Asking for Performance: Exploiting Developer Intuition to Guide Instrumentation with TimeTrial

Joseph M. Lancaster,
Joseph G. Wingbermuehle
Roger D. Chamberlain

Joseph M. Lancaster, Joseph G. Wingbermuehle, and Roger D. Chamberlain, “Asking for Performance: Exploiting Developer Intuition to Guide Instrumentation with TimeTrial,” in *Proc. of 13th International Conference on High Performance Computing and Communications* (HPCC), Sept. 2011.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Asking for Performance: Exploiting Developer Intuition to Guide Instrumentation with TimeTrial

Joseph M. Lancaster, Joseph G. Wingbermuehle, and Roger D. Chamberlain
Dept. of Computer Science and Engineering, Washington University in St. Louis
{lancaster,wingbej,roger}@wustl.edu

***Abstract*—Architecturally-diverse systems (containing co-processors such as reconfigurable logic and graphics engines) have received significant attention recently in the high performance computing community. They are new enough, however, that application development tools are quite limited. This paper describes our performance measurement system, TimeTrial, that automates performance measurements of diverse, streaming data applications. TimeTrial enables low-impact measurements by interpreting performance queries in the TimeTrial language and by compiling these queries to highly-specific, optimized instrumentation that aggressively performs lossy compression on the performance meta-data. Currently, TimeTrial supports multi-core processors and reconfigurable logic, with GPU support under development. We present the TimeTrial language and its associated compiler, including its use in optimizing the performance of an example computational science problem.**

I. Introduction

The use of non-traditional architectures for high-performance computing has received significant attention recently. Traditional multi-core processors are frequently augmented with co-processors that are tasked with executing performance-critical portions of an application. These co-processors might be field-programmable gate arrays (FPGAs) [9], graphics processing units (GPUs) [21], or hybrid processors such as the Cell [14]. We refer to them collectively as architecturally diverse systems.

Authoring applications for architecturally diverse 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, complicated by the fact that co-processors typically have poor internal visibility. Third, the application must be decomposed into components that both execute and properly interface with each other across the diverse platforms. 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. [30] 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.

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.

In our work, we focus on the class of streaming data applications. Examples of concurrency platforms that support the streaming data paradigm include Auto-Pipe [4], Brook [2], StreamIt [28], Wavescript [19], and Streamware [15] (see [25] for a survey of older streaming languages). The streaming data paradigm can be characterized by computation that is decomposed into isolated kernels (or blocks) with communication between kernels via explicit channels. Data is therefore "streamed" from one kernel to another over these channels. An example application is illustrated in Figure 1. Pseudo-random numbers are are generated in the `RNG` block and divided into two streams by the `Split` block. The two `Walk` blocks perform a random walk solution to Laplace's equation, the results of which are passed downstream to the `Avg` and `Print` blocks.

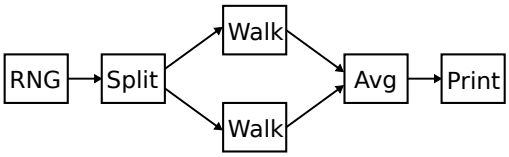

Fig. 1. An example streaming implementation of a Monte Carlo solution to Laplace's equation. Computation is performed within each block, communication is one-way along edges.

There are several benefits to authoring applications using this approach [6]: (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).

Obtaining peak performance is paramount when utilizing architecturally diverse systems (otherwise there is little gained from the increased design complexity). Since FPGAs offer little visibility into the design once deployed, application

authors are left without even the most basic information on the performance of their application. Traditional software tools such as `gprof` [13] and TAU [23] tend to be of marginal benefit because they are processor-centric and do not support meaningful metrics for architecturally diverse platforms (e.g., processor core → 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.

To address these concerns, we have created 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 application-specific 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 datasets.

This paper presents TimeTrial as incorporated with the Auto-Pipe [4] concurrency platform, enabling users to profile their streaming data applications by posing performance questions of the system. For example, a stream profile should be able to answer questions of the type:

- At what rate is data moving across the link that connects the `RNG` block to the `Split` block?
- Are the data rates balanced between the upper and lower branches of the application topology?
- What is the occupancy of the queues between each block?
- What fraction of the time is the `Avg` unable to continue processing because the queue into the `Print` block is full?
- What portion of the pipeline is limiting the achievable throughput?
- If that bottleneck were resolved, what would be the next bottleneck?

We will return to each of these questions below, describing how the TimeTrial performance monitoring system enables an application developer to learn the answers.

II. Background and Related Work

The TimeTrial language and compiler work with and complement the Auto-Pipe application development environment [4]. The Auto-Pipe tool set supports the development of streaming data applications deployed on architecturally diverse systems. An application's topology is described in the X coordination language [11] and individual block implementations are constructed using languages appropriate for the target execution platform (e.g., C/C++ for multi-core processors, VHDL/Verilog for FPGAs). Auto-Pipe has been used to develop applications over a wide range of fields, including astrophysics [29], cryptography [3], and computational finance [24]. While Auto-Pipe already supports performance evaluation via simulation [12], performance profiling of executing applications is a capability that did not previously exist in the Auto-Pipe environment.

Performance profiling of multi-threaded applications executing on multi-core processors is an active area of research, in large part due to recent increases in the number of cores per chip [27]. Stack-pointer tracing and instrumentation for path profiling are popular approaches. Message logging is also common in the MPI world [1]. However, techniques for profiling the performance of parallel applications executing on multi-core processors combined with accelerators have not received as much attention.

Analyzing and understanding the performance of a streaming application requires a different approach than that used by current tools. Profiling tools such as `gprof` [13] instrument a binary to log procedure calls. Such instrumentation can add an unacceptable overhead because the function calls sending and receiving data are likely to be frequently called, slowing down the application. `gprof` treats the portions of the application deployed on the accelerator as a black box since it has no support for measuring accelerator internal performance. In the worst case, `gprof` will provide an ambiguous picture of performance while adding unacceptable overhead. Other tools such as TAU [23] can reduce the overhead of profiling by throttling and by instrumenting only a subset of functions. These optimizations reduce overhead but do little to improve the scope of the performance picture obtained.

A performance-monitoring system for FPGAs has been developed by Koehler et al. at the Univ. of Florida [16]. This monitoring system is capable of counting events on arbitrary signals within a VHDL design and reporting these counts at a configurable frequency to the user. In contrast, TimeTrial maintains language independence by focusing on the edges between compute blocks rather than monitoring interior signals within a block. Second, TimeTrial monitors both the FPGA and multi-core processor portions of the deployed application to locate bottlenecks regardless of the processing resource on which they occur.

The system of Koehler et al. has recently been extended by Curreri et al. [7] 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. [8] explored the design of hardware probes for performance monitoring purposes in FPGA-deployed applications.

A performance profiler has also been developed for the TMD machine developed at the Univ. of Toronto [20]. This

profiler focuses on logging both MPI 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.

III. Architecture of the TimeTrial System

TimeTrial is made up of three major parts: the TimeTrial language, compiler, and agents. Statements in the language instruct the compiler "what and where" to measure, the compiler instruments the streaming application (including processor cores and FPGAs, if necessary), and the agents collect runtime statistics, logging them to disk. This section begins with an overview of the TimeTrial measurement system and then describes the language, compiler, and agents as integrated with the Auto-Pipe development environment and the X language.

Figure 2 shows a tandem streaming data application with TimeTrial instrumentation added. 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, is the responsibility of Auto-Pipe. For example, from block `A` to block `B` Auto-Pipe will instantiate a shared-memory buffer, while from block `B` to block `C` Auto-Pipe will invoke the appropriate low-level mechanisms to move data from software to the FPGA hardware.

On each computational resource, a TimeTrial agent is present 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 local agent 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.

A. TimeTrial 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. A proposed language for TimeTrial is published in [5].

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 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.

In the TimeTrial language, each statement directly corresponds to added instrumentation on one or more communication edges. We introduce the language using a few example queries. To ask the questions "What is the throughput of edge x?" and "How full is the queue on edge y?", one could write the following TimeTrial statements:

```
measure rate at x;
measure hist occupancy at y;
```

In both cases, the `measure` keyword instructs the TimeTrial compiler that we are asking for a measurement. All currently supported statements begin with the `measure` keyword. `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.

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

- `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

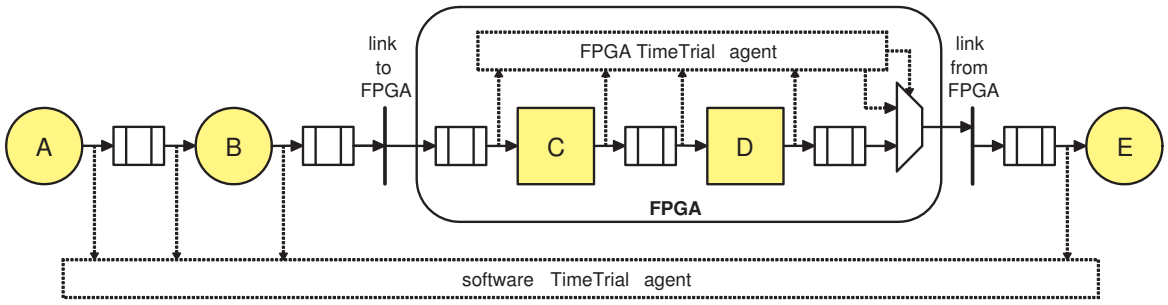

Fig. 2. An example streaming application mapped onto processor cores and an FPGA. The dotted lines represent TimeTrial instrumentation added to the application.

summarize that metric, the compiler can implement highly optimized, low-impact monitoring.

The following aggregation functions are supported:

- `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.

B. 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 [17]. To use the FPGA 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 agent to the software agent.

Edges between an FPGA block and a software block are made up of multiple tandem queues. There is a queue on the software side, a queue in the FPGA, and queues that reside at lower levels of the system, including the driver and FPGA bus interface. To allow visibility into these edges, the TimeTrial compiler instruments both the queue in software and the queue on the FPGA in a manner similar to the descriptions above. The queues that reside at lower levels are inaccessible, so they are not monitored. Comparing measurements from both the software and FPGA can give additional insight into the application's performance. For example, assume we have an FPGA block that feeds a software block. If the rate on the FPGA is high and the rate in software is low, we know that a queue is filling between the FPGA and software.

C. TimeTrial Agents

The TimeTrial system deploys two types of agents: a software agent that monitors taps from software portions of the application and an FPGA agent that monitors the FPGA portion. Both the software agent and FPGA agent aggregate event streams from taps over a period and log the results to disk. In TimeTrial, this period is referred to as a *frame* and is a configurable parameter representing either an interval of time or an event count.

Traditional performance tools such as `gprof` log or sample performance over the entire execution. 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. Figure 3 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 communicated to the software agent 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.

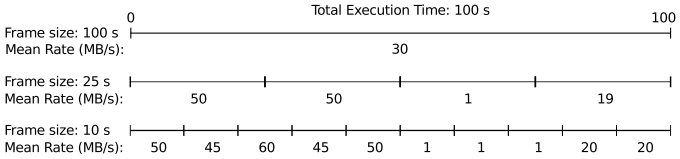

Fig. 3. Illustration of measuring with varying frame periods. Each frame results in one aggregated metric.

The TimeTrial compiler also instantiates an FPGA agent on each FPGA device where measurements are to be taken. Figure 4 shows an overview of the architecture of the FPGA agent. 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. 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.

The software TimeTrial agent is a separate process with which the application processes communicate using shared memory queues. Figure 5 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.

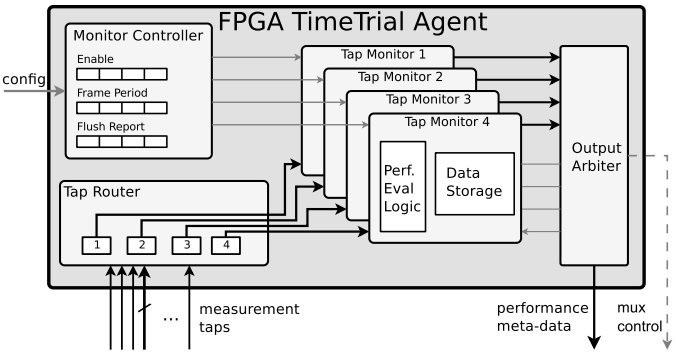

Fig. 4. Overview of the TimeTrial agent for the FPGA [17]. The FPGA agent is a high-speed, parameterized circuit designed to aggregate measurements on the FPGA.

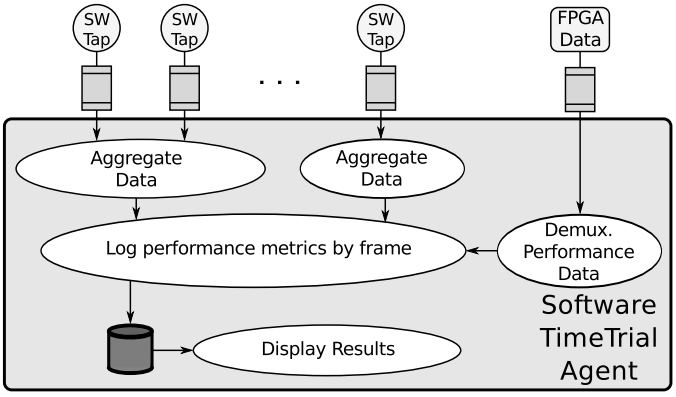

Fig. 5. Overview of the software TimeTrial agent. In addition to aggregating event streams online, the software agent is responsible for logging results from the FPGA agents to disk.

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 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 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. For instance, the graphs in this paper were produced using Mathematica.

IV. Using TimeTrial

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) [26] 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 [10]. There are several approaches for numerical solutions. One approach is Gauss-Seidel iterations [26], which converge quickly but require that the complete grid be stored in memory. Another method is Monte Carlo simulation, which is provably correct [22] 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 shows the pipeline 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.

The application works as follows. Pseudo-random numbers are generated (using the Mersenne twister algorithm [18]) in block `mt`. Block `s1` splits the stream of pseudo-random

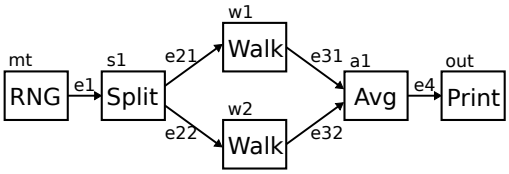

Fig. 6. Implementation of Monte Carlo solver for Laplace's equation.

numbers for use by two copies of a `Walk` block (called `w1` and `w2`) that perform a random walk executing the Monte Carlo solution [22]. 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 7 illustrates 4 `Walk` blocks).

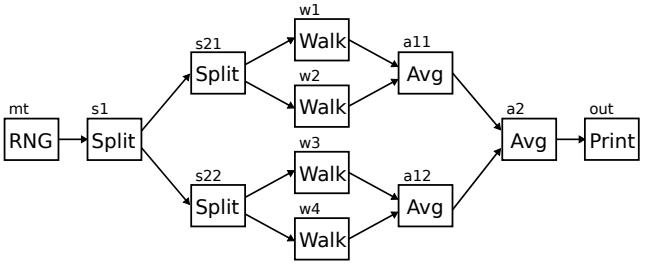

Fig. 7. Application topology for 4 independent random walks.

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 8.

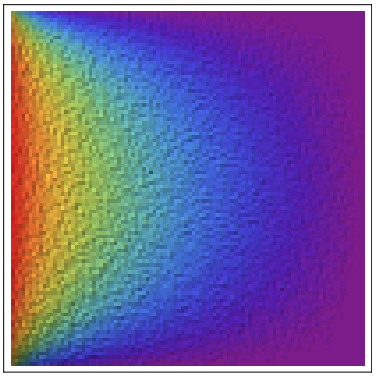

Fig. 8. Output from example 2-D temperature surface.

The experiments that follow are all executed on a system that has 2 six-core AMD processors (for a total of 12 cores) and an FPGA board connected via a PCI-X bus. Unless otherwise described, each block is mapped to a distinct core

(using processor affinity) and the software TimeTrial agent is mapped to a core without an application block.

We now reconsider the performance questions posed in the introduction. 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 9. 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 paper). 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 $1.5\times$ the inter-quartile range) are indicated by points on the graph. TimeTrial expresses throughput in terms of data element transfers per second. On edge `e1`, a data element is a single random number.

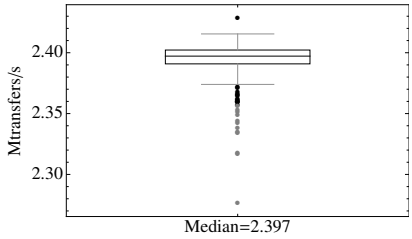

Fig. 9. Data rate across edge `e1`.

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 9, 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 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 10. In the histograms, the occupancy (horizontal axis) is in units of queue slots (data elements) and the percent time (vertical axis) is the time-weighted fraction of the total execution at each occupancy. 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 11. 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 (since replacing it with a faster implementation provided a greater than $3\times$ performance gain) and enable us to answer question six:

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

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 12. 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 timeline 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`.

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

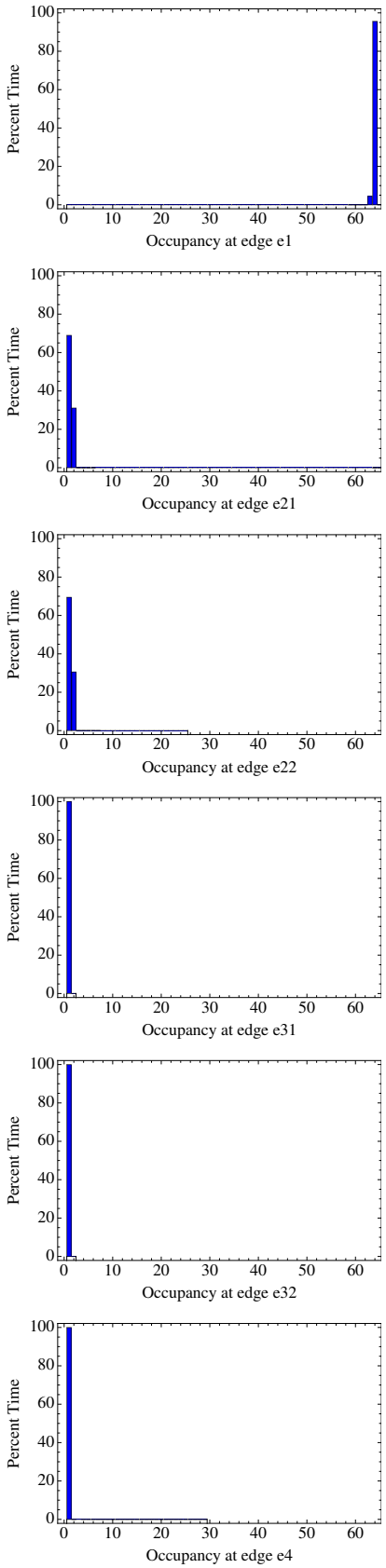

Fig. 10. Histograms of queue occupancies.

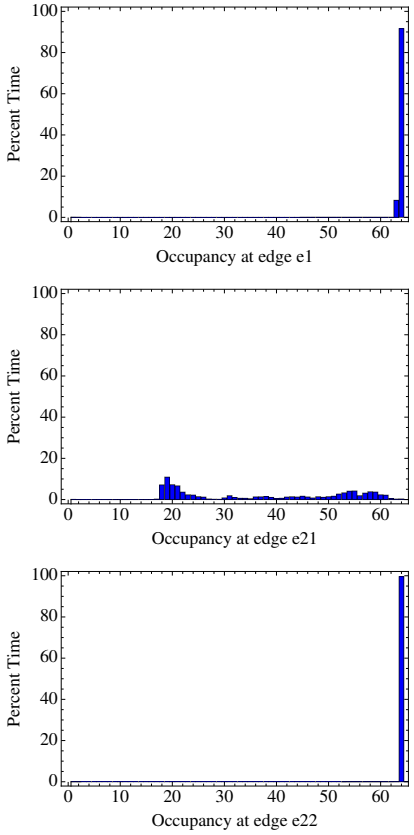

Fig. 11. Histograms of queue occupancies after replacing `s1` with a more efficient implementation.

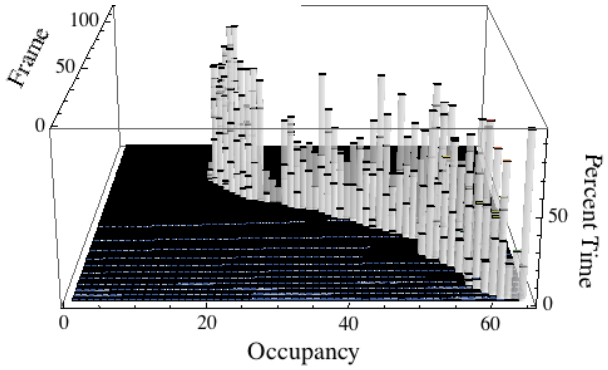

Fig. 12. Histogram of queue occupancy over time for edge `e21` (input to block `w1`).

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 software TimeTrial 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 13. 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.

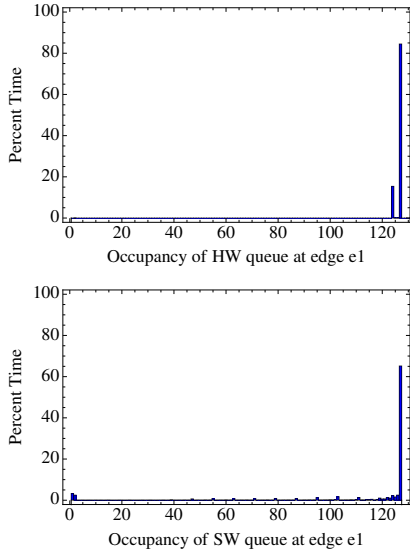

Fig. 13. Histograms of hardware and software queue occupancies for edge `e1` (output of the `RNG` block).

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 14 shows the throughput rate and Figure 15 shows 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).

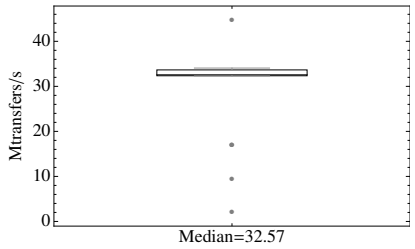

Fig. 14. Data rate across edge `e1` (output of the `RNG` block).

For this run, the `Split` blocks are once again the performance bottleneck. This is confirmed in Figure 16. 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 (i.e., the `RNG` block alone on the FPGA) the architectural diversity does not improve the performance

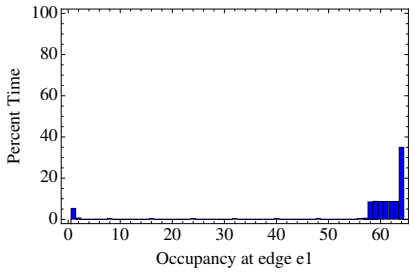

Fig. 15. Histogram of queue occupancy for edge `e1` (output of the `RNG` block).

of the application, this fact was not at all 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.

To assess the overhead that TimeTrial introduces, the experiments described above were executed both with and without the TimeTrial measurements present. The worst case increase in execution time was less than 2%, providing strong evidence that TimeTrial is effective at having a low impact on the performance of the measured application.

V. Conclusions and Future Work

The TimeTrial performance monitor provides low-impact assessment of streaming data computations guided by user input. The TimeTrial language enables users to straightforwardly describe the measurements that are desired, and the associated compiler and TimeTrial agents instrument and monitor the executing application.

TimeTrial imposes low overhead on the executing application, with the execution times for the experiments reported here being impacted by less than 2%. This is accomplished by ensuring that the software agent is allocated to a distinct processor core (i.e., not co-located on a core that is executing application code). In addition, the volume of performance meta-data that crosses the chip boundaries between FPGA and multi-core processor is limited by the aggressive data compression that can be performed by the FPGA agent.

While TimeTrial is complementary to the Auto-Pipe development environment, its design is not limited to Auto-Pipe. Any concurrency platform that supports the streaming data paradigm can use TimeTrial, as long as edges can be instrumented in the concurrency platform's runtime system.

Our continuing work is focused on enabling TimeTrial to reason more completely about application edges that cross platform boundaries. While the current TimeTrial monitors the queues at both ends of the edge (both in hardware and in software), we are investigating techniques that will allow the user to treat the edge (and its associated collection of physical queues) as a single virtual queue. This would have the benefit of further simplifying application performance understanding for users of TimeTrial.

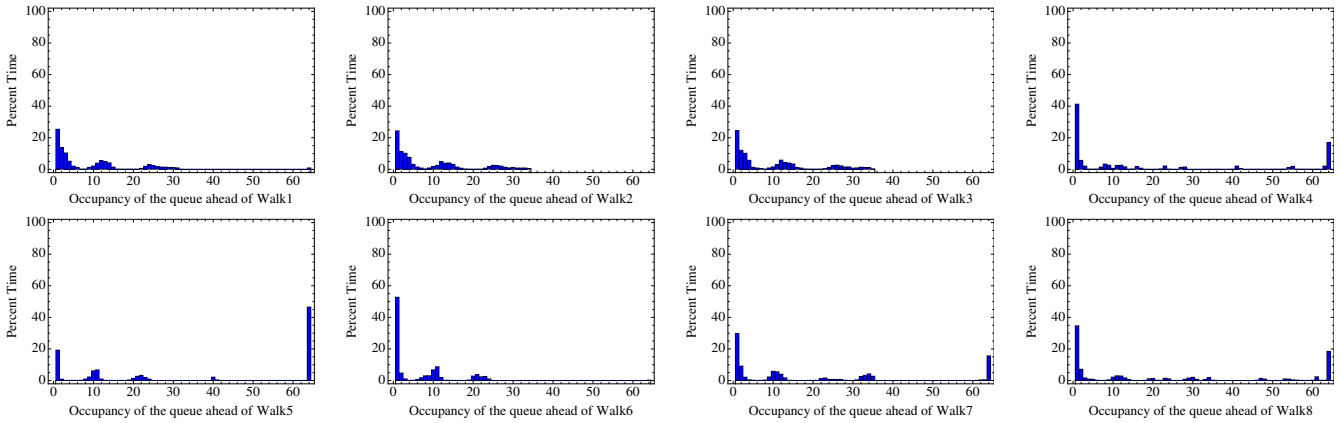

Fig. 16. Occupancies for the input queues to the walk blocks

ACKNOWLEDGMENT

This research was supported by NSF through grants CNS-0751212, CNS-0905368, and CNS-0931693.

REFERENCES

[1] R. A. Ballance and J. Cook, "Monitoring MPI programs for performance characterization and management control," in *Proc. of ACM Symp. on Applied Computing*, Mar. 2010, pp. 2305–2310.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, and K. Fatahalian, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[3] R. D. Chamberlain, J. Buhler, M. A. Franklin, and J. H. Buckley, "Application-guided tool development for architecturally diverse computation," in *Proc. of ACM Symp. on Applied Computing*, Mar. 2010, pp. 496–501.

[4] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-Pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

[5] R. D. Chamberlain and J. M. Lancaster, "Better languages for more effective designing," in *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms*, Jul. 2010.

[6] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron, "Visions for application development on hybrid computing systems," *Parallel Computing*, vol. 34, no. 4-5, pp. 201–216, May 2008.

[7] J. Curreri, S. Koehler, A. D. George, B. Holland, and R. Garcia, "Performance analysis framework for high-level language applications in reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pp. 5:1–5:23, Jan. 2010.

[8] R. A. DeVille, I. A. Troxel, and A. D. George, "Performance monitoring for run-time management of reconfigurable devices," in *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms*, Jun. 2005.

[9] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The Promise of High-Performance Reconfigurable Computing," *Computer*, vol. 41, no. 2, pp. 69–76, Feb. 2008.

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

[11] 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.*, Apr. 2006.

[12] S. Gayen, E. J. Tyson, M. A. Franklin, and R. D. Chamberlain, "A federated simulation environment for hybrid systems," in *Proc. of 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, Jun. 2007, pp. 198–207.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proc. of SIGPLAN Symp. on Compiler Construction*, 1982, pp. 120–126.

[14] M. Gschwind, "The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor," *Int'l J. of Parallel Programming*, vol. 35, no. 3, Jun. 2007.

[15] J. Gummaraju, J. Coburn, Y. Turner, and M. 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*, 2008, pp. 297–307.

[16] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, May 2008.

[17] J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proc. of 22nd IEEE Int'l System-on-Chip Conf.*, Sep. 2009, pp. 23–28.

[18] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[19] R. Newton, L. Girod, M. Craig, S. Madden, and G. Morrisett, "Wavescript: A case-study in applying a distributed stream-processing language," MIT, Tech. Rep. MIT-CSAIL-TR-2008-005, January 2008.

[20] 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*, 2008, pp. 113–120.

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, May 2008.

[22] J. F. Reynolds, "A proof of the random-walk method for solving Laplace's equation in 2-D," *The Mathematical Gazette*, vol. 49, no. 370, pp. 416–420, Dec. 1965.

[23] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int'l J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[24] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial Monte Carlo simulation on architecturally diverse systems," in *Proc. of Workshop on High Performance Computational Finance*, Nov. 2008.

[25] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

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

[27] N. R. Tallen, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," in *Proc. of ACM/IEEE Supercomputing Conf.*, Nov. 2009.

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

[29] E. J. Tyson, J. Buckley, M. A. Franklin, and R. 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*, vol. 585, no. 2, pp. 474–479, Oct. 2008.

[30] K. D. Underwood, K. S. Hemmert, and C. D. Ulmer, "From silicon to science: The long road to production reconfigurable supercomputing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 4, pp. 26:1–26:15, Sep. 2009.