the tandem-queue model on a few additional micro-benchmarks. Figure 5.9 shows the results of a trace-driven simulation giving true queue occupancy for an $M/M/1$ micro-benchmark with the mapping reversed to give a hardware-to-software virtual queue. (Here, `source` and its associated random number generator are mapped to the FPGA.) Figure 5.10 shows the results of the stochastic simulation using the tandem-queue model. As with the software-to-hardware virtual queue, there is reasonable agreement between the modeled queue occupancy and the true queue occupancy.

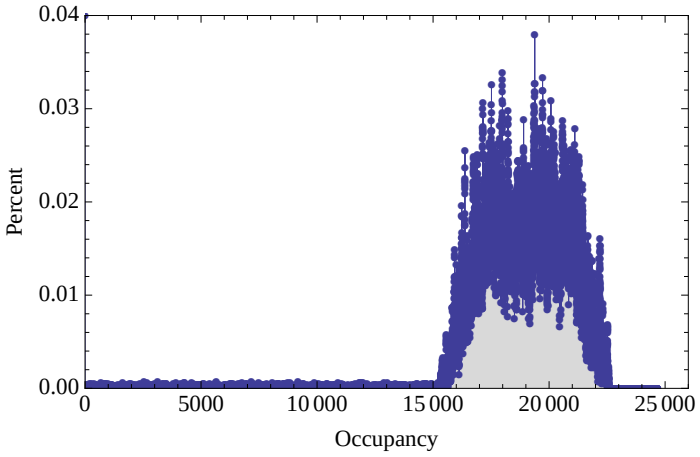


Figure 5.9: True queue occupancy in $M/M/1$ micro-benchmark with hardware-to-software virtual queue.

Here we explore the sensitivity of the model to the distribution of the service process. In this micro-benchmark, the `server` block is set to perform dequeue operations at a constant rate. This micro-benchmark corresponds to an $M/D/1$ queueing station, where the arrival process

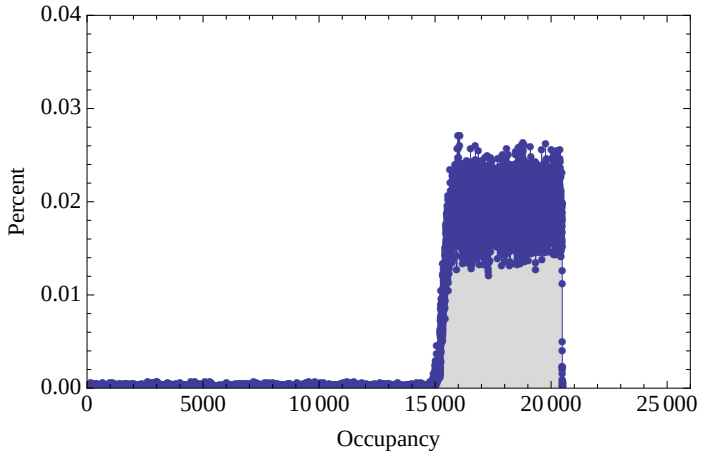


Figure 5.10: Modeled queue occupancy in $M/M/1$ micro-benchmark with hardware-to-software virtual queue.

is Markovian and the service process is deterministic. Figure 5.11 presents the trace-driven simulation results that indicate true queue occupancy for a software-to-hardware virtual queue for this micro-benchmark (i.e., `source` is mapped to a processor core and `server` is mapped to an FPGA). Figure 5.12 presents the stochastic simulation model of the same micro-benchmark.

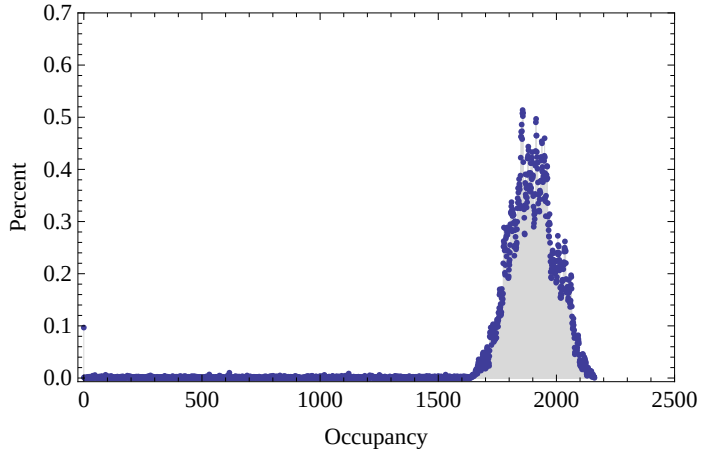


Figure 5.11: True queue occupancy in $M/D/1$ micro-benchmark with software-to-hardware virtual queue.

As is shown in the figures, the shape of the virtual queue occupancy is well represented in the stochastic simulation model. As with the previous micro-benchmarks, the occupancy is strongly influenced by the bulk transfer characteristics of the bus, so much so that it does not even qualitatively resemble the theoretically expected queue occupancy for an $M/D/1$ queueing station at all.

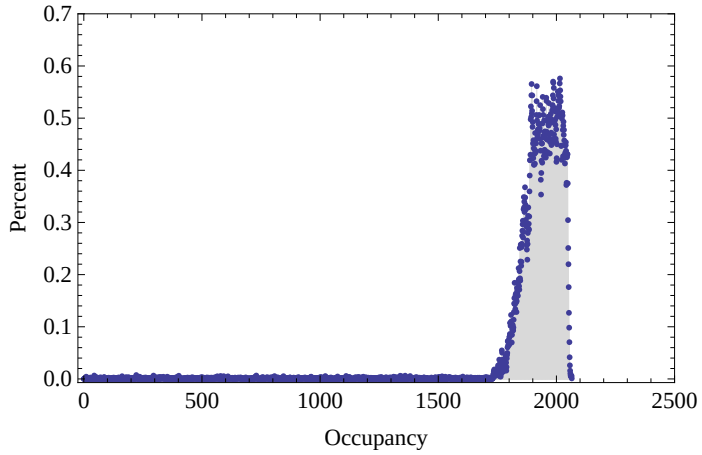


Figure 5.12: Modeled queue occupancy in $M/D/1$ micro-benchmark with software-to-hardware virtual queue.

In each of the above micro-benchmark examples, the stochastic simulation model does a reasonably good job of indicating the general properties of the occupancy of the virtual queue. However, there are instances where the stochastic simulation model will fail to accurately represent the true occupancy of the virtual queue. An example of this is illustrated in Figures 5.13 to 5.15, which show the predicted queue occupancy from the stochastic simulation (Figure 5.13), the measured sub-queue occupancy from the hardware side (Figure 5.14), and the measured sub-queue occupancy from the software side (Figure 5.15) for a micro-benchmark with a hardware-to-software virtual queue. In this case, the source random number stream had significant auto-correlation (i.e., the source was not iid).

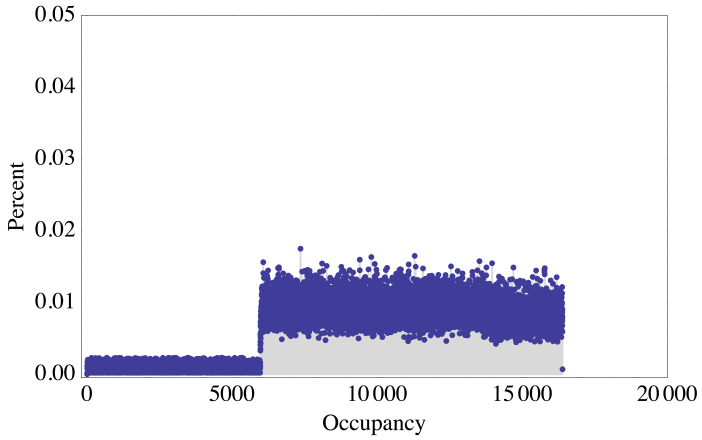


Figure 5.13: Modeled queue occupancy in correlated micro-benchmark with hardware-to-software virtual queue.

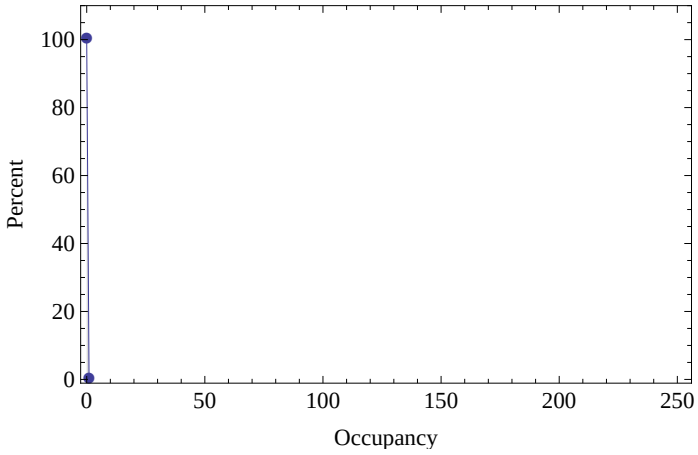


Figure 5.14: Hardware sub-queue occupancy in correlated micro-benchmark with hardware-to-software virtual queue.

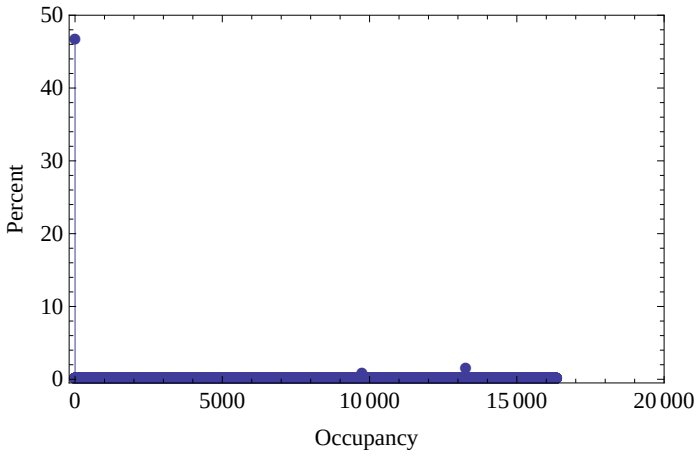


Figure 5.15: Software sub-queue occupancy in correlated micro-benchmark with hardware-to-software virtual queue.

It is clear from the figures that the sub-queue occupancies measured at the head and the tail of the virtual queue do not agree with the overall queue occupancy predicted by the stochastic simulation model. While for this particular instance the reason for the lack of agreement is well understood (a non-iid source), in general the use of the head and tail sub-queue occupancies serves as a check on the appropriateness of the stochastic simulation model, but does not necessarily give a reason when there isn't agreement.

5.4 Chapter Summary

This chapter has presented TimeTrial's approach to monitoring virtual queues, those queues that cross the boundaries of the computing platforms executing the application. For those measurements that require data collection at both the head and the tail of the queue, histograms of inter-insertion times and inter-departure times are collected at each end of the queue, and a stochastic discrete-event simulation model is used to recreate the dynamics of the virtual queue in question.

As a partial verification of the appropriateness of the stochastic simulation model, the portions of the virtual queue that are visible to TimeTrial are also monitored, and if the stochastic simulation results are not consistent with the measurements of the sub-queues at the head and the tail of the virtual queue, the stochastic simulation model is deemed incorrect and is discarded. In this way, TimeTrial works to not only model the activity within the virtual queue, but also helps to verify whether or not its internal model is appropriate.

Chapter 6

Measuring Performance with TimeTrial

This chapter demonstrates the use of TimeTrial to profile two streaming, heterogeneous applications. We start with an application that solves Laplace's equation using a Monte Carlo approach. Several different mappings are explored and TimeTrial is able to show the bottlenecks in each. Next, a streaming implementation of Basic Local Alignment Search Tool (BLAST) is profiled. TimeTrial provides guidance on queue provisioning, bottleneck locations, queue occupancy and the overhead of deadlock avoidance techniques. For both applications, TimeTrial provides illuminating feedback that would have been painstaking to gather otherwise.

6.1 Monte Carlo Solution to Laplace's Equation

Here we will illustrate the use of TimeTrial with an example application, a Monte Carlo solver for Laplace's equation. Laplace's equation is a second-order partial differential equation (PDE) [102] that has several uses, including modeling stationary diffusion (such as heat) and Brownian motion. For heat, given the temperature at the boundaries of an object, solutions to Laplace's equation provide the interior temperatures at equilibrium.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

The ease of solving Laplace's equation depends on the nature of the boundary conditions. An analytic solution exists for simple boundary conditions, however, for many boundary conditions, no analytic solution exists and numerical solutions are needed [37]. There are several approaches for numerical solutions. One approach is Gauss-Seidel iterations [102].

This method converges quickly, but it requires that the complete grid be stored in memory. Another method is Monte Carlo simulation. This technique is provably correct [92], but converges slowly. Nevertheless, this method is useful if a small number of interior points are needed. This is because the Monte Carlo method does not require storing the entire grid, since the grid is implicit, and only those points that are of interest need be computed.

Figure 6.1 shows an application topology of an Auto-Pipe implementation of a Monte Carlo solver for Laplace's equation. The labels within the blocks indicate the block function, and the labels above identify the individual blocks. Edge labels are also shown in the figure for reference below.

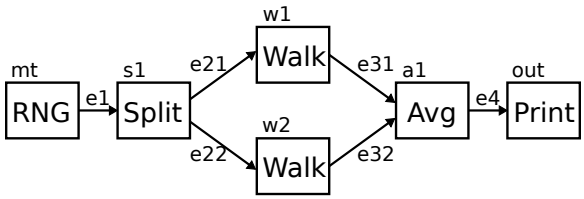


Figure 6.1: Topology of Monte Carlo solver for Laplace's equation using two independent walk blocks. Block instances names are given above each block, edge labels above each edge.

The application works as follows. Pseudo-random numbers are generated (using the Mersenne twister algorithm [76]) in block `mt`. Block `s1` splits the stream of pseudo-random numbers for use by two copies of a `Walk` block (called `w1` and `w2`) that perform a random walk executing the Monte Carlo solution [92]. Results from blocks `w1` and `w2` are combined in the `Avg` block `a1` and written to disk in block `out`. Greater or fewer `Walk` blocks are straightforwardly deployed with larger or smaller `Split` and `Avg` trees (e.g., Figure 6.2 illustrates 4 `Walk` blocks).

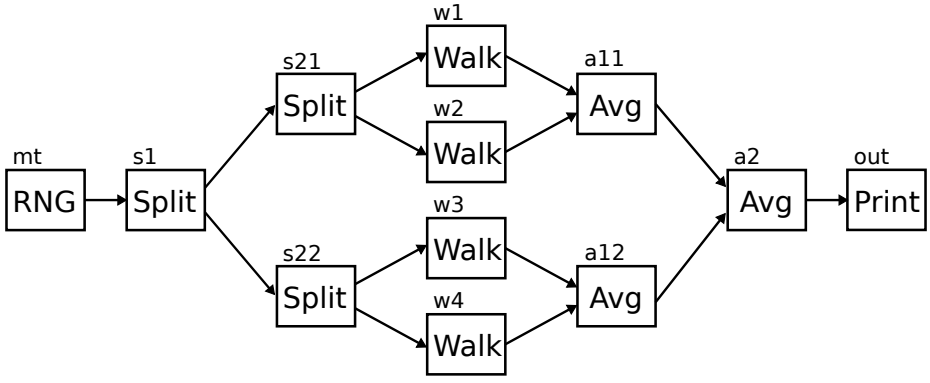


Figure 6.2: Topology using four independent random walks. Edges are labeled using a similar naming scheme as in Figure 6.1 but not shown for clarity.

For the example executions used here, a 2-D grid was fixed at size 100×100 and the boundary conditions were set to a square containing the grid. The temperature at the boundary was set to 0 for three sides (top, right, and bottom) and 100 for the fourth side (left). One thousand random walks were performed at each grid coordinate, evenly divided across the `Walk` blocks. The output of the application gives a 100×100 grid of temperatures. A plot of the output using colors to represent temperatures (blue being 0 and red being 100) yields the image of Figure 6.3.

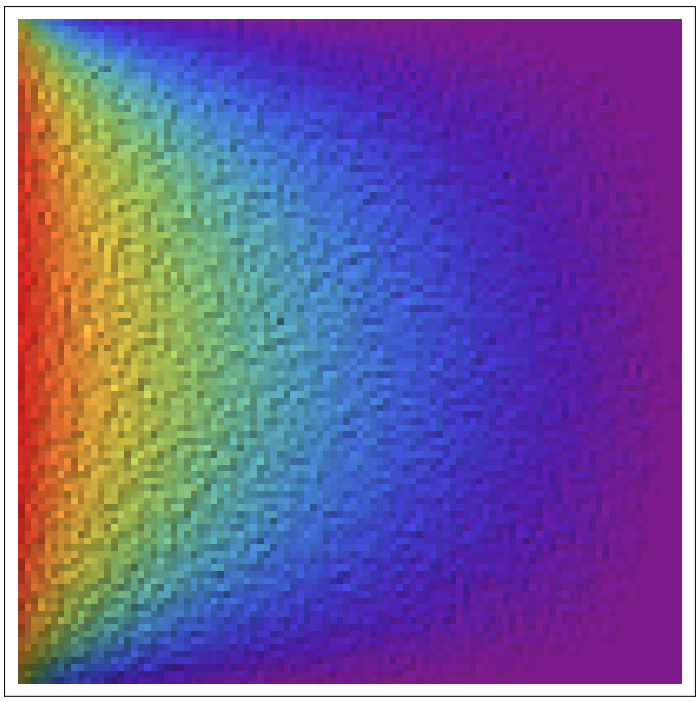


Figure 6.3: Output from example 2-D temperature surface.

We now reconsider the performance questions posed in the introduction. This question and the ones that follow are helpful for comparing the performance a developer expects given their knowledge of the design with reality and illustrate a way a user might query the performance using TimeTrial. The first question was:

> *At what rate is data moving across the link that connects the* ***`RNG`*** *block to the* ***`Split`*** *block?*

TimeTrial can answer this question via a straightforward `measure` statement:

```
measure rate at e1;
```

which generates the output shown in Figure 6.4. The figure provides a box-and-whisker plot of the data transfer rate for each frame (the frame period is 1 second for all of the example runs in this section). The median bar in the box is labeled on the graph (here, 2.4 Mtransfers/s), the 1st and 3rd quartiles are the top and bottom of the box, the whiskers indicate minimum to maximum (excluding outliers), and any outliers (defined as beyond 1.5× the inter-quartile range) are indicated by points on the graph.

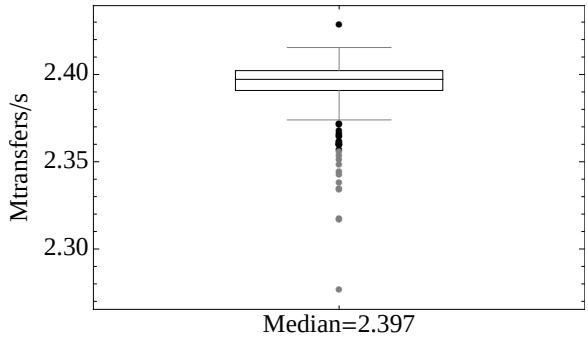


Figure 6.4: Box-Whisker plot of mean data rate per frame across edge `e1`.

The second question posed in the introduction was:

> *Are the data rates balanced between the upper and lower branches of the application topology?*

To answer this question, we ask for the data rates into the two `Walk` blocks:

```
measure rate at e21;
measure rate at e22;
```

These queries generate output of the same form as Figure 6.4, from which we can confirm that the data rates are reasonably balanced at 1.2 Mtransfers/s. (Note that to conserve space we will not reproduce the graphs generated by all of the example measure statements that are described.)

The third and fourth questions:

> *What is the occupancy of the queues between each block?*

What fraction of the time is backpressure being asserted from the `Print` *block to the* `Avg` *block?*

are both directly supported by `measure` statements in the language:

```
measure hist occupancy at e1;
measure hist occupancy at e21;
measure hist occupancy at e22;
measure hist occupancy at e31;
measure hist occupancy at e32;
measure hist occupancy at e4;
measure backpressure at e4;
```

and the histogram queries yield the graphs in Figure 6.5. The average backpressure at `e4` is always 0, so we do not show that graph.

These queue occupancy histograms enable us to answer question five:

What portion of the pipeline is limiting the achievable throughput?

by observing that all of the queues downstream of the `Split` block are empty almost all of the time while the queue upstream of of the `Split` block is full almost all of the time. This is a strong indication that the `Split` block is the rate limiting element in the pipeline.

Our suspicions are confirmed when we replace the original implementation of the `Split` block with a new implementation that is more efficient. The efficiency is increased by moving data through the block in larger chunks. With this new `Split` block, the median rate has increased to 8.6 Mtransfers/s across edge `e1` and the histograms of queue occupancies on edges `e1`, `e21`, and `e22` are shown in Figure 6.6. The histograms for edges `e31`, `e32`, and `e4` did not appreciably change and they are not replotted.

These plots both confirm the hypothesis that the `Split` block was the initial throughput bottleneck in the pipeline (since replacing it with a faster implementation provided a greater than 3× performance gain) and enable us to answer question six:

If that bottleneck were resolved, what would be the next bottleneck?

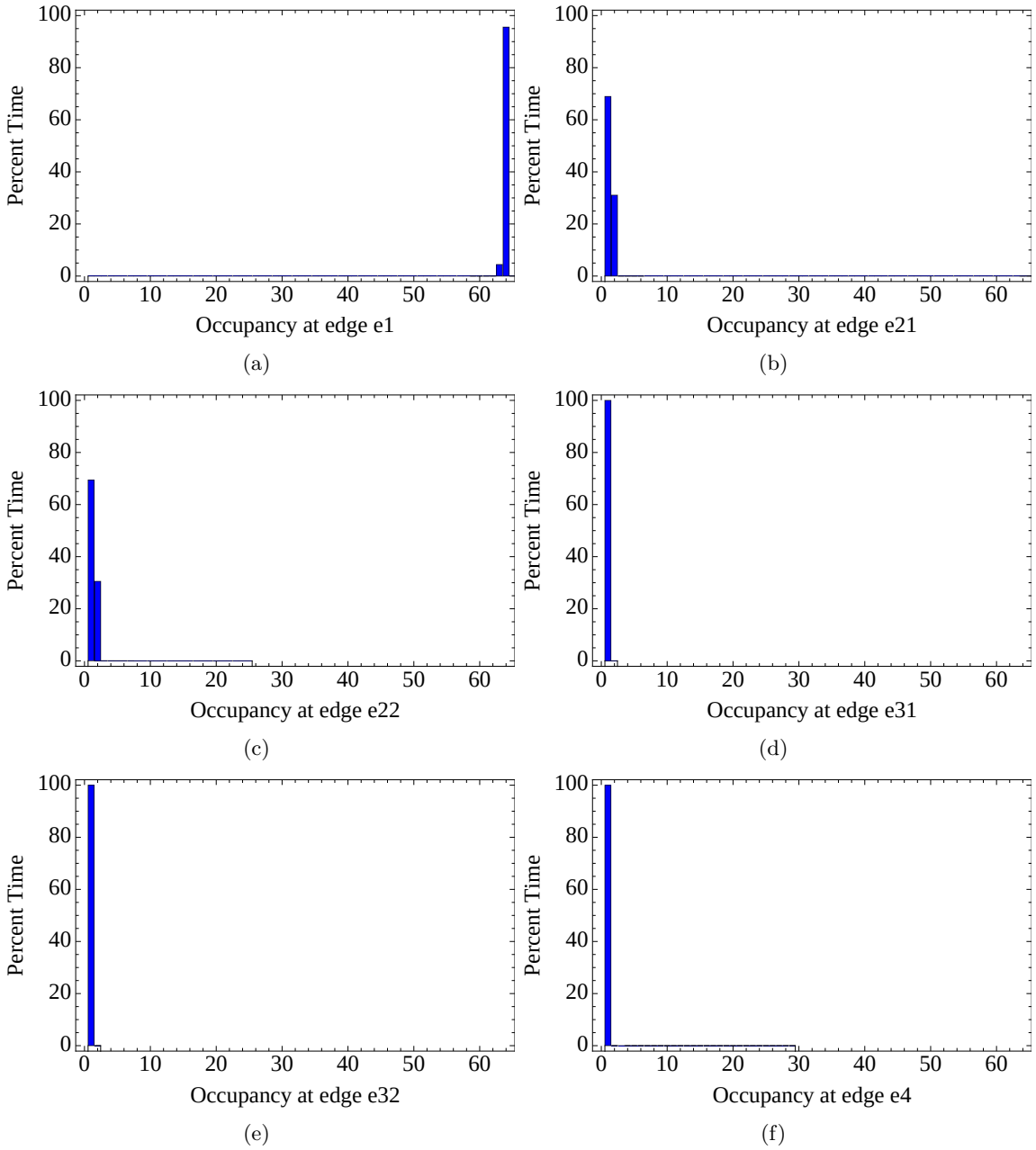


Figure 6.5: Histograms of queue occupancies for the topology shown in Figure 6.2.

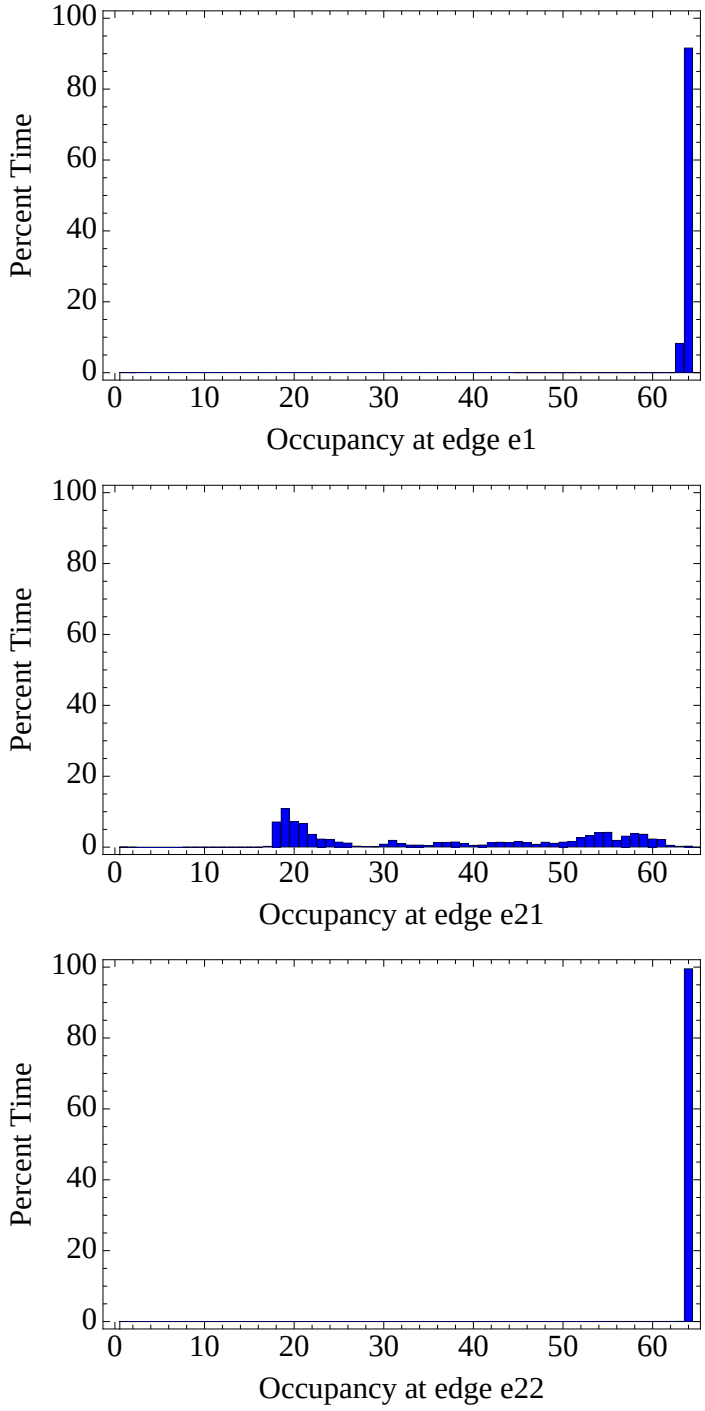


Figure 6.6: Histograms of selected queue occupancies for Figure 6.2 after replacing `s1` with a more efficient implementation.

Now, the queues into the `Walk` blocks are non-empty and the queues out of the `Walk` blocks are empty. This implies that the random walks are now the throughput limiting elements (i.e., the next bottleneck).

We can explore the dynamics of the queue leading into `Walk` block `w1` by plotting the queue occupancy histogram as a function of time. This is shown in Figure 6.7. In the perspective presented in the figure, time (indicated by frame) progresses into the page. At the early portion of the run the queue is full, and as the run progresses the occupancy falls off (although never becoming empty). This time line illustrates how the `Split` block maintains constant data rate at its two outputs, even if the consumption by the two `Walk` blocks isn't completely balanced, leading to a decrease in queue occupancy over time for `Walk` block `w1`.

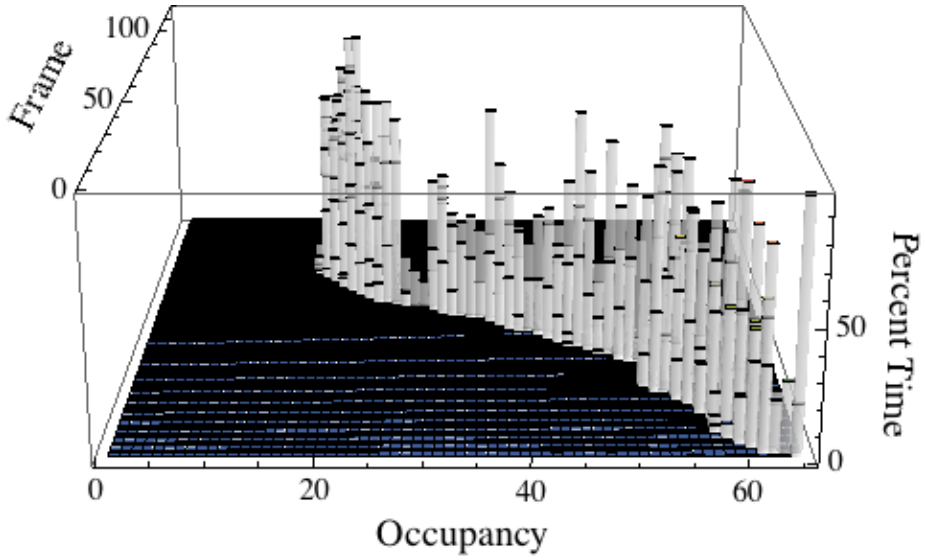


Figure 6.7: Histogram of queue occupancy over time for edge `e21` (input to block `w1`).

Given that the `Walk` blocks have been shown to be the current throughput bottleneck, we can explore further performance improvement by both altering the application's topology and exploiting the available architectural diversity in the execution platform. The execution platform has 12 cores and an FPGA co-processor, so the next implementation we will explore uses 8 cores for 8 `Walk` blocks, 2 cores for `Split` blocks, 1 core for the `Avg` and `Print` blocks, and reserves one core for the TimeTrial software agent. In addition, the `RNG` block is deployed on the FPGA. This illustrates the flexibility enabled by the Auto-Pipe development environment. If an FPGA implementation is available for a block, deploying the block on the FPGA only requires altering the mapping statements in the X language specification of the application.

The throughput rate coming out of the `RNG` block into the first `Split` block is now 33 Mtransfers/s, a 3.8-fold performance improvement over the previous execution, which had only two `Walk` blocks. The queue occupancies of this edge (both on the hardware side of the bus and the software side of the bus) are shown in Figure 6.8. The fact that they are both continually at capacity indicates that the `RNG` block is not limiting the pipeline throughput, and causes us to pose the question as to whether the FPGA co-processor is actually benefiting the application performance.

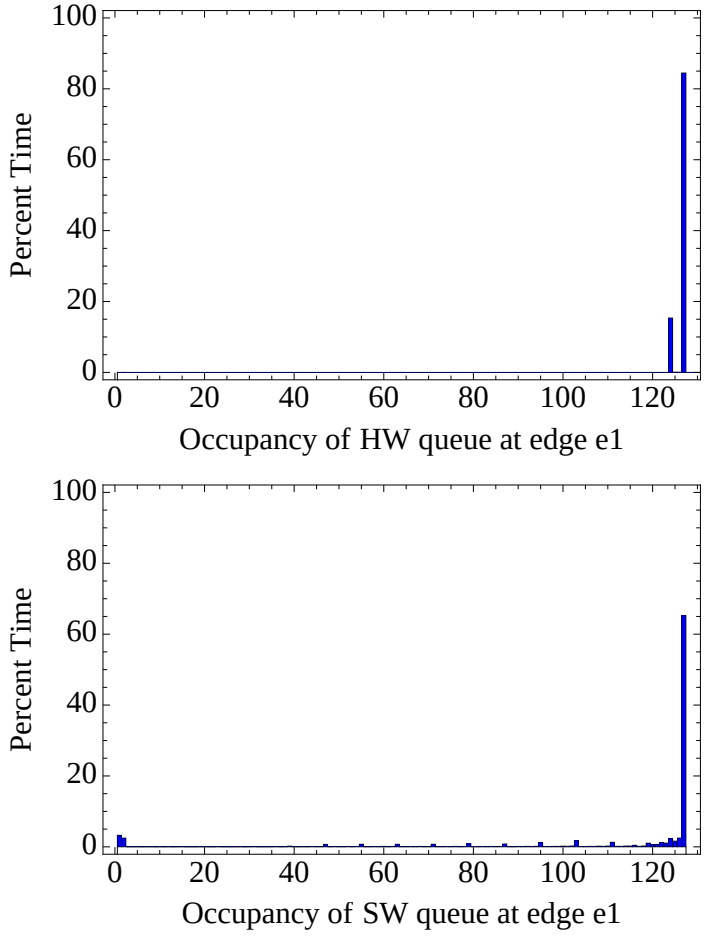
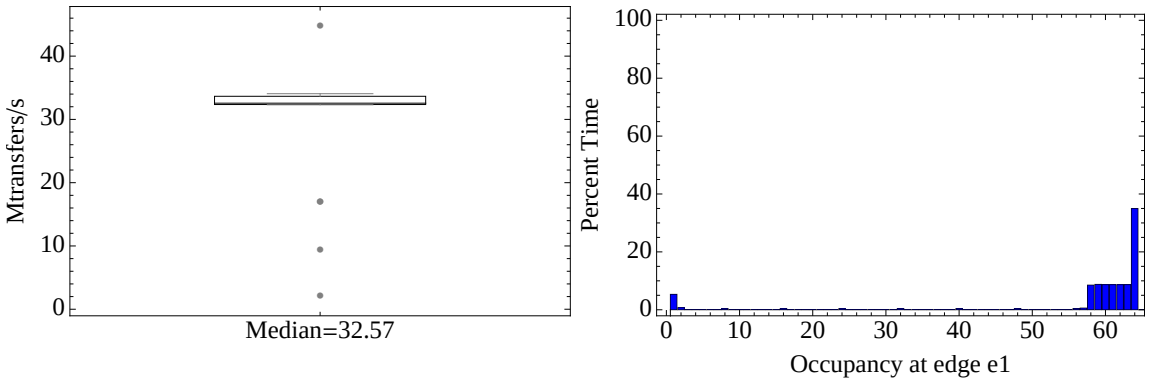


Figure 6.8: Histograms of hardware and software queue occupancies for edge `e1` (output of the `RNG` block).

We can explore the above question by mapping the `RNG` block to one core and assigning all of the `Split` blocks to a single core. Figure 6.9 shows the throughput rate and the queue occupancy out of the `RNG` block for this run. As can be seen in the graphs, the throughput is virtually the same and the queue occupancy is still quite high (indicating that the software `RNG` block has sufficient performance to keep up with the rest of the pipeline (i.e., the FPGA does not benefit the application's performance)).



(a) Box-Whisker Plot of mean rate per frame.

(b) Histogram of accumulated queue occupancy.

Figure 6.9: Performance measurements for edge `e1` (output of the `RNG`block).

For this run, the `Split` blocks are once again the performance bottleneck. This is confirmed by the information presented in Figure 6.10. Here, the queue occupancy histograms for all of the edges going into the eight `Walk` blocks are shown, and all eight queues have significant time during which they are empty.

While in this case the use of architectural diversity does not improve the performance of the application, this fact was not clear prior to the execution and measurement of performance. As with any empirical measurement, one must have access to the artifact being measured, and the Auto-Pipe environment facilitates quick transitions between block to compute resource mappings, greatly simplifying the task of understanding the performance implications of a wide variety of deployment options.

6.1.1 Virtual Queues

Here we illustrate the ability of TimeTrial to monitor temporal properties of an application, including over virtual queues. We set TimeTrial's frame period to 1 second and asked for the mean queue occupancy for the queue from `PRNG` to the first `Split` and also for the queue from the last `Avg` to `Print`. The application runs for 60 seconds. For this execution, all of the blocks except `Print` have been assigned to the FPGA and `Print` has been assigned to a processor core. As a result, the first monitored queue is entirely on the FPGA and the second monitored queue is a virtual queue, moving data from the FPGA to the processor core.

Figure 6.11 shows the mean queue occupancy over time for the queue immediately downstream of the `PRNG` block. The capacity of this queue is 128, and as is readily apparent

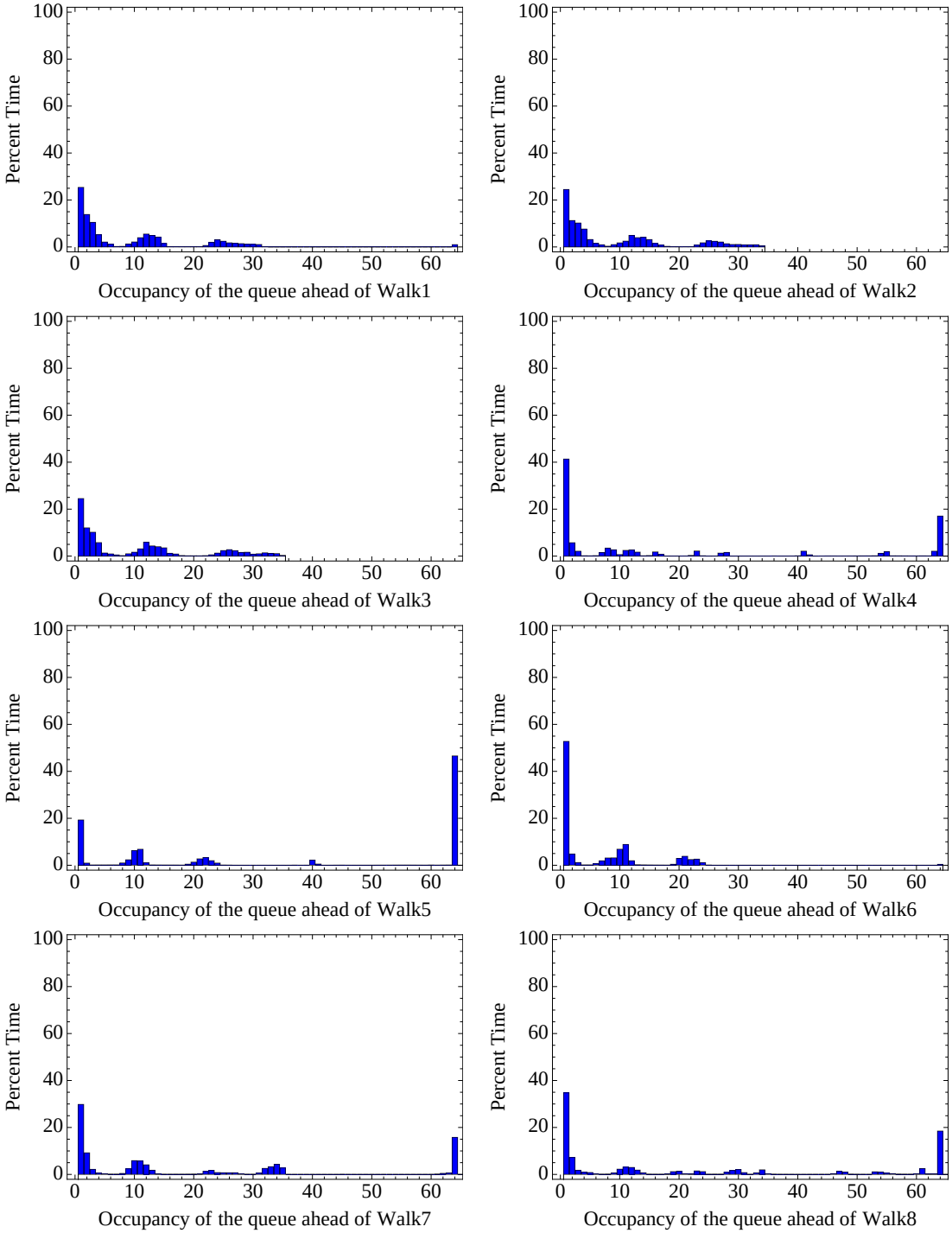


Figure 6.10: Occupancies for the input queues to the walk blocks

from the graph, within the first second of execution this queue is full and stays full for the duration of the run.

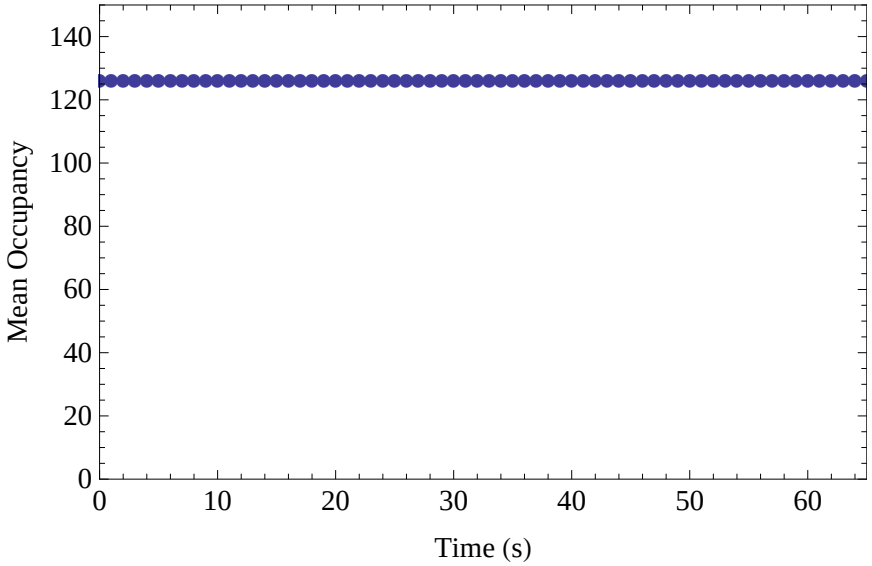


Figure 6.11: Mean queue occupancy out of `PRNG` block in Monte Carlo solution to Laplace's equation. This is a hardware-to-hardware queue, entirely within the FPGA.

Figure 6.12 shows the mean queue occupancy over time for the queue upstream of the `Print` block. Here, the effect of bulk transfers across the bus is clearly apparent. Data destined for the `Print` block is buffered, waiting to be transferred across the PCI-X bus, until the very end of the application's execution, at which point the buffers are flushed and all of the results are delivered to the `Print` block. Note that if the latency of individual data elements moving from `Avg` to `Print` is an important performance metric for this application, simply monitoring the average rate across the edge is not sufficient to discover the true nature of this virtual queue's activity.

Further monitoring using TimeTrial (not shown here) demonstrates that the `Walk` blocks are the current performance bottleneck, and increasing the parallelism to greater than 8 `Walk` blocks is an important step in improving the overall application performance.

6.2 Biosequence Search using BLASTN

In this section, we demonstrate our measurement infrastructure by characterizing the performance of Mercury BLASTN, a streaming, FPGA-based implementation of the NCBI BLASTN tool for DNA similarity search. TimeTrial is used to characterize many different

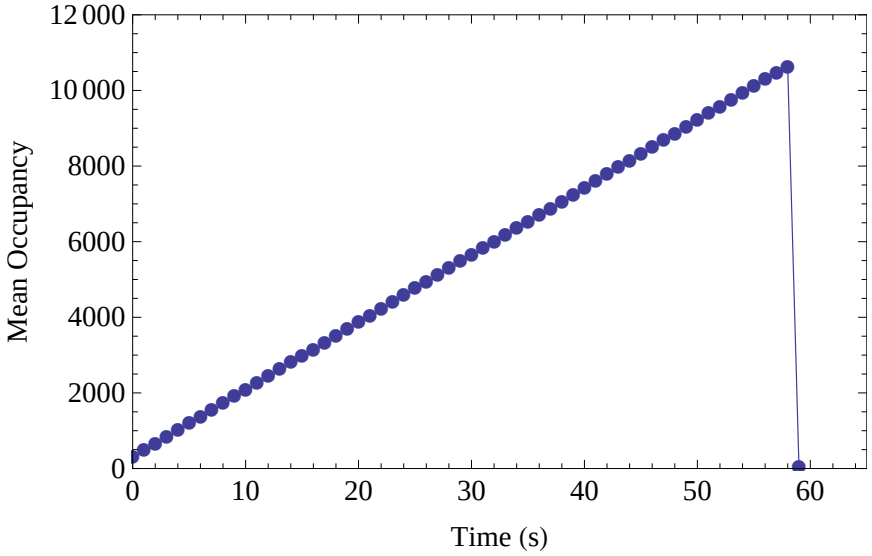


Figure 6.12: Mean queue occupancy into `Print` block in Monte Carlo solution to Laplace's equation. This is a hardware-to-software virtual queue, moving data from the FPGA to a processor core.

aspects of Mercury BLASTN. We show that TimeTrial can extract performance measurements from this application with low overhead and can discover the causes of bottlenecks that were previously unknown.

6.2.1 The BLASTN Application

BLAST, the Basic Local Alignment Search Tool [5], is widely used by molecular biologists to discover relationships among biological (DNA, RNA, and protein) sequences. The BLAST application compares a *query sequence* q to a database D of other sequences, identifying all *subject sequences* $d \in D$ such that q and d have small edit distance between them. The edit distance is weighted to reflect the frequency with which different mutations, or sequence changes, occur over evolutionary time. The BLASTN variant of BLAST expects both query and database to contain DNA sequences, which are strings composed of the four *bases* A, C, G, and T.

The BLAST application is a critical part of many computational analyses in molecular biology, including recognition of genes in a genome, assignment of biological functions to newly discovered sequences, and clustering large groups of sequences into families of evolutionarily related variants. The last decade of advances in high-throughput DNA sequencing have led

to exponential increases in the sizes of databases, such as NCBI GenBank [82], used in these analysis, and in the volume of novel DNA sequence data to be analyzed.

BLASTN is conceptually a streaming application, composed of a linear 3-stage pipeline of increasingly expensive but increasingly accurate search operations performed on a database stream (see Figure 6.13). In stage 1, "seed matching," BLASTN detects short exact substrings, or words, that are common to both the query and a database sequence, using a hash table of all words in the query. In stage 2, "ungapped extension," the region surrounding each word is searched to detect pairs of longer substrings that differ by just a few base mismatches. Finally, the small fraction of words that generate such an "ungapped" pair are passed to stage 3, "gapped extension," which searches the region around them for pairs of substrings with small edit distance, allowing for base substitutions, insertions, and deletions. Only matches that pass this final stage, called "gapped alignments," are reported to the user.

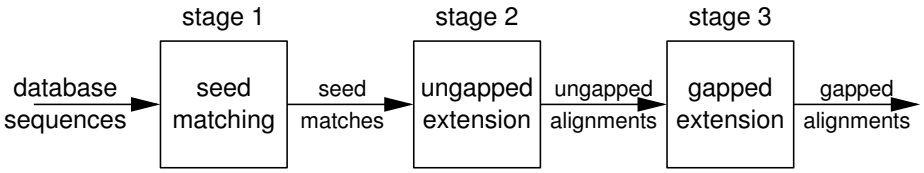


Figure 6.13: BLASTN functional pipeline.

A typical execution of BLASTN includes a number of queries, each of which is compared to a large genomic database (several Gbases in size). Measurements of end-to-end execution time show significant variation depending on the contents of the query and the database. We illustrate the use of TimeTrial to investigate these execution time variations. We set the TimeTrial data frame size equal to a single pass of a database through the pipeline for an individual query. Multiple passes (each with a distinct query) then comprise the complete execution run.

6.2.2 Mercury BLASTN

Mercury BLASTN [65, 67] accelerates the BLASTN algorithm using a heterogeneous system consisting of both FPGAs and general-purpose CPUs. Figure 6.14 shows the mapping to a heterogeneous platform. The FPGA is used for the first two stages of the BLASTN computation, which dominate the running time, while the last stage is assigned to a set of processor cores. On the FPGA, stage 1 is divided into two parts: stage 1b implements the BLASTN query hash table in SRAM attached to the FPGA, while stage 1a hashes the same query words into a Bloom filter [15], a "lossy" lookup table that can be implemented

efficiently internally to the FPGA. Stage 1a proves that most words in the database do not occur in the query, passing only a few percent of the database's words through to the SRAM lookup of stage 1b. Stage 2a further filters the results by expanding the context around around the seed using a one dimensional dynamic programming algorithm.

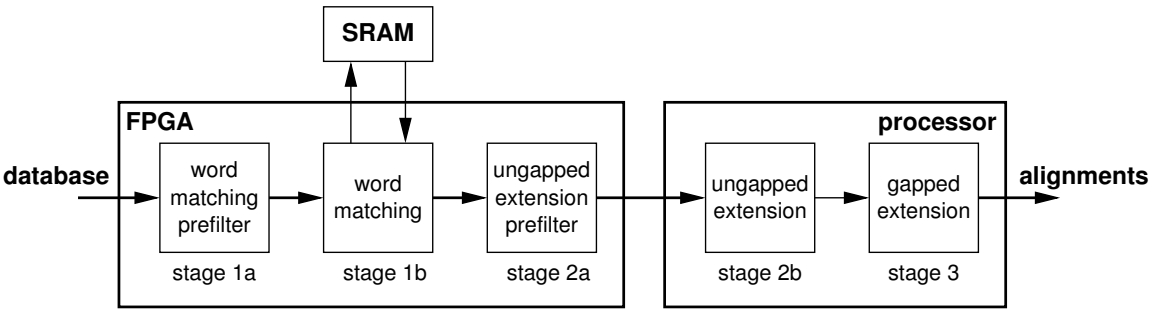


Figure 6.14: Overview of Mercury BLASTN deployment.

6.2.3 Provisioning FPGA Queue Sizes

The original implementation of Mercury BLASTN was designed to carefully balance the loads on the various stages so that no one stage was too great a bottleneck. A deep understanding of its performance was achieved by studying a near-cycle-accurate simulator of the implementation written in C. However, the original design has since been ported to a new, more modern FPGA hardware platform that changes the clock speeds of most components and greatly alters the properties (latency, bandwidth, etc.) of the attached SRAM. Rather than resurrect and modify the old simulator, we opt to use TimeTrial to directly measure the performance of the new implementation.

Mercury BLASTN contains over 20 queues. These queues can consume significant resources if they are over-provisioned. The first goal is to appropriately size a queue by directly measuring the occupancies. One potential performance issue with Mercury BLASTN is a high latency SRAM might require a large buffer for high throughput operation. There is a queue, we'll call it the SRAM queue, in front of the SRAM to buffer words from stage 1a while the SRAM is processing older words. Our goal is to size this queue so that it handles the majority (e.g., $> 95\%$) of the cases without over-utilizing resources.

Mercury BLASTN was tested using the NCBI mammalian RefSeqs[2] as the database, with queries sampled from non-mammalian RefSeqs[3]. The hardware platform consists of 2 quad-core AMD Opteron CPUs running at 2.4 GHz, 16 GB of system RAM, running CentOS 5.3

[2] `ftp://ftp.ncbi.nih.gov/refseq/release/vertebrate_mammalian/`
[3] `ftp://ftp.ncbi.nih.gov/refseq/release/vertebrate_other/`

operating system. The FPGA card contains an Xilinx Virtex 4 LX100 speed grade 12 part and communicates with the processors via PCI-X bus running at 133 MHz.

We begin this investigation with the SRAM queue potentially over-provisioned to hold 512 elements, based on our understanding of the application. Due to the latency of the SRAM, up to 28 more entries might fill the queue after it asserts back pressure. This effectively makes the queue size 484, with 28 slots being reserved for in-flight SRAM responses and 484 slots for smoothing out bursty application behavior. This queue was then instrumented to record a cycle-accurate histogram of the queue occupancy over the entire experiment listed above.

Figure 6.15 shows the results. From this data, we concluded that we may be able to significantly reduce the size with nominal performance loss since the majority of the time the queue occupancy is much less than 484.

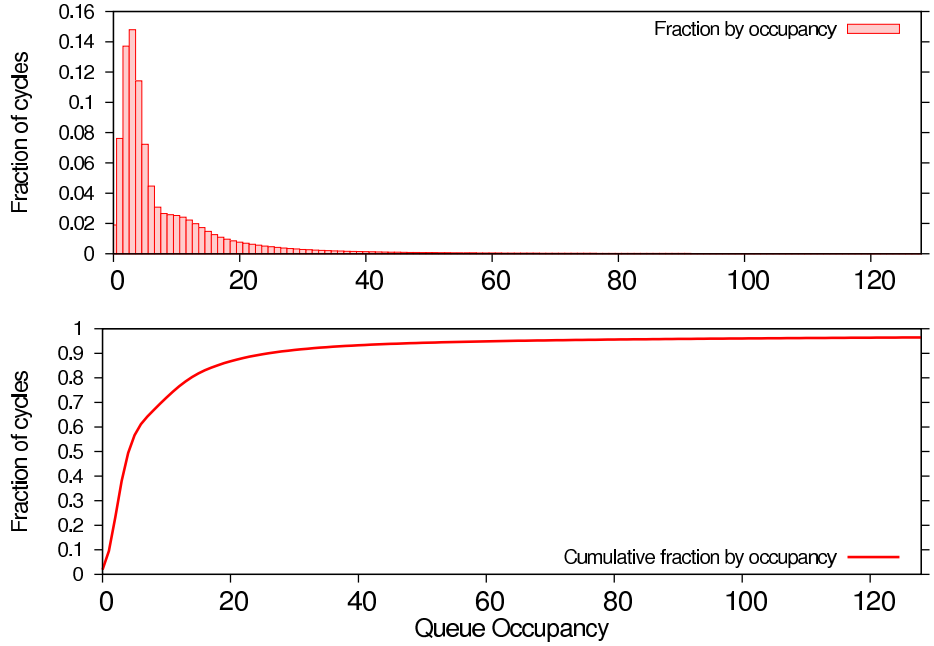


Figure 6.15: Measurement results from measuring the SRAM queue of total size 512. The top graph shows the normalized histogram of the counts of queue occupancy over the entire execution. The bottom graph shows the cumulative fraction of the queue occupancy.

The total number of cycles to execute the run was 8.6176×10^{10}, which was obtained by summing the number of cycles in every frame. From Figure 6.15, we can estimate that decreasing the queue to 128 (an effective length of 100) we will increase the runtime by approximately 2%, while utilizing 1/4 the amount of resources. To verify this, we repeated the experiment with the SRAM queue size set to 128. This configuration took 8.6788×10^{10} cycles to finish, a decrease in performance of 0.7%, even better than expected.

This experiment helped us free up 4 Block RAMs that can be utilized in other parts of the design.

6.2.4 Profiling Mercury BLASTN

In a more extensive investigation, we instrumented Mercury BLASTN in a detailed manner, executed two different large genomic data sets, and analyzed the performance results from TimeTrial. Since Mercury BLASTN is known to have performance characteristics that depends on the data sets that are executed, the focus of this investigation is to determine how executing different genomic data sets affects its performance. TimeTrial accomplishes this by measuring the utilization of communication edges and the distribution of the queue occupancies in the application.

TimeTrial instrumentation was added to every top-level block in Mercury BLASTN. Figure 6.16 shows the functional pipeline of Mercury BLASTN in more detail, highlighting the points where the TimeTrial performance monitor is tapping signals of interest. The application consists of multiple copies of each stage executing in parallel. Stages 1a, 1b, and 2 are deployed on FPGAs (there are two FPGAs utilized in this configuration), and stage 3 is deployed on a collection of processors. Ahead of stage 1a is a pair of software threads that manage delivery of the queries and database to the hardware pipelines, and between stages 2 and 3 is a set of software threads (one per FPGA pipeline) that collect ungapped alignments from the hardware and enqueue them for stage 3 processing.

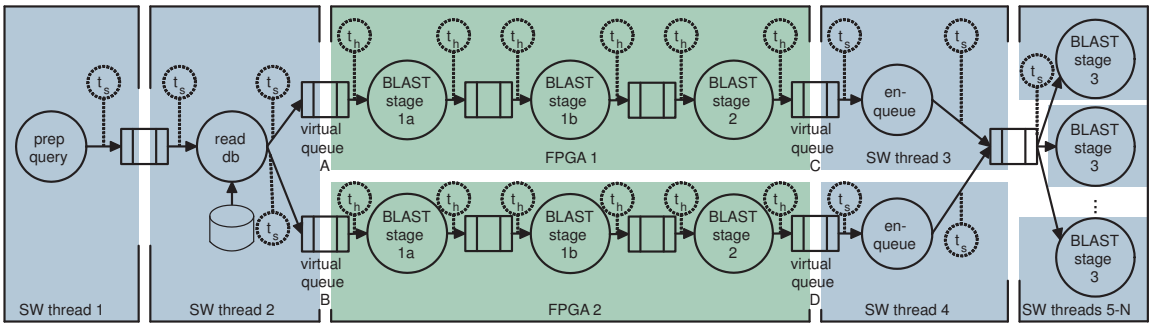


Figure 6.16: BLASTN functional pipeline detail. Taps in the software subsystem are labeled t_s and taps in the hardware subsystem are labeled t_h. Virtual queues A through D cross the HW/SW boundary.

Since the BLAST application employs filtering as part of its basic operation, the entries in each queue vary as one moves down the pipeline. At the far left, individual queries are prepared for delivery to the FPGA, and queue entries for the leftmost queue represent

these queries. In virtual queues A and B, the database is being streamed into the FPGA, so queue entries represent database characters. The individual queues within the FPGA and virtual queues C and D coming out of the FPGA each store entries that are represented by an ordered pair of (query position, database position). Finally, the enqueue software functions aggregate the ungapped alignments out of stage 2 and package them by query, thereby making the entries in the last queue again equal to individual queries.

Due to the existence of one common queue between the set of FPGA-deployed pipelines and the set of stage 3 threads, the number of stage 3 threads can be altered to accommodate a lower or higher computational load in stage 3. This is indicated in the figure by software threads labeled 5 through N, where N is settable at execution time.

In the pure software queues, enqueue and dequeue event traces are processed into histograms of queue occupancies. However, as was mentioned earlier, queues that cross the HW/SW boundary are combined into one virtual queue. To determine the occupancies of these queues, we measure the event trace on the software side and measure the mean throughput on the FPGA side. The queue occupancy is then simulated based on the event trace in software combined with an exponentially distributed inter-dequeue time parametrized by the empirically obtained mean. The occupancies of the queues residing entirely on the FPGAs are measured on chip by the FPGA monitoring agent and communicated to the software monitor.

To profile the performance of Mercury BLASTN, two sets of sequences were compared and TimeTrial collected runtime statistics at native application speeds. The data sets used in the experiments is as follows:

- Experiment 1: The first data set consists of a recombinant viral DNA (sequenced in-house at Washington University in St. Louis, 293×10^6 bases) against the 19th assembly of the human genome[4] (3×10^9 bases). The viral DNA was split up into 65,400 size queries, each of which Mercury BLASTN compared to the entire human genome.
- Experiment 2: The second data set consists of the non-mammalian RefSeqs[5] (302×10^6 bases) as the query. The NCBI mammalian RefSeqs[6] (788×10^6 bases) are used as the database. Again, the non-mammalian RefSeqs are split up into smaller segments (48,000 bases) and compared to the entire mammalian RefSeq data set.

[4] `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/`
[5] `ftp://ftp.ncbi.nih.gov/refseq/release/vertebrate_other/`
[6] `ftp://ftp.ncbi.nih.gov/refseq/release/vertebrate_mammalian/`

Mercury BLASTN splits the query sequence into smaller, overlapping sequences with 5% overlap. Then, the database is streamed through once for each query to complete the entire comparison. The hardware platform consists of 2 quad-core AMD Opteron processors running at 2.4 GHz, 16 GB of system RAM, running the CentOS 5.3 operating system. The FPGA board contains a Xilinx Virtex 4 LX100 speed grade 12 part and communicates with the processors via a PCI-X bus running at 133 MHz.

Even though Mercury BLASTN is not currently implemented within the Auto-Pipe framework, the measurements can be expressed in the TimeTrial language. TimeTrial measurement statements for Mercury BLASTN application example are shown in Figure 6.17. We measure the average utilization at 4 taps within each FPGA, and the queue occupancy of several queues in software, hardware, and those crossing the boundaries. TimeTrial is configured to aggregate statistics over each pass of the database, one for each query. This corresponds to a data frame set to the size of the database for each of the two experiments (e.g. frame size is 3×10^6 bases for Experiment 1).

TimeTrial's profile results for the accelerated BLASTN are shown in Figure 6.18. Software processes/threads are shown as circles, while FPGA modules are shown as squares. The two FPGAs are aggregated into one set of measurements for clear presentation. The solid lines between stages represent the communication links that transfer data from one stage to the next. The dashed lines are *back-pressure* signals, a control signal which, when active, indicates that a stage is busy and no data should be sent. Data is summarized through box-whisker plots and histograms of queue occupancy. The box-whisker plots show the variability of each measurement averaged over a single pass of the database (the data frame size is set to one pass of the database). In other words, TimeTrial records one value (e.g. mean throughput at a point) for each pass, and these values are combined into a box-whisker plot containing the median and quartiles. Similarly, each pass over the database results in a queue occupancy histogram for each instrumented queue. Each measurement is then accumulated to form an aggregate view of the occupancy over the entire execution.

The upper left-hand side of the pipeline shows the ingest rate of the FPGA, while the results below show the percent utilization of the communication links between FPGA stages. The utilization of the egress link is also shown at the output of stage 2. The right-hand side shows queue occupancy histograms for major software and hardware queues. In addition, back-pressure signal utilization is measured within the FPGA and is shown as horizontal box plots.

Studying Figure 6.18, we find that in both cases, the software portion of BLASTN is non-limiting. We draw this conclusion from the full queue at the ingress of the FPGA and

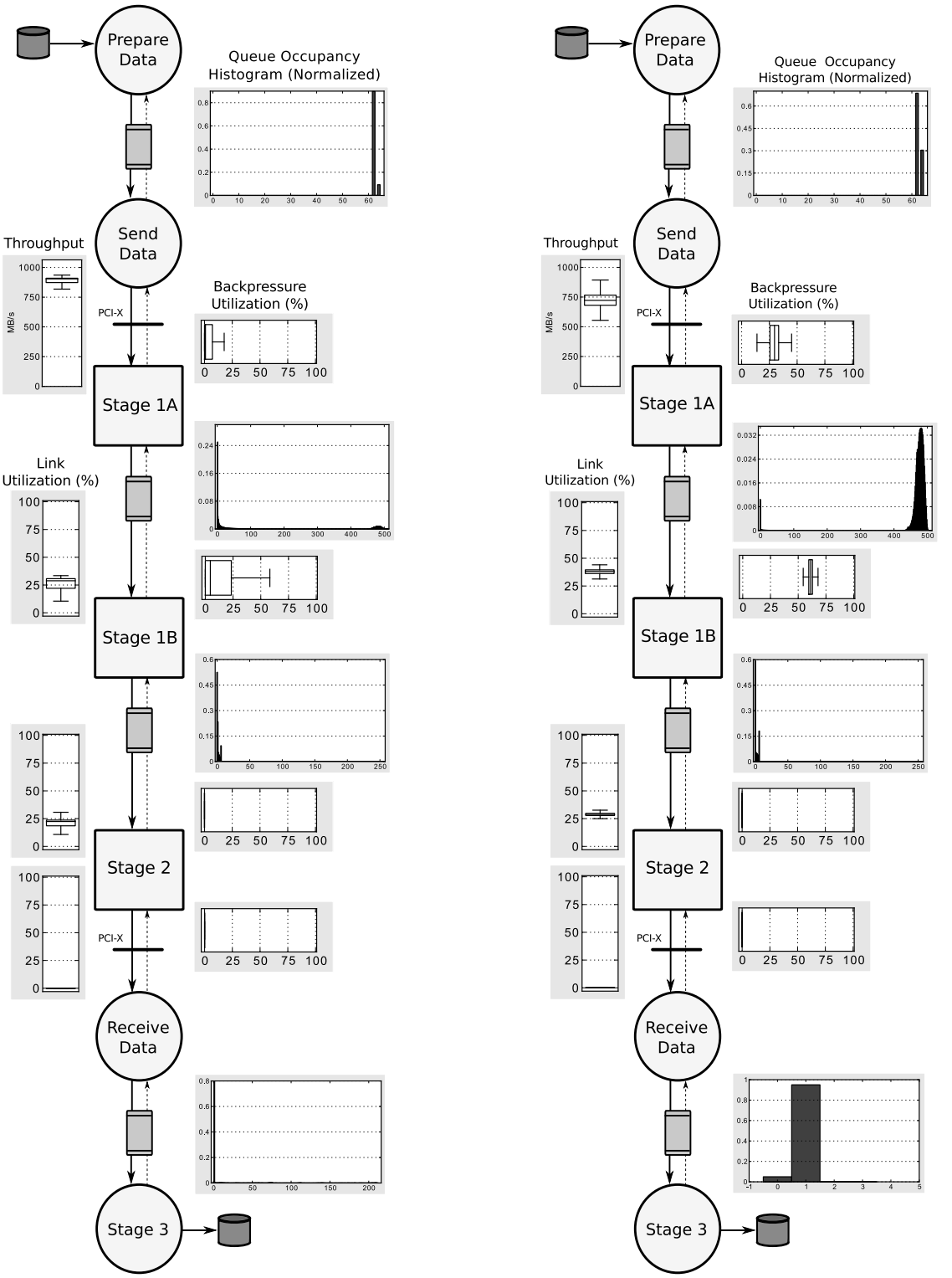
```
// edge labels (X language)
queryq: prepquery.out -> readdb.in
vqA: readdb.out -> FPGA1.BLASTstage1a.in
vqB: readdb.out -> FPGA2.BLASTstage1a.in
hw1q1b: FPGA1.BLASTstage1a.out -> FPGA1.BLASTstage1b.in
hw2q1b: FPGA2.BLASTstage1a.out -> FPGA2.BLASTstage1b.in
resultsq: enqueue.out -> BLASTstage3.in

// measure utilization on FPGA1
measure mean util at FPGA1.BLASTstage1a.in
measure mean util at FPGA1.BLASTstage1b.in
measure mean util at FPGA1.BLASTstage2.in
measure mean util at FPGA1.BLASTstage2.out

// measure utilization on FPGA2
measure mean util at FPGA2.BLASTstage1a.in
measure mean util at FPGA2.BLASTstage1b.in
measure mean util at FPGA2.BLASTstage2.in
measure mean util at FPGA2.BLASTstage2.out

// measure histograms of queues
measure hist occupancy at queryq
measure hist occupancy at vqA
measure hist occupancy at vqB
measure hist occupancy at hw1q1b
measure hist occupancy at hw2q1b
measure hist occupancy at resultsq
```

Figure 6.17: Equivalent TimeTrial language statements for BLASTN application runs.



(a) Experiment 1 profile

(b) Experiment 2 profile

Figure 6.18: TimeTrial measurement results from running BLASTN on two large genetic data sets.

the empty queue at its egress. However, there is some burstiness in stage 3 processing for Experiment 1: about 20% of the results from the FPGA over-run the available processors and start to fill the queue. Experiment 2 never has more than one element in this queue.

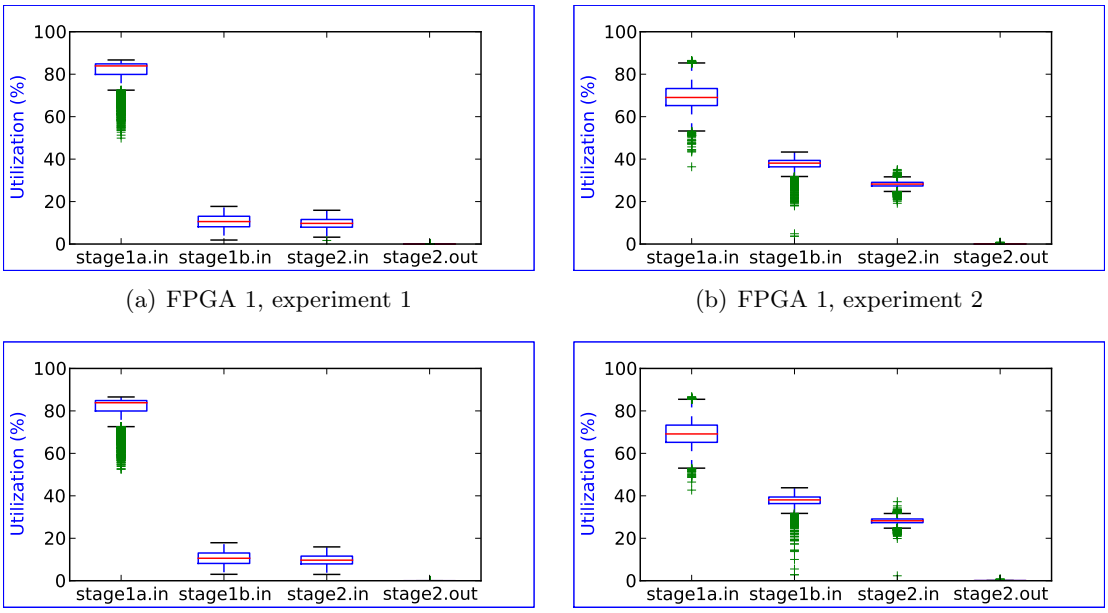
The hardware shows a different story between the two experiments. Experiment 1 has a much higher and less variable throughput compared to Experiment 2. In both experiments, when the throughput is less than the maximum possible line rate (∼875 MB/sec), the slowdown is caused by stage 1B asserting back-pressure. Experiment 2 exhibits much higher back-pressure at stage 1B, indicating that this stage is a bottleneck. This observation is corroborated by the high occupancy of the stage's input queue. In both experiments, blocks downstream of stage 1B have plenty of capacity and almost never assert back-pressure. We conclude that Experiment 1 stresses stage 1B in about half of the passes over the database, but that the buffering upstream tends to ameliorate this. In Experiment 2, stage 1B is more severely stressed, filling the upstream buffers and causing lower ingest rates.

In these examples, TimeTrial paints a clear profile of the application performance and allows the developer to efficiently locate bottlenecks. Figure 6.18 shows a high-level analysis of the performance. To determine what is causing the poor performance inside stage 1B, we instrumented BLASTN with additional taps that monitor signals and state inside the stage. Using these taps, a number of inefficiencies were discovered that contributed to the performance loss; the biggest insight was gained from tracking the FSM that controls reads and writes to the SRAM. We discovered that the hash table caused collisions, and hence multiple SRAM probes, for more than half of all initial accesses to the SRAM. Unfortunately, this condition results in the most inefficient use of the SRAM for this design, since performance of sequential lookups is limited by the SRAM's round-trip latency. To correct this issue, the hash table size should be increased to use more of the available SRAM capacity on the board. As a result, hash table collisions will be greatly reduced and stage 1B will utilize the SRAM more efficiently.

Figure 6.18 refers to a deadlock-free version of Mercury BLASTN that has been through many iterations of tuning with TimeTrial. Even though Mercury BLASTN is typically drawn as a linear pipeline the implementation actually uses a separate channel for the database from stage 1a to stage 2a, bypassing stage 1b. This structure is actually a fork-join structure and is susceptible to deadlock if the two streams get far enough apart. Since BLASTN stages act a filters, the streams can diverge due to stage 1a and 1b processing potentially large amounts of input without creating output and stage 2a having a finite buffer size to hold the database. To avoid deadlock, stages 1a and 1b both create *null* messages at certain intervals during periods of no real output to indicate to stage 2a that

portions of the database can be discarded. In what follows we show more detailed results from a previous version of Mercury BLASTN that was susceptible to deadlock because the null message rate was set too low.

Figure 6.19 shows the measured results of each of the utilization of the communication channels. For each experiment, the box-whisker plots represent the variability of the measured average utilizations for each frame. In both experiments, FPGA 1 and FPGA 2 have very similar utilization profiles and the utilization of the stage2.out edge is very low and hence unlikely to be the pipeline bottleneck. There is a significant difference in the utilization of all three edges for experiment 1 relative to experiment 2. Compared to the deadlock-free version of Mercury BLASTN above, the utilization of the edges following 1a are lower for experiment 1. This is due to a lower rate of null messages being sent from stage 1a to stage 1b and from stage 1b to stage 2a. Note that in both the deadlock-free and this version, the ingest rate is nearly identical indicating that the deadlock-avoidance algorithm has little effect on the overall performance.



(a) FPGA 1, experiment 1

(b) FPGA 1, experiment 2

(c) FPGA 2, experiment 1

(d) FPGA 2, experiment 2

Figure 6.19: Communication link utilization broken down by FPGA.

Figures 6.20 and 6.21 show histograms of the occupancies of all the instrumented queues for each experiment over all database passes. Similar to the utilization graphs described above, columns of plots correspond to a single experiment. The effects of the different data sets

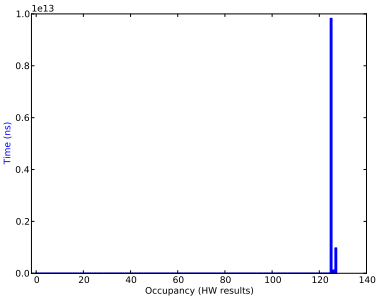
on the queue occupancies can be observed by comparing across rows. The virtual queue occupancies are obtained by simulating the queue for five random database passes.

The occupancies of the queue in the first row of Figure 6.20 are significantly different. This row represents queues that have been prepared and are ready to send to the FPGAs (the queue between SW thread one and two in Figure 6.16). Recall that Mercury BLASTN compares each query to the entire database. The lower queue occupancy can be explained by looking at the relative size of a the databases in the two experiments. Since experiment 1 has a much smaller database, there is less time to prepare each query per database pass. This was an unexpected source of performance limitation in the Mercury BLASTN architecture. If smaller database sizes are needed, the performance of the system can be improved by optimizing this task or adding an additional query preparation thread.

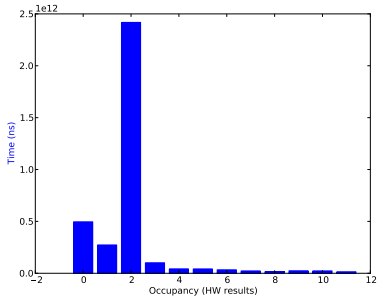
The second and third rows show the calculated occupancies for the virtual queues A and B. Since the workload is fairly evenly divided between FPGAs, the occupancies of A and B are similar for all experiments. The virtual queues histograms all have a flat nature since the enqueue rate is very high (e.g. copying from user-space into kernel buffers) and are drained at a much slower but relatively consistent rate within the FPGA.

The bottom row of Figure 6.20 shows the occupancy of the results queue. The occupancy of this queue is determined by the volume of results coming out of the FPGA as well as the rate at which the software servers are able to process this data. While experiment 1 still has a relatively low mean, there is a considerable tail on this histogram. This tail is due to some queries of the experiment producing a very large number of ungapped alignments, temporarily overwhelming the software server capacities. In other words, the output from the FPGAs is much more bursty than in experiment 2. This burstiness is in no small part due to the differences in base composition between a subset of the queries and the database. High commonality in base composition can yield an increase in match rate out of the filter stages (i.e., stages 1a, 1b, and 2).

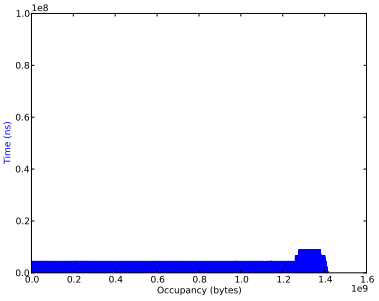
Observing the queue inside the FPGA, the queue only backs up significantly on experiment 1, where it stays full most of the time. The utilization of `stage1a.in` in Figures 6.19(b) and 6.19(d) confirm that the utilization of the input stream is lower that the other experiments due to the FPGA servers slowing. This is due to data-dependent filtering behavior of the algorithms executed in the servers filtering less data and filling the capacities of the on-chip queues. Similar to the software queues, the two FPGAs have very similar occupancies for each experiment.



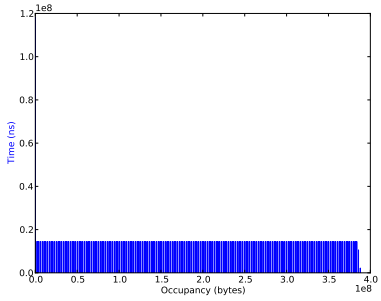
(a) Query queue, exp. 1



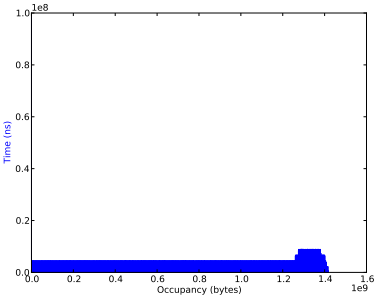
(b) Query queue, exp. 2



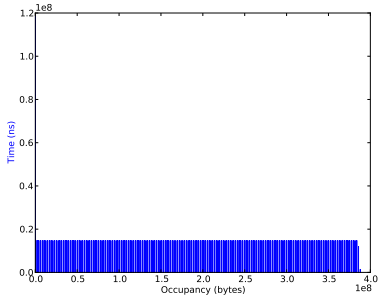
(c) Virtual queue A, exp. 1



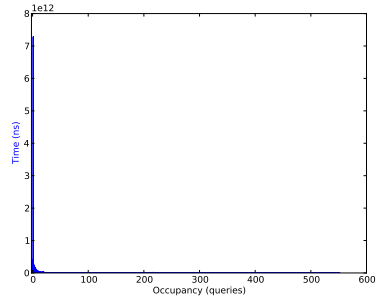
(d) Virtual queue A, exp. 2



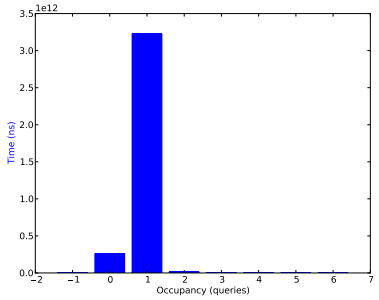
(e) Virtual queue B, exp. 1



(f) Virtual queue B, exp. 2

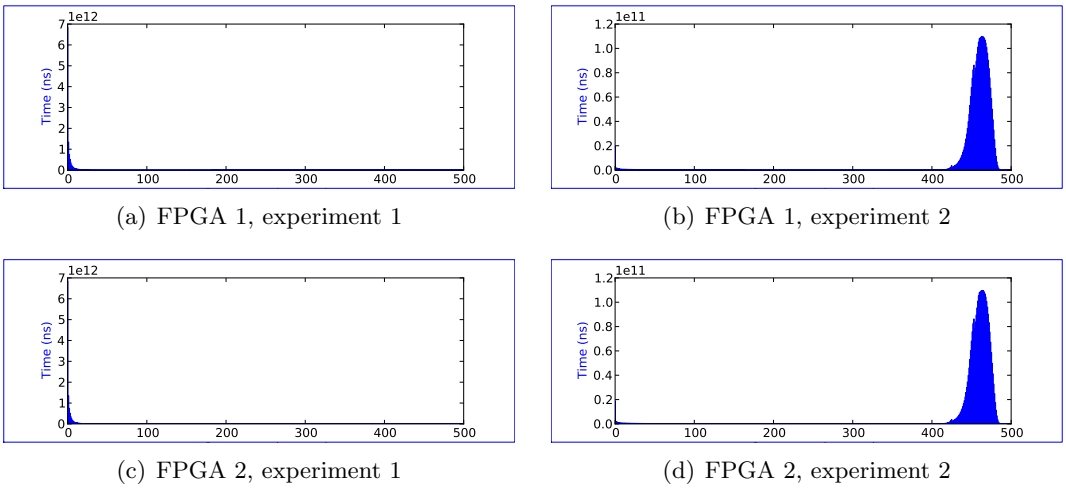


(g) Results queue, exp. 1



(h) Results queue, exp. 2

Figure 6.20: Software-only and virtual queue occupancy histograms.



(a) FPGA 1, experiment 1

(b) FPGA 1, experiment 2

(c) FPGA 2, experiment 1

(d) FPGA 2, experiment 2

Figure 6.21: FPGA-only stage 1a to stage 1b queue occupancy histograms.

Overheads and Impact

Using TimeTrial to profile an application incurs some unavoidable overhead. We define overhead as the amount of extra resources and communication that is utilized by the TimeTrial system. Impact is the observable difference in runtime due to instrumentation. In what follows, we quantify the the overhead and impact on Experiment 2.

The TimeTrial software instrumentation adds 100 MB (of 16 GB) of main memory overhead to the FIFOs and aggregation functions in the agent. The agent utilizes 27% of one CPU for Experiment 1 and 35% of one CPU for Experiment 2. Communication overhead is primarily due to the software taps sending 46,912 enqueue and dequeue events to the agent. Each event is 64 bytes leading to a total performance meta-data communication overhead of 2.86 MB.

Table 6.1 shows the performance impact both in terms of elapsed wall clock time. Both experiments show negligible percentage perturbation due to TimeTrial instrumentation. The largest increase in run time was seen in Experiment 1, a marginal increase of approximately 4.16 minutes for a 3 hour experiment. All these results are considered an acceptable trade off given the quality and amount of measured results.

The resource overhead of the FPGA agent is shown in Table 6.2. The utilizations for all resource types are single-digit percentages for calculating 30 different application measurements. This is acceptable as long as the application leaves enough free resources for the agent to fit. The instrumentation has no effect on the clock rate for applications clocked

Table 6.1: Performance impact of instrumenting Mercury BLASTN with TimeTrial.

Experiment Number	Elapsed Time no Instr. (sec)	Elapsed Time w/Instr. (sec)	% Diff.
1	11244	11494	2.2
2	3464	3517	1.5

under ∼275 MHz. The FPGA agent sends a record 11,056 bytes long containing all of its performance measurements at the end of each frame. There are 11,728 frames resulting in an extra ∼124 MB of data crossing the PCI-X bus (out of 4.2 TB over the course of the run).

Table 6.2: FPGA agent resource utilization.

Resource Type	Flip Flop	LUT	BRAM	DSP48	f_{max}
Available	98,304	98,304	240	96	∼300 MHz
Used	5,164	6,980	2	1	275 MHz
Utilization (%)	5.3	7.1	0.8	1.0	N/A

We quantify the impact of instrumenting BLASTN by first running an experiment without instrumentation followed by the same experiment with instrumentation. We record the percent difference in user time, system time, wall clock time using the Linux *time* command. User time increased by 3.1%, system time by 8.1% and wall clock time (total run time) by 0.2%. The increases in user and system time reflect the processor cycles dedicated to monitoring, while the very modest increase in total run time reflects the minimal impact that TimeTrial imposes on the application.

Trying to decipher the dynamic performance behavior of Mercury BLASTN without TimeTrial is a very challenging task. Ad ad hoc performance monitoring approaches attempted before TimeTrial yielded very limited insight into the performance at best and completely ambiguous results in many other cases. TimeTrial enabled whole-application performance profiling of Mercury BLASTN, providing precisely the information the developer needs on multi-terabyte data sets in an automated fashion.

6.2.5 Measuring Latency in Mercury BLASTN

To illustrate the capabilities of TimeTrial to measure latencies, we instrumented Mercury BLASTN with different taps, some deployed on the FPGA and others on the processors. Figure 6.22 shows the deployment of the application.

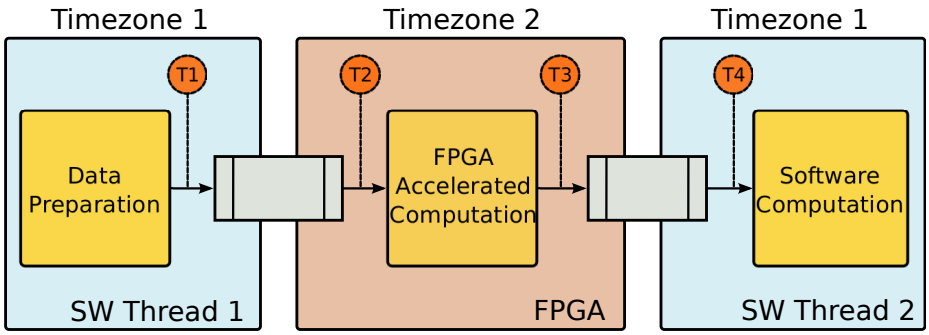


Figure 6.22: Deployment of Mercury BLASTN on a diverse computer. Dashed lines and circles represent the locations of TimeTrial instrumentation taps.

Mercury BLASTN has several timezones. On the processor portion of the application, there is one timezone per processor core. However, we utilize the cycle-counter synchronization feature of modern Linux kernels which synchronizes the cycle counters across multiple cores and multiple chips. This leaves us with effectively one timezone (+/- 1000 processor cycles) on the processor portion of Mercury BLASTN. The FPGA portion of Mercury BLASTN is currently configured so that all the stages run in the same clock domain. When Mercury BLASTN is configured with multiple clock domains, TimeTrial provides one cycle counter per domain. In our configuration, however, this leaves us with two distinct timezones. When data is transferred between timezones, TimeTrial must employ its notion of virtual time to normalize the performance results.

In Mercury BLASTN, the virtual queues to and from the FPGA previously had unknown latency characteristics for data flowing through the application. The two virtual queues are shown in Figure 6.16 as single queues going to and from the FPGA. Using TimeTrial, four different latencies of interest were measured for each pass of the database. First, the latency for sending the first DNA base from the processor to the FPGA was recorded. Referring to Figure 6.16, this corresponds to the latency of the first datum from T1 to T2. One would expect this to be the minimum latency scenario since the queueing delay should be minimized due to empty queues. The latency from T1 to T2 of the last DNA base in the database stream from processor to FPGA was also measured. In a similar manner, the latency of the first result returned from the FPGA to the processor (i.e., from T3 to T4) was also measured. Finally, the latency of the end of stream marker from T3 to T4 was

measured. All four of these measurements require crossing timezones into a common virtual time.

The timezone properties of the experimental system are as follows. AMD Opteron processors running at 2.311 GHz are used as for all the cores. The processor cycle counts are converted to virtual time (i.e. nanoseconds) using a 0.43253 ns/cycle scale factor. FPGA cycle counts also need to be transformed from a 133 MHz clock rate. As a result, $s_P = 0.43253$ ns/-cycle and $s_F = 7.5$ ns/cycle. The value of b_F was experimentally determined using the methodology presented in Chapter 4.

Figure 6.23 shows box-whisker plots of the above described latency measurements to compare each of 1000 48 kilo-base sequences against a 900 mega-base sequence. The latency profiles vary significantly between the four measurements. The first measurement has consistently low latency due to the virtual queue being empty when the first datum is sent. The second measurement has much higher latency because of the added delay from the filled virtual queues. The data-dependent flow throttling from the FPGA application adds to the spread in the latencies. The third measurement has the most variability of any of the measurements because the filtering nature of Mercury BLASTN which frequently does not produce hits for a large portion of the stream. The last datum returned from the FPGA has consistently low latency in large part because there is never significant data volume returning to the processor from the FPGA for this data set. As a result, this virtual queue is never full and the end of data marker that follows the last datum ensures that all the buffers in the path are immediately flushed.

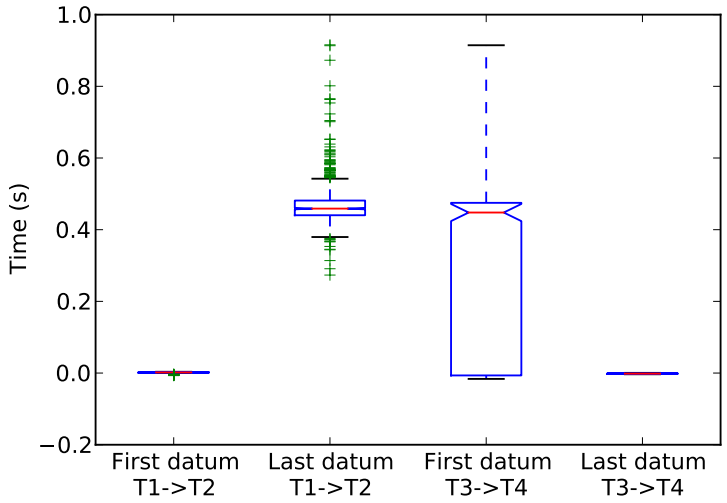


Figure 6.23: Four latency measurements across virtual queues in Mercury BLASTN. The box-whisker plots indicate quartiles (within 1.5 inter-quartile range) and outliers.

6.2.6 Deadlock Avoidance in Mercury BLASTN

In yet another context, TimeTrial was used to evaluate the impact of different algorithms for avoiding deadlock. The algorithms were implemented in Mercury BLASTN and the additional communication overheads were measured and compared to predicted values.

While the high-level picture of the Mercury BLASTN pipeline is a simple tandem sequence (see Figure 6.14), the actual topology has a split-join structure. Li et al. [71] analyzed application topologies of this class that include filtering in one or more compute nodes (as is the case for Mercury BLASTN) and showed conditions under which these applications can deadlock if the buffers between compute nodes are bounded. They also provided three algorithms that provably avoid deadlock through the use of dummy messages, messages that do not have a data payload, but are used exclusively for deadlock avoidance purposes.

This work was extended in [72], using TimeTrial to measure the required dummy message counts for Mercury BLASTN for all three algorithms for a variety of assigned buffer sizes. Figure 6.24 illustrates the split-join topology that is present in Mercury BLASTN. Table 6.3 shows the dramatic implications for dummy message counts required to avoid deadlock as a function of the algorithm chosen and the total buffer size. Over seven orders of magnitude variation are seen in the message counts.

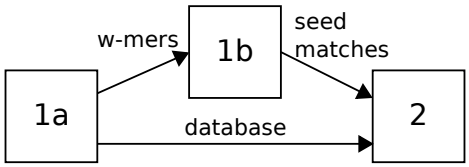


Figure 6.24: The first two stages of Mercury BLASTN [72].

Table 6.3: Measured dummy message counts from stage 1a for Mercury BLASTN.

	Dummy	message	count
Total buffer size (msgs)	32	256	2048
Algorithm 1	787×10^9	787×10^9	787×10^9
Algorithm 2	25×10^9	3×10^9	0.4×10^9
Algorithm 3	36×10^9	36×10^6	$72,000$

6.3 Chapter Summary

In this chapter we utilized TimeTrial to measure the performance of two applications. The first application, a Monte Carlo based solver for Laplace's equation was configured in two different topologies. The fist had two independent walk blocks operating in parallel. Measurements were made and the bottleneck was determined to be in the `Split` block. The implementation of this block was optimized and a performance gain was realized. The next bottleneck was determined to be in the `Walk` blocks. A new topology was generated that used 8 `Walk` blocks. This application was instrumented and achieves a 13.75× speed up over the original implementation. TimeTrial now shows the bottleneck to once again be in the `Split` block.

The second application described was Mercury BLASTN. First, TimeTrial was used to provision an FPGA queue. Then taps were inserted into every stage of the pipeline, queues occupancies were measured, and the bottleneck was shown to lie in stage 1b. Two different experiments were measured, showing unique performance profiles for each. In experiment 1, the bottleneck lies in the I/O bus transferring data to the FPGAs. In experiment 2, the bottleneck is shown to be stage 1b. The exact cause of this bottleneck was determined by exposing finite state machine state signals to the top level port and profiling the time spent in each state. We learned that the next step to higher performance is to increase the size of the hash tables in the SRAM to better utilize the resources on the FPGA board. The overheads and impact of instrumentation were shown to be low. Note that this is but one iteration of the optimization process; TimeTrial has been used to illuminate bottlenecks in 6 other instances not described here. To demonstrate capabilities to measure latency, the latency of two virtual queues were measured and presented. Since Mercury BLASTN has highly data-dependent performance behavior, significant variation in the wait times in the queue were observed. Finally, TimeTrial was used to evaluate the overheads of different deadlock avoidance technique, quantifying their impact on the application performance.

Overall, TimeTrial has proved extremely helpful for performance debugging of heterogeneous applications even in the cases where the instrumentation taps were added manually. Currently, the manual instrumentation method has only been used by one developer (i.e. me) to measure the performance of Mercury BLASTN as described previously. In this mode, TimeTrial collects wholesale metrics after the initial taps are inserted (i.e. the only manual part is adding taps for instrumentation) providing the same performance measurements with a larger user overhead for adding new measurements. However, TimeTrial combined with Auto-Pipe has been employed by several members of our research group on a large number of heterogeneous applications. In each case, the user was able to successfully employ

TimeTrial and interpret the results armed with only the documentation of the language and a simple set of instructions. TimeTrial can also be used to empirically parametrize analytic models, providing a framework to explore the effects of candidate optimizations of an application to determine future bottlenecks. Combined with Auto-Pipe, TimeTrial greatly reduces the burden on the developer by providing an automated, straightforward path to measure any number of performance aspects of an application.

Chapter 7

Calibrating and Validating Performance Models

Queuing theory has been used to model many systems, from simple single-server queuing stations such as a bank teller to complex distributed software systems such as web servers [22]. Here, we are interested in the use of TimeTrial to support model development and use. We explore this question by investigating a Jacksonian queuing network model of Mercury BLASTN deployed on a combination of CPUs and FPGAs. We compare performance predictions from the model to empirical measurements from the executing application.

The TimeTrial performance monitor is used to both calibrate and validate the performance model. We calibrate the model by measuring the input arrival rate, the service rates for each queuing server, and the branching probabilities in the queuing network topology. We validate the model by predicting server utilizations and queue occupancies and comparing the predictions to empirical measurements on two distinct input data sets. The material in this chapter is published in [35] and reflects a collaboration between the co-authors.

7.1 Performance Models for Streaming Applications

Figure 7.1 shows an example streaming application with two stages. If two compute resources are available, the two stages can be mapped to distinct resources and pipelining can be exploited to achieve parallel execution.

There are two commonly used approaches to modeling the performance of applications of this type: mean value models and queuing models. In a mean value model, the performance of each stage (e.g., achievable throughput, execution time for each input data element,

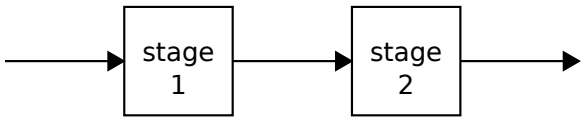


Figure 7.1: Streaming data application with two pipelined stages.

etc.) is characterized by its mean. Assuming the stages have been characterized by their individual throughput (inverting the per-input execution time if that is what is provided), the modeled throughput achievable by the entire pipeline is simply the minimum throughput achievable by each stage.

In a queuing model, the characterization of stage performance is expanded to include not only a mean but also a distribution. In this case, the "service rate" of the stage is modeled stochastically (commonly with a parametrized distribution) and the queuing of data elements in front of each stage is also included in the model. Figure 7.2 illustrates a 2-stage queuing network model of the 2-stage pipelined example application above. In the standard notation of queuing networks, λ represents the mean input arrival rate, μ_1 represents the mean service rate for stage 1, and μ_2 represents the mean service rate for stage 2.

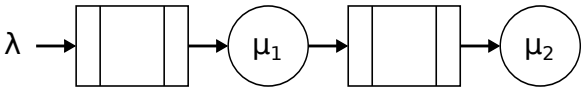


Figure 7.2: Queuing network model of pipelined application.

Inputs to a queuing model might include the mean service rates for each stage, and the queuing model then could be used to predict the queue occupancies and/or server utilizations for a range of possible input arrival rates. Queuing models of this type have been used to describe web servers [22], cyclic SPMD (single program, multiple data) applications executing on MIMD platforms [30], a long-lived transaction processing system for database management systems [73], stream-based sorting [89], as well as Mercury BLAST [34, 35].

Our purpose in this chapter is to investigate the use of TimeTrial for calibrating and validating models of this type. By calibration, we mean the extraction of model parameters that can be input into the performance model. In the example above, the input arrival rate and the two service rates would be considered model input parameters. We wish to measure them using TimeTrial so they can be provided as inputs to the performance model. The user can query these values from TimeTrial with queries of the form:

```
measure mean rate at Stage1.in
```

```
measure mean rate at Stage1.in when
    (occupancy at Stage1.in > 0 & !backpressure at Stage1.out)

measure mean rate at Stage2.in when
    (occupancy at Stage2.in > 0 & !backpressure at Stage2.out)
```

The first measure statement yields λ and the second two yield μ_1 and μ_2.

By validation, we mean the comparison of some prediction made by the model with measurements made by TimeTrial. This enables the model user to gather empirical evidence to assess the quality of the model. In our example, the queue occupancies and utilizations are readily monitored by TimeTrial:

```
measure mean occupancy at Stage1.in
measure mean occupancy at Stage2.in
measure mean util at Stage1.in
measure mean util at Stage2.in
```

While the above statements ask for the mean values of the queue occupancies, the user could alternatively ask for the histograms of the queue occupancies. In many queuing models, the full distribution of queue occupancy is predictable in the theory, and a Q-Q plot or Chi-square test could be used to compare the empirical histogram collected by TimeTrial with the theoretical distribution predicted by the queuing model.

The above concepts are next made more concrete by calibrating and validating a queuing model of the Mercury BLASTN application.

7.2 Modeling Mercury BLASTN

Here we focus on modeling Mercury BLASTN [19, 65], the accelerated BLASTN used in the previous chapter. Mercury BLASTN's three-stage pipeline is repeated in Figure 7.3 and a brief review is given here. In the first FPGA stage, BLASTN detects *seed matches*, which are exact substring matches of length 11 between the query and the database. Mercury BLASTN divides this stage's work into two parts: stage 1a, in which each database word is checked against on-chip Bloom filters [15] built from the query to eliminate most non-matching words, and stage 1b, in which the locations of matching query and database words, if any, are identified using an SRAM-based hash table.

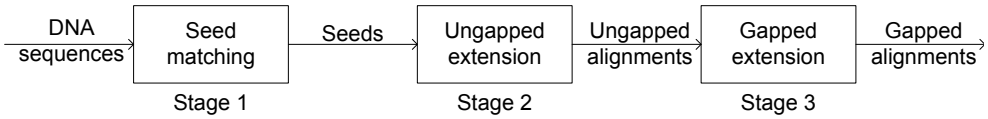


Figure 7.3: Mercury BLASTN.

Seed matches are forwarded to the second FPGA stage, *ungapped extension*. This stage checks whether each seed match is part of a larger *ungapped alignment*, i.e. a region in which the query and database differ by a small number of character substitutions. Ungapped alignments passing this stage are forwarded to the third stage, *gapped extension*, which runs in software. Gapped extension determines whether each seed match is part of an even larger region with small edit distance, this time permitting character substitutions, insertions, and deletions. Regions that pass this final test represent strong *gapped alignments* between query and database, which are reported to the user.

7.3 Queuing Theory Performance Model

Figure 7.4 shows a queuing network used to model Mercury BLASTN. The queuing network is Jacksonian [57, 58], meaning that the individual queuing stations are Markovian (Poisson arrival process with rate λ, exponentially distributed service times with rate μ) and the queues are assumed to have infinite capacity (an M/M/1 queuing model). It is worth pointing out here that the actual application exhibits none of these properties. Arrivals are not Poisson, service times are not exponential, and the physical queues are finite in capacity. Whether or not the distributions are crucial is one of the relevant questions addressed in this chapter.

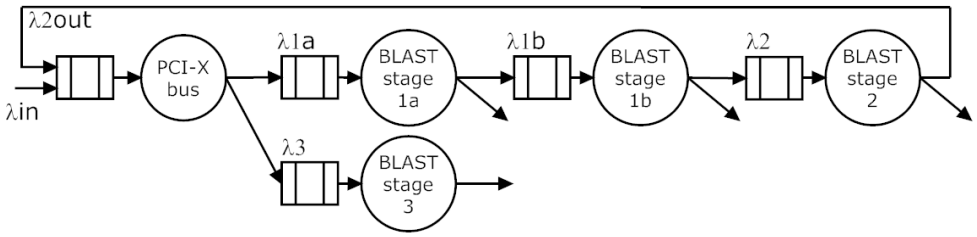


Figure 7.4: Queuing model for Mercury BLASTN.

The model is nominally composed of 5 queuing stations: the PCI-X bus that interconnects the processor and the FPGA, as well as BLASTN stages 1a, 1b, 2, and 3. Our true focus is on the portions of the application that are executed on the FPGA, so we will concentrate our attention on the servers modeling BLASTN stages 1a, 1b, and 2.

Starting from the left, λ_{in} is the rate at which DNA sequence bases are consumed. They are queued for delivery across the PCI-X bus and then delivered to stage 1a. As a result, $\lambda_{1a} = \lambda_{in}$. Stage 1a filters the stream of incoming bases into hits that are passed to stage 1b (with probability p_{1a1b}) or dropped (with probability $(1 - p_{1a1b})$). Stage 1b generates seeds that are passed to stage 2 (with probability p_{1b2}) or dropped (with probability $(1 - p_{1b2})$). Stage 2 subsequently generates alignments that are either passed to stage 3 via the PCI-X bus (with probability p_{23}) or dropped (with probability $(1 - p_{23})$). The above relates the set of λs by the following system of equations:

$$\begin{aligned} \lambda_{1a} &= \lambda_{in} \\ \lambda_{1b} &= p_{1a1b}\lambda_{1a} \\ \lambda_{2} &= p_{1b2}\lambda_{1b} \\ \lambda_{2out} &= p_{23}\lambda_{2} \\ \lambda_{3} &= \lambda_{2out}. \end{aligned}$$

Using classic results from queuing theory [62], the utilization for each server i is

$$\rho_i = \lambda_i/\mu_i,$$

the mean queue occupancy for queue i is

$$N_{Q,i} = \rho_i^2/(1 - \rho_i),$$

and the probability of n or more elements in station i is

$$P_i[N \geq n] = \rho_i^n.$$

7.4 Model Calibration and Validation

We use the TimeTrial performance monitor to make empirical measurements on the executing system. To calibrate the queuing model, we use TimeTrial to measure the input arrival rate (λ_{in}), the service rates of stages 1a, 1b, and 2 (μ_{1a}, μ_{1b}, and μ_2), and the branching probabilities (p_{1a1b}, p_{1b2}, and p_{23}). To validate the queuing model, we use TimeTrial to measure the server utilizations of stages 1a, 1b, and 2 (ρ_{1a}, ρ_{1b}, and ρ_2) and the queue

occupancies of the queues into stages 1b and 2 ($N_{Q,1b}$ and $N_{Q,2}$). To complete the validation, we compare the empirically measured server utilizations and queue occupancies to the predictions made by the model.

Each of the above measurements is made for 2 distinct test cases. The runs are as follows:

- Run 1: The first data set is the human chromosome 22 (from build 19 of the human genome) divided into 1,134 65,400-base segments as the query. The database consists of the 9th build of the mouse genome (2.7 GBases).
- Run 2: The second data set consists of comparing all the non-mammal vertebrate mRNA split into 8,608 65,400-base segments as the query. The queries were searched against all the mammal mRNA in the NCBI RefSeq repository (791 Mbases) as the database.

Table 7.1 gives the model inputs, while Table 7.2 provides a comparison between model predictions and empirical measurements. Note that the units for the service rates vary as one moves down the pipeline. Where needed, appropriate units conversions are incorporated into the model (e.g., DNA bases are encoded as 4 bits per base, so one byte of data transfer across the PCI-X bus delivers 2 bases to the input of stage 1a). In addition, the service rate for stage 1b is a non-linear function of whether or not the external memory port is saturated. Additional details of the model are described by Dor in [34].

Table 7.1: Input parameters to queuing model.

Parameter	Value for Run 1	Value for Run 2
λ_{in}	900 MB/s	720 MB/s
μ_{PCI}	1 GB/s	1 GB/s
μ_{1a}	2.1 Gbases/s	2.1 Gbases/s
μ_{1b}	130 Mseeds/s	50 Mseeds/s
μ_2	133 Maligns/s	133 Maligns/s
p_{1a1b}	0.018	0.035
p_{1b2}	0.88	0.76
p_{23}	0.0002	0.0004

Starting with the server utilization results, we observe that there is a close match in virtually every case between the model predictions and the empirical measurements. This is not surprising, as the server utilizations are generally based primarily on mean job flow rates and are fairly insensitive to the distributions. This implies that the distributional assumptions present in the M/M/1 queuing models will not inordinately impact the quality of the model.

Table 7.2: Model predictions vs. empirical measurements.

Parameter	Model Prediction	Empirical Measurement	Error
Run 1			
ρ_{1a}	0.84	0.85	0.01
ρ_{1b}	0.25	0.23	0.02
ρ_2	0.21	0.21	0
$N_{Q,1b}$	0.08	approx. 0	0.07
$N_{Q,2}$	0.06	1.2	1.1
Run 2			
ρ_{1a}	0.68	0.68	0
ρ_{1b}	0.999	0.93	0.07
ρ_2	.29	.29	0
$N_{Q,1b}$	7500	580	6900
$N_{Q,2}$	0.12	1.7	1.6

Turning to the queue occupancies, for 3 of 4 cases we again have a very close match between the model predictions and the empirical measurements. The one significant discrepancy is for the queue associated with stage 1b in run 2. Here, the high server utilization indicates that this server is the performance limiting bottleneck in the application. The physical queue is of length 600 entries, so the empirical queue occupancy cannot grow larger than that. The model predicts a much larger queue occupancy. Both the model and the empirical results are indicating that the queue will fill; however, the infinite queue capacity in the model is not capped by the length of the physical queue.

7.5 Chapter Summary

Here we have illustrated the use of straightforward queuing models to describe the performance of high performance streaming data applications and used TimeTrial to calibrate and validate this model. In general, they do surprisingly well at predicting the performance properties of the real application. Where there are discrepancies, the model can assist in understanding those discrepancies. For example, in the actual system, there is backpressure being asserted upstream of stage 1b due to the fact that the queue is full. While the notion of backpressure doesn't exist explicitly in the current queuing model, we can estimate it by asking the model for the probability that the queue occupancy is greater than the actual

capacity of the queue. For run 2 this gives us $P_{1b}[N \geq 600] = 0.92$, a likely event. Additional details on this proposed paradigm as well as its suitability for an M/M/1/K queuing model can be found in [34].

Chapter 8

Conclusions and Future Research

In this dissertation we have enabled developers to measure the performance of streaming, heterogeneous systems through the design and implementation our performance debugging tool, TimeTrial. We showed how TimeTrial can enable low-impact measurements across FPGA and multi-core processors. The TimeTrial measurement language provides a straightforward mechanism for developers to specify which aspects of a program are to be monitored and in what way they are to be monitored. Through integration with the Auto-Pipe system, developers can specify desired measurements and these measurements are automatically added to the resulting program. TimeTrial was demonstrated measuring two applications and providing helpful guidance where optimization efforts should be focused.

We now return to the research questions posed in Chapter 1 and describe how the work presented in this dissertation addresses those questions. Questions 1 and 2 asked the following:

> How can the performance of a distributed, streaming application be measured? What aspects of performance should be measured?
>
> How can the performance of an streaming application distributed across different FPGA accelerators and processor cores be measured?

TimeTrial's profiles of streaming applications focus on performance metrics that enable a developer to find the location of the bottleneck in the system. These metrics include throughput, latency or utilization. TimeTrial in its current form is able to measure properties of communication links and ports. Instead of deciding the specifics of what data should be collected for the developer, TimeTrial enables the developer to choose from a broad class of measurements to be included in the profile. To support heterogeneous application measurements, TimeTrial deploys a monitoring agent that is responsible for collecting data on each computation resource. The FPGA agents communicate the measurements back

to the software agent where a profile of the whole application is developed. Techniques to measure the performance of such an application deployed on FPGAs and processor cores were presented in Chapter 3.

Question 3 related to the ability to collect the desired information:

> What data needs to be collected in order to answer the above questions? Can this data be collected in a low-impact manner and still satisfactorily answer those questions? What techniques should be used?

TimeTrial monitors events on communication channels. By time-stamping and counting these events, the TimeTrial agents are able to transform these event streams into performance metrics. TimeTrial supports metrics that are both relevant to performance and relatively easy to compute. To further reduce impact, the developer can choose which metrics, how they are summarized and what portions of the application to measure. By letting the developer select the metrics of interest, overhead and impact can be reduced compared to measuring all metrics all the time. The results from profiling micro-benchmarks and real heterogeneous applications show that this approach has minimal impact on the executing application.

Dealing with pieces of the system that are not directly measurable, question 4 asked:

> How does one build a profile of resources that have limited inherent visibility (e.g. system buses)?

A specific example of such a resource, a PCI-X bus, was shown to be able to be profiled with TimeTrial. The approach taken was to measure the pieces of the system that were available, measure properties of the traffic on the ingest and egress of the bus, and use a stochastic simulation of the bus to recreate the queue occupancy. This technique was validated against a micro-benchmark that allowed the true queue occupancy to be replayed. This approach proved to give good qualitative insight into the performance of that resource. Finally, a mechanism was described that allowed the developer to test when the model assumptions had failed and the simulation result should be discarded.

The final question relates to whether TimeTrial can be used to support modeling:

> How does one use the collected performance data to calibrate or validate performance models?

We addressed this question by developing a strategy to use TimeTrial to calibrate a performance model and then validate its performance on an application. We illustrated its effectiveness validating a straightforward queueing model. More robust models could be handled in a similar manner.

In conclusion, TimeTrial provides an automated measurement system for understanding the runtime performance of streaming, heterogeneous applications with little impact on those applications.

8.1 Future Research

A number of challenges remain to be addressed and we list some of the more interesting ones here.

Extend support to GPUs. Supporting GPUs would allow measurement of applications that utilize FPGAs, GPUs and multi-core processors together. The design approach in TimeTrial makes it relatively easy to address GPUs. A new TimeTrial agent would have to be written to support the new architecture. Then, the agent could be deployed on each GPU in a similar manner to the software or FPGA agents and metrics could be computed online using a subset of GPU cores.

Implement conditionals in the TimeTrial compiler. An extension of the current implementation is enabling the Auto-Pipe compiler to support conditional measurement as well as assert statements. Doing so would open up opportunities for more optimizations when adding instrumentation. For instance, adding online support for assertion checking would allow the TimeTrial system to further lower its impact since less data would have to be logged and communicated between computational resources. In addition, conditional measurements could be used to calibrate a performance model more precisely and assertions could validate the model under different circumstances.

Improve fidelity of virtual queue measurements. For more detailed virtual queue profiling, one might develop a state-space Markovian model of Figure 5.7's tandem queueing network with a bounded capacity queue between the two servers. This has the potential to provide an analytic solution, which might obviate the need to perform stochastic simulation. Even if a closed-form solution to the analytic model is unavailable, a numeric solution can be less expensive than a stochastic discrete-event simulation run. Alternatively, approximate solution methods for the bounded capacity queue with upstream blocking do exist (see,

e.g., [66] and [91]), which can also be less expensive to compute than simulation. Further extensions might include automated queue modeling and validation of the model through TimeTrial measurements.

Automate performance model generation. An interesting open research area is automatically generating lightweight, empirically-parametrized performance models. TimeTrial can already collect much of the relevant parameters. By integrating it with a model generator, developers could spend less time developing their own models and more time analyzing the model and optimizing the application.

Automate bottleneck detection. Extending TimeTrial to automatically find the performance limiting segments of the application could be useful. One could approach this by implementing some techniques described in the research literature.

Extend automated profiling targets. Currently, TimeTrial focuses on profiling an application through its communication. With a non-trivial amount of implementation effort, it could be extended to measure the performance within blocks. This is non-trivial since the vision of using a co-ordination language is to use the designers preferred language for block implementations. Hence, many languages would have to be parsed and instrumented for this to become a reality. Another approach for software (and potentially GPU) blocks is to integrate the analysis from available tools (e.g. TAU) into TimeTrial's performance profile.

References

[1] Mohamed Abouellail, Esam El-Araby, Mohamed Taher, Tarek El-Ghazawi, and Gregory B. Newby. DNA and protein sequence alignment with high performance reconfigurable systems. In *Proc. of 2nd NASA/ESA Conf. on Adaptive Hardware and Systems*, pages 334–341, 2007.

[2] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[3] V. Aggarwal, R. Garcia, G. Stitt, A. George, and H. Lam. SCF: A device- and language-independent task coordination framework for reconfigurable, heterogeneous systems. In *Proc. of 3rd Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 19–28, November 2009.

[4] V. Aggarwal, A. George, K. Yalamanchili, C. Yoon, H. Lam, and G. Stitt. Bridging parallel and reconfigurable computing with multilevel PGAS and SHMEM+. In *Proc. of 3rd Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 47–54, November 2009.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[6] D. L. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA-CPU computation components: A missing link. *IEEE Micro*, 24(4):42–53, 2004.

[7] Annapolis Micro Systems, Inc. `http://www.annapmicro.com`.

[8] Lilian Atieno, Jonathan Allen, Dennis Goeckel, and Russell Tessier. An adaptive Reed-Solomon errors-and-erasures decoder. In *Proc. of Int'l Symposium on Field Programmable Gate Arrays*, pages 150–158, February 2006.

[9] Zachary K. Baker and Viktor K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proc. of 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

[10] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Robert A. Ballance and Jonathan Cook. Monitoring MPI programs for performance characterization and management control. In *Proc. of ACM Symp. on Applied Computing*, pages 2305–2310, 2010.

[12] Michela Becchi, Mark Franklin, and Patrick Crowley. Performance/area efficiency in embedded chip multiprocessors with micro-caches. In *Proc. of ACM Int'l Conf. on Computing Frontiers*, May 2007.

[13] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. of 37th IEEE/ACM Int'l Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Peter Bellows. High-visibility debug-by-design for FPGA platforms. *J. Supercomput.*, 32(2):105–118, 2005.

[15] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, May 1970.

[16] Michael D. Bond and Kathryn S. McKinley. Continuous path and edge profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140, Washington, DC, USA, 2005. IEEE Computer Society.

[17] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *Proc. ACM/IEEE Supercomputing Conference*, 2000.

[18] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[19] Jeremy D. Buhler, Joseph M. Lancaster, Arpith C. Jacob, and Roger D. Chamberlain. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In *Proc. of Reconfigurable Systems Summer Institute*, July 2007.

[20] W. H. Burge. Stream processing functions. *IBM J. Res. Dev.*, 19:12–25, January 1975.

[21] Doug Burger, Todd M. Austin, and Stephen W. Keckler. Recent extensions to the simplescalar tool suite. *SIGMETRICS Perform. Eval. Rev.*, 31:4–7, March 2004.

[22] Giuliano Casale, Mi Ningfang, and Evgenia Smirni. Versatile models of systems using map queueing networks. In *IEEE Int'l Parallel and Distributed Processing Symp.*, Apr. 2008.

[23] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, E.F. Berkley Shands, and Naveen Singla. Auto-Pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, March 2010.

[24] Roger D. Chamberlain, Joseph M. Lancaster, and Ron K. Cytron. Visions for application development on hybrid computing systems. *Parallel Computing*, 34(4-5):201–216, May 2008.

[25] Roger D. Chamberlain, Eric J. Tyson, Saurabh Gayen, Mark A. Franklin, Jeremy Buhler, Patrick Crowley, and James Buckley. Application development on hybrid systems. In *Proc. of ACM/IEEE Supercomputing Conf.*, November 2007.

[26] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. of Int'l Symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, 2005.

[28] Wang Chen, Panos Kosmas, Miriam Leeser, and Carey Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proc. of Int'l Symposium on Field Programmable Gate Arrays*, pages 213–222, February 2004.

[29] ClearSpeed Technology plc. `http://www.clearspeed.com`.

[30] Paolo Cremonesi and Claudio Gennaro. Integrated Performance Models for SPMD Applications and MIMD Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1320–1332, December 2002.

[31] John Curreri, Seth Koehler, Alan D. George, Brian Holland, and Rafael Garcia. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst.*, 3(1):5:1–5:23, January 2010.

[32] Siddhartha Datta, Parag Beeraka, and Ron Sass. RCBLASTn: Implementation and evaluation of the BLASTn scan function. In *Proc. of 16th Int'l Symp. on Field-Programmable Custom Computing Machines*, April 2009.

[33] Ryan A. DeVille, Ian A. Troxel, and Alan D. George. Performance monitoring for run-time management of reconfigurable devices. In *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms*, June 2005.

[34] Rahav Dor. Against all probabilities: A modeling paradigm for streaming applications that goes against common notions. Technical Report WUCSE-2010-30, Dept. of Computer Science and Engineering, Washington University in St. Louis, 2010.

[35] Rahav Dor, Joseph M. Lancaster, Mark A. Franklin, Jeremy Buhler, and Roger D. Chamberlain. Using queuing theory to model streaming applications. In *Proc. of Symp. on Application Accelerators in High Performance Computing*, July 2010.

[36] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, February 2008.

[37] S. J. Farlow. *Partial Differential Equations for Scientists and Engineers*. Dover Publications, 1993.

[38] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proc. of the ACM Conf. on Graphics Hardware*, pages 133–137, 2004.

[39] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, 2001.

[40] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.

[41] N. Galoppo et al. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. of the ACM/IEEE Supercomputing Conf.*, pages 3–14, 2005.

[42] Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, and Roger D. Chamberlain. A federated simulation environment for hybrid systems. In *Proc. of 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, pages 198–207, June 2007.

[43] Saeed Ghanbari, Gokul Soundararajan, and Cristiana Amza. A query language and runtime tool for evaluating behavior of multi-tier servers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 131–142, New York, NY, USA, 2010. ACM.

[44] N. Govindaraju et al. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of SIGMOD Int'l Conf. on Management of Data*, pages 325–336, 2006.

[45] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proc. of SIGPLAN Symp. on Compiler Construction*, pages 120–126, 1982.

[46] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, and A. George. Hardware/software interface for high-performance space computing with FPGA coprocessors. In *Proc. of IEEE Aerospace Conference*, March 2006.

[47] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. of 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, 2008.

[48] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proc. of Int'l Symposium on Field Programmable Gate Arrays*, pages 162–170, February 2004.

[49] Martin C. Herbordt, Josh Model, Yongfeng Gu, Bharat Sukhwani, and Tom VanCourt. Single pass, BLAST-like, approximate string matching on FPGAs. In *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 217–226, 2006.

[50] M.C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and Tom VanCourt. Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33:741–756, 2007.

[51] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 1997.

[52] D.R. Horn et al. ClawHMMER: A streaming HMMer-search implementation. In *Proc. of ACM/IEEE Supercomputing Conf.*, pages 11–19, 2005.

[53] Richard Hough, Praveen Krishnamurthy, Roger D. Chamberlain, Ron K. Cytron, John Lockwood, and Jason Fritts. Empirical performance assessment using soft-core processors on reconfigurable hardware. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[54] Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell. Exploring the cache design space for large scale CMPs. *SIGARCH Comput. Archit. News*, 33(4):24–33, 2005.

[55] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proc. of 19th ACM Int'l Conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM Press.

[56] Intel Corporation. Using Intel's VTune Counter Monitor, January 2005.

[57] J. Jackson. Network of waiting lines. *Management Science*, 5(4):518–521, 1957.

[58] J. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[59] A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain. FPGA-accelerated seed generation in Mercury BLASTP. In *Proc. of 15th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[60] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[61] Hari Kannan, Fei Guo, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to QoS: Case studies in CMP resource management. In *Proceedings of the 2nd Workshop on Design, Architecture, and Simulation of Chip-Multiprocessors*, December 2006.

[62] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley & Sons, 1975.

[63] Gernot H. Koch, W. Rosenstiel, and U. Kebschull. Breakpoints and breakpoint detection in source-level emulation. *ACM Trans. Des. Autom. Electron. Syst.*, 3(2):209–230, 1998.

[64] Seth Koehler, John Curreri, and Alan D. George. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parallel Computing*, 34(4-5):217–230, May 2008.

[65] Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, Arpith Jacob, and Joseph Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 49(1):101–121, October 2007.

[66] Praveen Krishnamurthy and Roger D. Chamberlain. Analytic performance models for bounded queueing systems. In *Proc. of Workshop on Advances of Parallel and Distributed Computing Models*, April 2008.

[67] Joseph M. Lancaster, Jeremy Buhler, and Roger D. Chamberlain. Acceleration of ungapped extension in Mercury BLAST. *Microprocessors and Microsystems - Embedded Hardware Design*, 33(4):281–289, 2009.

[68] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[69] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[70] Miriam Leeser, Shawn Miller, and Haiqian Yu. Smart camera based on reconfigurable hardware enables diverse real-time applications. In *Proc. of 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 147–155, 2004.

[71] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[72] Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain, and Joseph M. Lancaster. Deadlock-avoidance for streaming applications with split-join structure: Two case studies. In *IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 333–336, July 2010.

[73] Deron Liang and Satish K. Tripathi. Performance Analysis of Long-lived Transaction Processing Systems with Rollbacks and Aborts. *IEEE Trans. Knowl. Data Eng.*, 8(5):802–815, October 1996.

[74] J. D. C. Little. A proof for the queueing formula: $L = \lambda W$. *Operations Research*, 9:383–387, May–June 1961.

[75] Aqeel Mahesri, Nicholas J. Wang, and Sanjay J. Patel. Hardware support for software controlled multithreading. In *Proceedings of the 2nd Workshop on Design, Architecture, and Simulation of Chip-Multiprocessors*, December 2006.

[76] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[77] Mercury Computer Systems, Inc. `http://www.mc.com`.

[78] Aleksandar Milenković and Milena Milenković. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Trans. Model. Comput. Simul.*, 17, January 2007.

[79] Aleksandar Milenković and Milena Milenković. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Trans. Model. Comput. Simul.*, 17, January 2007.

[80] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37 –46, nov 1995.

[81] Nallatech Ltd. `http://www.nallatech.com`.

[82] National Center for Biotechnology Information. Growth of GenBank, 2002. `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html`.

[83] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42:89–100, June 2007.

[84] Ryan Newton, Lewis Girod, Michael Craig, Samuel Madden, and Greg Morrisett. Wavescript: A case-study in applying a distributed stream-processing language. Technical Report MIT-CSAIL-TR-2008-005, MIT CSAIL, January 2008.

[85] D. Nunes, M. Saldana, and P. Chow. A profiler for a heterogeneous multi-core multi-FPGA system. In *Proc. of Int'l Conf. on Field Programmable Technology*, pages 113–120, December 2008.

[86] Technical Brief: Nvidia GeForce 8800 GPU Architecture Overview. `http://www.nvidia.com/object/IO_37100.html`.

[87] Open Source Community. mpiP: Lightweight, Scalable MPI Profiling, 2011. `http://mpip.sourceforge.net`.

[88] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[89] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Design-space optimization for automatic acceleration of streaming applications. In *Symp. on Application Accelerators in High Performance Computing*, July 2010.

[90] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrière, and Eric Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079 – 2088, 2010. ICCS 2010.

[91] H.G. Perros and T. Altiok. Approximate analysis of open networks of queues with blocking: Tandem configurations. *IEEE Trans. Soft. Eng.*, 12:450–461, 1986.

[92] J. F. Reynolds. A proof of the random-walk method for solving Laplace's equation in 2-D. *The Mathematical Gazette*, 49(370):416–420, December 1965.

[93] E. Roesler and B. Nelson. Debug methods for hybrid CPU/FPGA systems. In *Proc. of IEEE Int'l Conference on Field-Programmable Technology*, pages 243–251, December 2002.

[94] Tobias Schumacher, Christian Plessl, and Marco Platzner. IMORC: Application mapping, monitoring and optimization for high-performance reconfigurable computing. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 275–278, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[95] Ronald Scrofano, Maya Gokhale, Frans Trouw, and Viktor K. Prasanna. Hardware/software approach to molecular dynamics on reconfigurable computers. In *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 23–34, 2006.

[96] Silicon Graphics, Inc. `http://www.sgi.com`.

[97] Anahita Shayesteh, Glenn Reinman, Norman Jouppi, Suleyman Sair, and Tim Sherwood. Dynamically configurable shared CMP helper engines for improved performance. *SIGARCH Comput. Archit. News*, 33(4):70–79, 2005.

[98] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[99] SRC Computers, Inc. `http://www.srccomputers.com`.

[100] Christoph Steigner and Jorgen Wilke. Performance tuning of distributed applications with CoSMoS. In *Proc. of International Conference on Distributed Computing Systems*, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[101] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[102] W. A. Strauss. *Partial Differential Equations: An Introduction*. Wiley, 1992.

[103] Bassam Tabbara, Enrica Filippi, and Luciano Lavagno. Fast hardware-software co-simulation using VHDL models. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 64–71, New York, NY, USA, 1999. ACM Press.

[104] Tensilica Inc. `http://www.tensilica.com`.

[105] Justin Thiel and Ron K. Cytron. Splice: A standardized peripheral logic and interface creation engine. In *Proc. of Reconfigurable Architectures Workshop*, March 2007.

[106] W. Thies, M. Karczmarek, and S.P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of 11th Int'l Conf. on Compiler Construction*, pages 179–196, 2002.

[107] J.G. Tong and M.A.S. Khalid. Profiling CAD tools: A proposed classification. In *Proc. of International Conference on Microelectronics*, pages 253–256, December 2007.

[108] Justin L. Tripp, Henning S. Mortveit, Anders A. Hansson, and Maya Gokhale. Metropolitan road traffic simulation on FPGAs. In *Proc. of 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 117–126, 2005.

[109] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proc. of 18th ACM/SIGDA Int'l Symp. of Field Programmable Gate Arrays*, pages 115–124, February 2010.

[110] Eric J. Tyson, James Buckley, Mark A. Franklin, and Roger D. Chamberlain. Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system. *Nuclear Inst. and Methods in Physics Research A*, 585(2):474–479, October 2008.

[111] Keith D. Underwood, K. Scott Hemmert, and Craig D. Ulmer. From silicon to science: The long road to production reconfigurable supercomputing. *ACM Trans. Reconfigurable Technol. Syst.*, 2(4):26:1–26:15, September 2009.

[112] Neil Vachharajani, Matthew Iyer, Chinmay Ashok, Manish Vachharajani, David I. August, and Daniel Connors. Chip multi-processor scalability for single-threaded applications. *SIGARCH Comput. Archit. News*, 33(4):44–53, 2005.

[113] Jeffrey S. Vetter and Patrick H. Worley. Asserting performance expectations. In *Proc. of ACM/IEEE Supercomputing Conference*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[114] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language.* Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[115] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. Advanced components in the variable precision floating-point library. In *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–258, 2006.

[116] Rick Weber, Akila Gothandaraman, Robert J. Hinde, and Gregory D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *IEEE Trans. on Parallel and Distributed Systems*, 22(1):58–68, January 2011.

[117] T.C. Weekes et al. VERITAS: the very energetic radiation imaging telescope array system. *Astroparticle Physics*, 17(2):221–243, May 2002.

[118] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, November 1977.

[119] Tilman Wolf and Mark A. Franklin. Performance models for network processor design. *IEEE Trans. on Parallel and Distributed Systems*, 17(6):548–561, June 2006.

[120] XtremeData, Inc. `http://www.xtremedatainc.com`.

[121] David Zaretsky, Gaurav Mittal, Xiaoyong Tang, and Prith Banerjee. Overview of the FREEDOM compiler for mapping DSP software to FPGAs. In *Proc. of 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 37–46, 2004.

Low-Impact profiling of heterogeneous applications, Lancaster, Ph.D. 2011