TimeTrial: A Low-Impact Performance Profiler for Streaming Data Applications

Joseph M. Lancaster
E. F. Berkley Shands
Jeremy D. Buhler
Roger D. Chamberlain

Joseph M. Lancaster, E. F. Berkley Shands, Jeremy D. Buhler, and Roger D. Chamberlain, "TimeTrial: A Low-Impact Performance Profiler for Streaming Data Applications," in *Proc. of IEEE International Conference on Application-specific Systems, Architectures and Processors* (ASAP), Sept. 2011, pp. 69-76.

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

TimeTrial: A Low-Impact Performance Profiler for Streaming Data Applications

Joseph M. Lancaster, E. F. Berkley Shands, Jeremy D. Buhler, and Roger D. Chamberlain
Dept. of Computer Science and Engineering, Washington University in St. Louis

Abstract—**Finding performance bottlenecks in application-specific systems is becoming increasingly labor-intensive as the capabilities of these systems improve. The complex platforms required to meet today's high application performance demands put pressure on developers to sustain current design cycles. Application developers need better tools to diagnose performance issues that arise when utilizing real-world application-specific platforms, from embedded applications to high-performance computational science applications.**

In this paper, we present TimeTrial, a runtime performance monitoring system that profiles streaming data applications deployed on architecturally diverse computers. TimeTrial is designed to operate with minimal impact on the executing application, exploiting user directives to aggressively perform lossy compression on performance meta-data. We present the design of the TimeTrial performance monitor and demonstrate its use in discovering performance bottlenecks in a production computational biology application.

Keywords-**Stream Computing, Runtime Monitor**

I. Introduction

Application-specific systems have been successful in accelerating applications in a wide variety of computing domains. Recently described examples include computational biology [1], [2], computational chemistry [3], and high-performance signal processing [4]. As the name suggests, each application-specific 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 application-specific integrated circuits (ASICs). FPGAs are a popular component in many application-specific systems due to the high degree of parallelism available and flexibility to reconfigure the architecture for each application.

However, effectively utilizing such 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 application-specific system relative to that of a traditional multi-core platform. As application-specific system designs become ever larger and more complex, the challenges become ever more acute.

The success of future designs for large, complex application-specific systems is contingent on improvements in developer tools. Underwood et al. [5] list twelve essential elements for reconfigurable logic (i.e., FPGAs) to achieve true acceptance in the high-performance computing community, and tools of various forms comprise two of their twelve elements: an improved programming environment (compilation from high-level languages that express application-level parallelism), and improved infrastructure (debuggers, correctness checkers, and performance analysis tools). Performance analysis is the subject of this paper.

Obtaining peak performance from an application-specific 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 left without even the most basic performance information about these components of their designs. Traditional software tools such as `gprof` [6] and TAU [7] 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 $\rightarrow$ 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.

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 in a low-impact manner 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. In contrast, simulation permits observation of execution on only small datasets, increasing the likelihood that events of importance to performance will be missed or under-sampled.

TimeTrial is designed to profile streaming applications. Stream processing is characterized by large volumes of data that are "streamed" from one computational kernel to the next over explicit communication paths. The paradigm is used in a number of real-world applications from signal processing to large-scale information processing systems. Languages that directly support streaming applications include Brook [8], StreamIt [9], Streams-C [10], Streamware [11], and X/Auto-Pipe [12]. Stream processing has many advantages for application-specific systems, such as supporting high-throughput communication, explicit wide and deep parallelism, and disjoint memory spaces.

Figure 1 shows an example of a signal-processing streaming application with three major stages. In stage 1, a sensor (e.g., a gamma ray telescope [13]) 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. [4] describe the implementation of this application using FPGAs to execute the computationally expensive stage 2 and multi-core processors for stages 1 and 3.

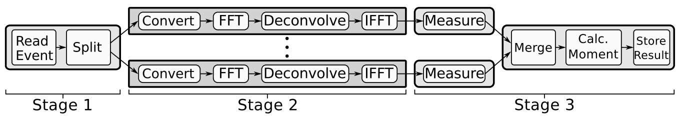


Fig. 1. An example streaming application from computational astrophysics [4]. Computation is performed within each kernel, communication is one-way along edges.

A profile of a streaming data application is somewhat 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, 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 at the input to 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?

In what follows, we describe the design of TimeTrial and demonstrate its ability to measure the performance of streaming applications deployed across traditional processors and FPGA accelerators. We show how TimeTrial can be used to answer questions like those above. To provide empirical evidence of TimeTrial's utility, we describe its use to provide application-wide performance measurements of a streaming, FPGA-accelerated implementation of the National Center for Biotechnology Information's (NCBI) BLASTN [14]. For BLASTN, we discover the bottleneck stage is a function of the input data set. The over-utilization of an SRAM-based lookup table is identified, pointing to specific redesign options that can improve overall performance across input data sets.

II. Related Work

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 node [15]. Stack-pointer tracing and instrumentation for path profiling are popular approaches. Message logging is also common in the MPI world [16]. 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` [6] 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 called frequently, slowing down communication. `gprof` treats the portions of the application deployed on the accelerator as a black box because 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 [7] 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 at the Univ. of Florida [17]. 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 links between computational kernels rather than monitoring interior signals within a kernel. Second, TimeTrial monitors both the FPGA and processor portions of the deployed application to locate bottlenecks regardless of the processing resource on which they occur.

A performance profiler has also been developed for the TMD machine developed at the Univ. of Toronto [18]. This profiler focuses on logging MPI communication calls and logging 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 data aggregation of performance metrics to reduce storage requirements.

We have previously published a description of TimeTrial's FPGA subsystem [19]. Here, we extend the description of the system to include the software agent, discuss our approach to handling timestamps across distinct compute resources, and provide a substantial example use case of the system.

III. The TimeTrial Runtime Performance Monitor

For applications executing on traditional processor cores, performance is typically understood via binary instrumentation that generates statistics at runtime. The lack of architectural diversity on such platforms simplifies the infrastructure required to generate statistics, and shared storage can suffice for collection of data. Hot spots in the code are found by program-counter sampling or by procedure logging. While every instruction and every event could be sampled to provide the greatest detail, the overhead of collecting, storing, and processing such information would be overwhelming. Instead, profilers typically seek a reasonable balance between fidelity and overhead. The overhead of tracking an application's performance is then some acceptable use of processor and storage resources to develop and store the performance data.

TimeTrial was written to support both parallel streaming semantics and different types of accelerator architectures. Currently, multi-core processors and FPGAs are supported. TimeTrial enables the the developer to indicate which portions of the application should be monitored, thereby reducing the burden on the runtime infrastructure. This allows the measurement system to target exactly the application's portions of interest (e.g., potential hot spots), regardless of computational resource.

A streaming application is naturally decomposed into computational kernels, called *blocks*. Data is communicated to and from blocks via edges to form a data-flow network, the *application topology*. Subsets of blocks then get deployed onto either individual processor cores or one of the accelerator devices available in the system. The assignment of blocks to computational resources is given by the application's *mapping*. Streaming applications use FIFOs between blocks to decouple the latency of a block from its throughput. In our streaming style, a FIFO is included by default on each edge between ports on connected blocks. An example of a simple streaming application topology is the tandem pipeline of Figure 2. Five blocks are included, with blocks A, B, and E mapped to processor cores and blocks C and D mapped to an FPGA.

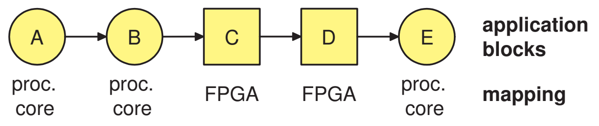


Fig. 2. Example application topology (A → B → C → D → E) and its mapping to processor cores and FPGA.

TimeTrial measures the performance of a streaming application by implementing directives from the developer 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.

While many streaming languages (e.g., Brook [8], StreamIt [9]) 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 [12]). By focusing on the application topology rather than block internals, TimeTrial is able to instrument the communication edges when 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. The constraint that monitoring be at the ports of 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 a (potentially newly created) output port.

One drawback when working with co-processor accelerators is that the memory capacity available to hold performance meta-data can be very limited. Additionally, the communication link(s) between the accelerator(s) and processor cores might already be saturated by the application or might be under-provisioned. Simply storing raw 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 online, summarizing the performance over a user-specified period. Currently supported data reduction operations include min, max, mean, standard deviation, and histogram.

Aggregating performance measurements into 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. To enable the user to control this tradeoff, 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.

Even very small time or data periods will yield some benefit over storing raw traces. Consider an execution where the developer simply wants to know the histogram of occupancy of a FIFO. A trace-based approach would record a stream of

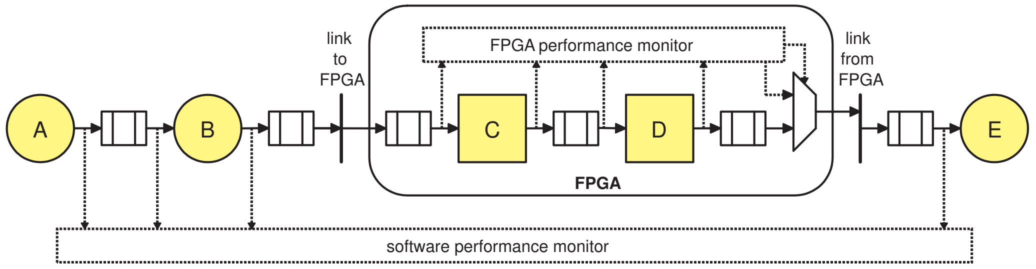


Fig. 3. Example application (A → B → C → D → E) deployed on an architecturally diverse computer comprised of processor cores and an FPGA. The dotted lines and boxes illustrate the runtime instrumentation of the application via TimeTrial.

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.

Figure 3 shows TimeTrial instrumenting the application of Figure 2. For each compute resource, a TimeTrial monitoring agent is added to the application to collect and communicate performance meta-data to the TimeTrial software agent process. In the figure, these are labeled as the FPGA performance monitor and the software performance monitor. *Taps* are inserted at the FIFOs in each communication link.

The computational resources used to implement the TimeTrial monitoring agents are dedicated resources (i.e., resources not shared with the application). On the FPGA, this is accomplished by allocating area on the FPGA to the monitoring agent (FPGAs are not well-suited to sharing on-chip resources in any event). On the multi-core processor, the monitoring agent is a distinct process that can be scheduled to any un- or under-utilized core. In Figure 3, 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.

A. FPGA Subsystem

As the FPGA agent within TimeTrial has been previously described [19], we will limit the discussion here to a brief overview. Aggregation operations are implemented using tap monitors, constructed using dedicated resources on the FPGA. Figure 4 shows the block diagram of the FPGA performance monitor instantiated with four tap monitors.

The tap router module is responsible for choosing and routing the subset of application taps necessary for the operation of each tap monitor. A tap monitor implements one specific performance-monitoring task, ranging from a simple counter to a more complex combination of operations as described

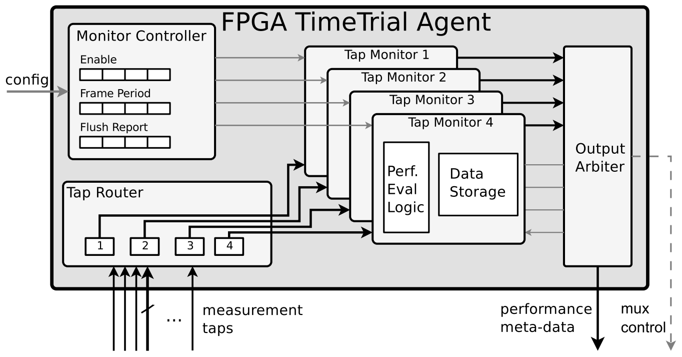


Fig. 4. Detailed view of the FPGA performance monitor (TimeTrial agent) shown with four tap monitors [19]. Data paths are shown as dark lines, control paths are gray lines.

previously. After evaluation of the performance metric, the result is stored in the meta-data storage block.

Because each tap monitor operates 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. The report is then inserted into the data stream via the multiplexer in Figure 3.

B. Software subsystem

The TimeTrial software performance monitor consists of two parts: a set of software taps and a data collection and processing agent. Figure 5 shows how these parts are connected.

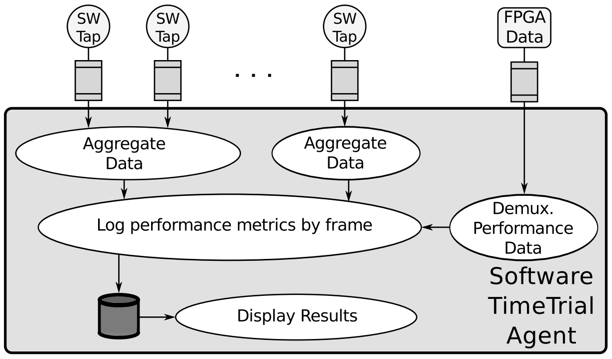


Fig. 5. Detailed view of the software TimeTrial agent.

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. A software tap is 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.

The software performance monitor agent is responsible for reading from one or more tap FIFOs and aggregating data from the taps into performance metrics. The developer specifies the metric and which tap(s) to use, and the agent will collapse event traces into a performance summary. For example, the agent can periodically transform enqueue and dequeue events from a FIFO to a histogram of queue occupancy over a given period. The software agent is executed on an unused or underutilized processor core (according to the scheduler's preference) in order to minimize interference to the application's execution.

Aggregated metrics from each FPGA agent are communicated in a similar manner via memory-based FIFOs, once the performance meta-data is demuxed from the application data. The software agent therefore has access to all the performance statistics for a run. These performance statistics are saved to disk prior to post-processing and presentation to the user.

C. Cross-platform issues: Timezones and Virtual Time

Given that the various computational resources on an architecturally diverse system do 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.

With knowledge of the relative rates of the two available time stamps, t_P and t_F, one can relate these time stamps to virtual time t_V via a set of linear transformations, i.e.,

$$t_V = s_P \cdot t_P + b_P \tag{1}$$

and

$$t_V = s_F \cdot t_F + b_F. \tag{2}$$

Here, 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. This method is similar to that used in NTP [20].

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.

IV. Profiling BLASTN with TimeTrial

In this section, we demonstrate our measurement infrastructure by characterizing the performance of a streaming, FPGA-based implementation of the NCBI BLASTN tool for DNA similarity search. We show that we can extract performance data from this application with low overhead and can discover bottlenecks that were previously unknown.

A. The BLASTN Application

BLAST, the Basic Local Alignment Search Tool [14], 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 [21], 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. In stage 1, 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, 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, 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 all three stages are reported to the user.

B. Profiling Accelerated BLASTN

The accelerated BLASTN [22] algorithm uses a diverse architecture consisting of both an FPGA and multi-core processors. The FPGA is used for the first two stages of the BLASTN computation, which dominate the running time, while the last stage is executed in software. Each FPGA-based stage is internally pipelined and parallelized to a fine grain. In particular, 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* [23], a "lossy" lookup table that can be implemented efficiently internally to the FPGA. Stage 1A filters out most words in the database that do not occur in the query, passing only a few percent of the database's words through to the SRAM lookup of stage 1B.

The original implementation of the accelerated BLASTN was designed to carefully balance the loads on the various stages so that no one stage was too great a bottleneck. Our 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 simulator, we opt to use TimeTrial to directly measure the performance of the new implementation.

TimeTrial instrumentation was added to the top-level blocks for BLASTN. We then executed BLASTN to compare two sets of sequences and collected runtime statistics at native application speeds. The first run, Exp. 1, compares recombinant viral DNA (sequenced in-house, 293×10^6 bases) against the 19th assembly of the human genome[1] (3×10^9 bases). Exp. 2 compares the non-mammalian RefSeqs[2] (302×10^6 bases) to the NCBI mammalian RefSeqs[3] (788×10^6 bases). The accelerated BLASTN splits one of each pair of sequences into many small ($\sim 65 \times 10^3$ bases) queries, then performs multiple passes over the other sequence (the database) to complete the entire comparison. The hardware platform consists of two quad-core AMD Opteron processors running at 2.4 GHz, with 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.

TimeTrial's profile results for the accelerated BLASTN are shown in Figure 6. Software processes/threads are shown as circles, while FPGA modules are shown as squares. A queue is placed between each two stages in the pipeline to smooth out burstiness. 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, which when active indicate 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 aggregated through the median and quartiles on the plot. Similarly, each pass over the database results in a queue occupancy histogram for each instrumented queue. These individual results are 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 boxplots.

Studying Figure 6, we find that in both cases, the software portion of BLASTN is non-limiting. We conclude this from the full queue at the ingress of the FPGA and the empty queue at its egress. However, there is some burstiness in stage 3 processing for Exp. 1: about 20% of the results from the FPGA over-run the available processors and start to fill the queue. Exp. 2 never has more than one element in this queue.

The hardware shows a different story between the two experiments. Exp. 1 has a much higher and less variable throughput compared to Exp. 2. In both experiments, when the throughput is less than the maximum possible line rate ($\sim$875 MB/sec), the slowdown is caused by stage 1B asserting back-pressure. Exp. 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 Exp. 1 stresses stage 1B in about half of the passes over the database, but that the buffering upstream tends to ameliorate this. In Exp. 2, stage 1B is more severely stressed, filling the upstream buffers and causing lower ingest rates.

In these examples, TimeTrial paints a clear picture of the application profile and allows efficient bottleneck discovery. Figure 6 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 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 new board. As a result, hash table collisions will be greatly reduced, and stage 1B will utilize the SRAM more efficiently.

C. Overhead and Impact

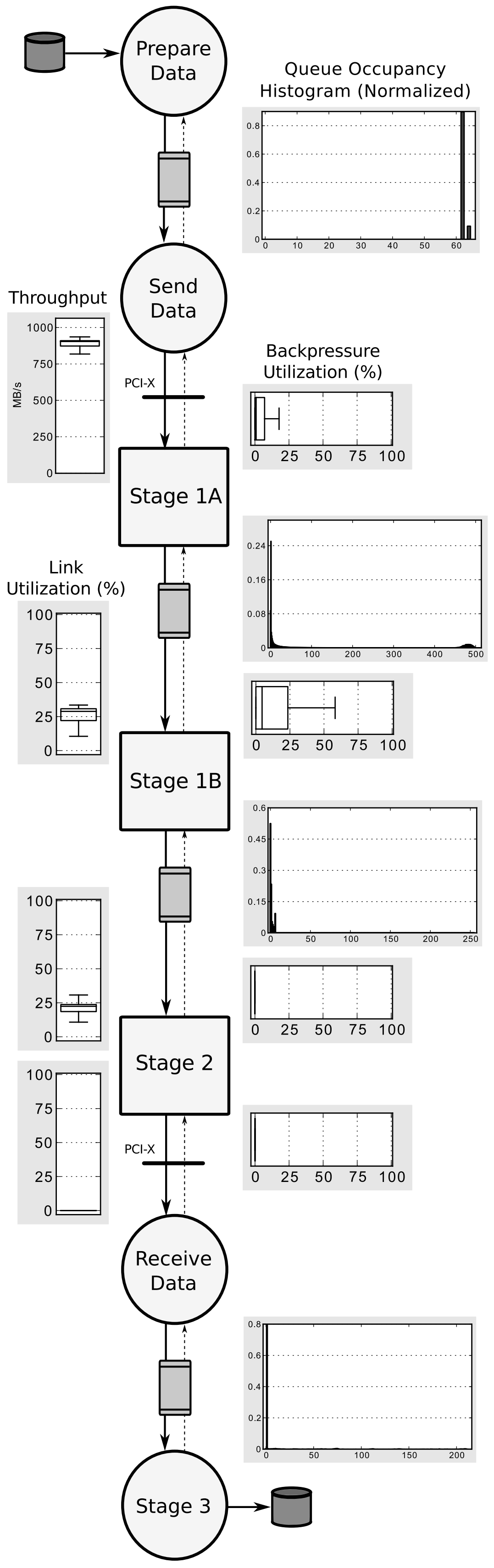
Using TimeTrial to profile an application incurs some unavoidable overhead. We define overhead as the amount of extra resources and communication that is utilized by the TimeTrial system. Impact is the observable difference in runtime due to instrumentation. In what follows, we quantify the the overhead and impact on a run similar to Exp. 1.

The software instrumentation utilizes 100 MB (of 16 GB) of main memory allocated to the FIFOs and aggregation functions in the agent. The agent utilizes 27% of one core.

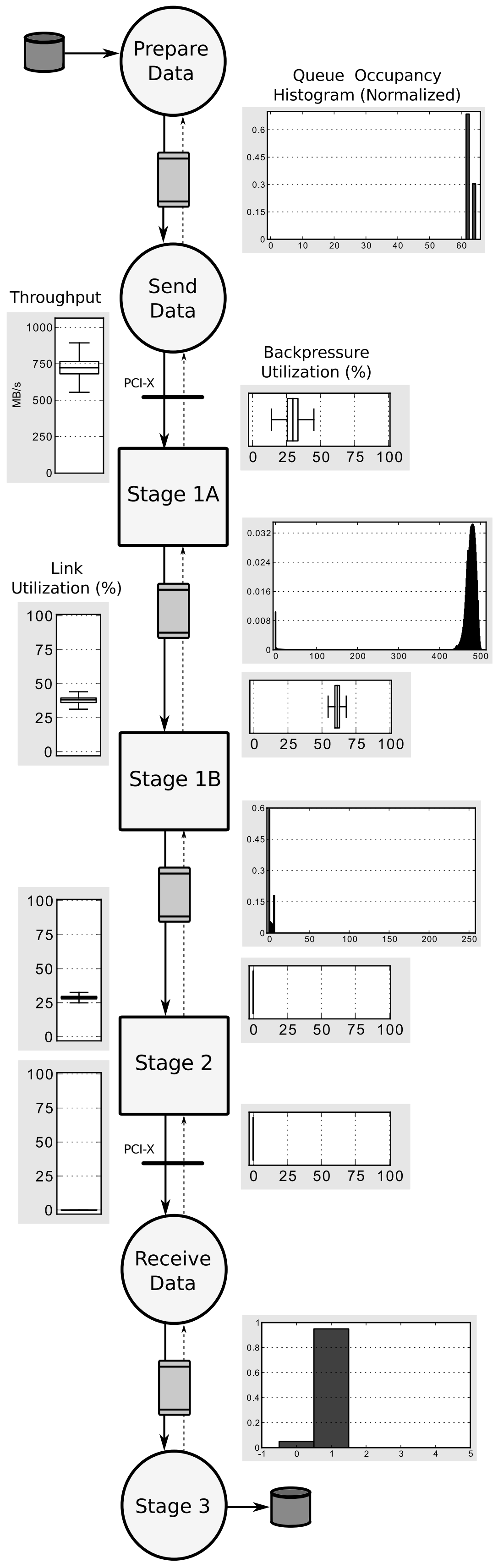
[1]http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/

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

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



(a) Experiment 1 profile



(b) Experiment 2 profile

Fig. 6. TimeTrial measurement results from running BLASTN on two large genetic datasets.

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 communication overhead of 2.86 MB.

The resource overhead of the FPGA agent is shown in Table I. The utilizations for all resource types are single-digit percentages for calculating thirty 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 as long as the application is clocked 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 I
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, and 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.

V. Conclusions and Future Work

This paper describes the runtime performance monitor TimeTrial, which provides novel insight into the performance of streaming applications deployed onto multi-core processors and FPGAs. Through aggressive data aggregation, TimeTrial is able to acquire performance data in a low-impact manner. Results are presented showing how TimeTrial helped to find bottlenecks in an accelerated implementation of BLASTN and directed future engineering effort towards a better performing implementation.

Currently, a TimeTrial compiler is being constructed to automatically instrument applications written in a streaming language. Also, we are exploring ways to integrate GPUs into the system. Understanding the performance of applications deployed on architecturally diverse systems is a difficult task, and TimeTrial makes the job considerably easier.

Acknowledgements

This research was supported by the National Science Foundation through grants CNS-0751212, CNS-0905368, and CNS-0931693.

References

[1] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, “Single pass streaming BLAST on FPGAs,” *Parallel Computing*, vol. 33, pp. 741–756, 2007.

[2] S. Datta, P. Beeraka, and R. Sass, “RCBLASTn: Implementation and evaluation of the BLASTn scan function,” in *Proc. of 16th Int’l Symp. on Field-Programmable Custom Computing Machines*, Apr. 2009.

[3] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, “Comparing hardware accelerators in scientific applications: A case study,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 58–68, Jan. 2011.

[4] 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.

[5] 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, Sep. 2009.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, “Stream-oriented FPGA computing in the Streams-C high level language,” in *Proc. of IEEE Int’l Symp. on FPGAs for Custom Computing Machines*, 2000, pp. 49–58.

[11] 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.

[12] 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.

[13] T. Weekes *et al.*, “VERITAS: the very energetic radiation imaging telescope array system,” *Astroparticle Physics*, vol. 17, no. 2, pp. 221–243, May 2002.

[14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” *J. of Molecular Biology*, vol. 215, pp. 403–10, 1990.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] D. L. Mills, “A brief history of NTP time: Memoirs of an Internet timekeeper,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 9–21, April 2003.

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

[22] J. D. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. Chamberlain, “Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture,” in *Proc. of Reconfigurable Systems Summer Institute*, Jul. 2007.

[23] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, May 1970.