# Understanding the Performance of Streaming Applications Deployed on Hybrid Systems

**Joseph Lancaster**
**Ron Cytron**
**Roger D. Chamberlain**

Joseph Lancaster, Ron Cytron, and Roger D. Chamberlain, "Understanding the Performance of Streaming Applications Deployed on Hybrid Systems," in *Proc. of Next Generation Software Workshop*, April 2008 (associated with IPDPS).

Dept. of Computer Science and Engineering
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130-4899

# Understanding the Performance of Streaming Applications Deployed on Hybrid Systems

Joseph Lancaster, Ron Cytron, and Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
lancaster@wustl.edu, cytron@wustl.edu, roger@wustl.edu

## Abstract

*Significant performance gains have been reported by exploiting the specialized characteristics of hybrid computing architectures for a number of streaming applications. While it is straightforward to physically construct these hybrid systems, application development is often quite difficult. We have built an application development environment, Auto-Pipe, that targets streaming applications deployed on hybrid architectures. Here, we describe some of the current and future characteristics of the Auto-Pipe environment that facilitate an understanding of the performance of an application that is deployed on a hybrid system.*

## 1. Introduction

Recent years have seen the emergence of a number of differing computing technologies. Single-processor systems have yielded the mainstream to multi-core versions of general-purpose processors (GPPs), chip multiprocessors (CMPs) with many cores are now available [12], heterogeneous CMPs are used both in gaming systems and for computational science [9], graphics processing units (GPUs) are used for general-purpose computation [1], and field-programmable gate arrays (FPGAs) have been used to accelerate many applications. We refer to systems constructed using a variety of the above technologies as hybrid systems.

Each of the above technologies has its own architecture, memory subsystem, language for authoring applications, performance capabilities and limitations, algorithmic strengths and weaknesses, and communities of proponents and detractors. Our research focus is on enabling application development across a hybrid system, allowing the application developer to assign portions of the application to appropriate subsystems that are well-suited to the computations assigned.

An important class of applications that can effectively exploit the capabilities of hybrid systems is streaming applications. Here, data to be processed are streamed into the system, a variety of pipelined computations are performed on the data, and the results are then streamed out of the system. Examples of streaming applications include sensor-based signal processing, media (audio and video) processing, and many data-intensive scientific codes [3].

The Auto-Pipe development environment [2, 4] supports the design and deployment of streaming applications on hybrid systems. Auto-Pipe already contains a federated simulation infrastructure [5] that provides timestamped data traces for each arc in the dataflow representation of a streaming application. Here, we describe the principles that are guiding our runtime performance monitoring system that is currently under development.

## 2. Streaming Applications and Hybrid Systems

Generally, streaming applications can be thought of as course-grained dataflow computations in which computation blocks, or kernels, are interconnected by arcs over which data is communicated. An example application topology is illustrated in Figure 1. Block A is a data source, the output data stream from block A is delivered as an input stream to block B, etc.

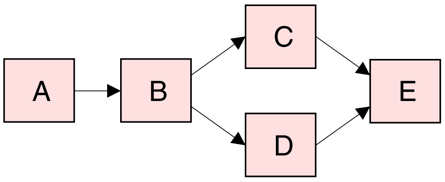

**Figure 1. Sample application dataflow graph.**

In most streaming languages (e.g., Brook [1], StreamIt [11]), the computation within the blocks and the interconnections of the blocks are all expressed in a

This work was supported by NSF grant CNS-0720667.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

common language. In the Auto-Pipe environment, block computations are expressed in a language suited to the computational resource(s) onto which the block may be allocated. For example, blocks that may be allocated to traditional processor cores are coded in C/C++ while blocks that may be allocated to FPGAs are coded in VHDL or Verilog. The interconnections between blocks are expressed in a coordination language, X [4], which is independent of the block expression language(s).

Streaming applications are well-suited to execution on hybrid architectures, in part because hybrid systems frequently have distributed memory infrastructures, and the explicit expression of required data movement inherent in streaming languages enables the memory to be effectively utilized. Today, it is fairly straightforward to physically construct a hybrid system. Figure 2 shows an example of a hybrid system constructed using dual-core AMD Opterons, an off-the-shelf graphics card connected to HyperTransport (HT) via a PCIe bus, and an FPGA card connected via a PCI-X bus. Note that while the DRAM memory on the processors is generally accessible by all of the GPP cores, the memories associated with the GPU and the FPGA are segregated (both from the main memory and each other).

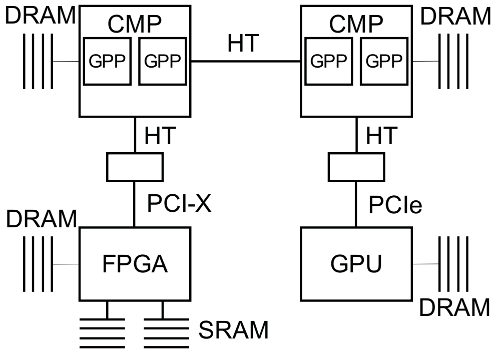

**Figure 2. Example hybrid system.**

By associating device numbers with each of the computational components of the hybrid system, one can explicitly map blocks in the streaming application to devices in the hybrid computer. Figure 3 illustrates a circumstance where block A has been allocated to Device 1, blocks B, C, and D have been allocated to Device 2, and block E has been allocated to Device 3. If, for example, Device 1 is one of the processor cores and Device 2 is the FPGA, the Auto-Pipe environment automatically provides the infrastructure for data delivery from the output port of block A to the input port of block B, moving the data across the PCI-X bus.

Prior to actual deployment on the hybrid system, the X-Sim federated simulation tool, which is part of the Auto-Pipe environment, provides a simulation model of the streaming application as mapped to the various devices in the hybrid system. The figure also shows the timestamp trace files collected by X-Sim. At an outbound arc departing a device, the times that data elements are available are recorded in the file labeled $T_{out}$. Associated with inbound arcs are timestamp files that record the time data elements are available, $T_{avl}$, and the time data elements are consumed, $T_{in}$.

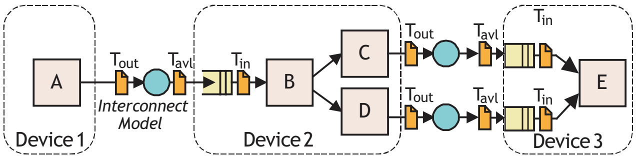

**Figure 3. Data traces collected with X-Sim[5].**

X-Sim can be used to verify both functional correctness and performance characteristics of the application by examining the trace files created as part of the simulation execution. Essentially, the simulator provides complete observability into the communication arcs of the streaming application. (Note that while the example illustration only shows traces being created for inter-device communication, there are mechanisms available for tracing interior arcs as well.)

Once the application developer is happy with what can be learned from simulation, the task moves to actual deployment, executing the streaming application directly on the hybrid system. In the Auto-Pipe system, the developer specifies which application blocks are to be mapped to which devices in the hybrid system, and Auto-Pipe provides the appropriate communications infrastructure to effect the necessary data movement. In the example above, data from a processor core to the FPGA was moved across the PCI-X bus. Data delivery between two processor cores uses the native shared memory, data to/from the graphics engine uses PCIe, etc.

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

## 3. Understanding Runtime Performance

For non-streaming applications and relatively monolithic platforms, performance is typically understood via instrumentation that generates statistics at runtime. The relative lack of 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, a reasonable balance between fidelity and overhead is typically struck. 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.

However, developing a reasonable global view of a distributed application's performance is not as straightforward. Distributed applications do not typically share storage, nor is there a universal program counter that accurately reflects the state of the application. In this paper, we propose methods for understanding the performance of streaming applications on diverse (*ergo* distributed) platforms.

Ideally, we would like to have timestamps for every datum as it flows through the system, much like the current simulation system offers, but operating at full system speed. Additionally, this information should come at no overhead cost so the measurements are believable. Achieving this is impossible on most real systems as the bandwidth and storage constraints are the limiting factors. Given that we are bound by this restriction, what is a reasonable thing to do?

One approach is to enable significant customizability into the specific performance data to be collected, allocating resources targeted directly at the performance question of interest. An example of a system of this type is described by Hough et al. [8], which instruments a soft-core processor deployed on an FPGA. Alternatively, one can deploy numerous performance counters across a chip (e.g., as on the Cell [6]). The above techniques, however, are generally focused on an individual computing resource. Our interest is in the complete hybrid system, composed of a collection of computing resources.

In massively parallel compute clusters, the dominate programming paradigm is via message passing (typically using MPI). In this environment, there are tools that collect and present each message [7]. These tools, however, can significantly impact the temporal properties of the executing program, thereby altering the target they wish to measure.

Here we offer a method of performance monitoring in a hybrid streaming system that allows for excellent data characterization and flexibility in trading-off temporal resolution for higher precision performance metrics. In addition, the system is focused on offering quantitative measures of the uncertainty present in both temporal and performance metrics so the user can draw statistically-sound conclusions which can then guide future implementation choices. Next, we describe the salient features that make this monitoring approach appropriate for this problem domain.

One popular (and sensible) approach to monitoring the performance of a program is to aggregate performance information over the entire run of a dataset. This can typically be done incrementally, minimizing the impact of the performance monitoring task on the dynamic behavior of the application itself. The aggregate information is output at the end of the run and shown to the developer. Aggregate results can truly be beneficial to the user: they can be used to check the validity of analytic models or simply alert the developer to unexpected performance issues. Unfortunately, this approach misses a key piece of information: when or where (causally) do performance anomalies occur during the run. This information can inform developers of real-time systems of performance-related issues which are considered bugs.

On the other hand, one can imagine a system where every datum is accompanied by meta-data that describes its performance at every step of the algorithm, as is the case in simulation systems such as X-Sim. Attempting to monitor programs at this granularity is an infeasible task but for the most trivial of programs and performance metrics because of computational and bandwidth restrictions in real systems.

The above two approaches can be viewed as being at opposite ends of a spectrum of information resolution. Aggregate statistics have very limited information concerning the temporal properties of the application, while per-datum meta-data has complete information. Fortunately, as system designers, we are not limited to either end of the spectrum and in fact we believe the system should allow flexibility as to where in that spectrum the user wishes to operate. We assert that one can craft a healthy balance between the quality of temporal information and precision of the performance metrics.

In order to manage temporal performance information we borrow a concept of framed computation from the media compression literature. Instead of measuring over the entire execution of a program, the measurement is divided into measurement "frames." A frame is simply the length of an individual measurement in time or the size of a measurement defined in datum count. Statistics are initialized each time a new frame is encountered and the results are reported at the end of a frame. In this way, the user is given an ordered *set* of performance metrics from which he/she can reason about the behavior of the application. The cardinality of the set for a given run length is determined solely by the chosen frame size. The set of frames, or a subset of frames, can also be aggregated in many cases to provide performance information about the entire run if the user so desires.

In an ideal environment (without resource constraints), the user could specify a frame size of one datum, since this provides the most information about the execution of the application. Realistically, the frame size should be carefully chosen to provide the sampling frequency that barely exceeds the Nyquist frequency of the variability of the system in order to minimize interference with the execution of the application itself. Note that in many cases, even this basic condition may not be feasible.

Collecting performance information in frames essentially aggregates only the data within the frame instead of the data over the entire run. Hence, the developer will receive temporal information about the behavior of the system. Some information is still lost, however, for frame sizes greater than one datum. To help the user understand the significance of reported performance results, we propose quantitative measures of the uncertainty of aggregated statistics so that the trade-off of larger frames can be more easily evaluated.

For instance, assume that within a frame one collects the average time a datum spends in a particular stage of an algorithm. A frame size of one would tell exactly how long each datum resides in that stage. Collecting only one frame over the entire run will give the overall average execution time of that stage. In a realistic system, the operating point will likely somewhere on the continuum between the two. If the performance behavior of the application is highly dependent on data-driven effects, the frame size must be small enough to capture the essence of this property. Hence it is extremely important that the quality of the statistics be carefully evaluated by the developer.

In the above example, a simple statistic, the mean execution time, was used as the performance metric. We believe that flexibility in the metrics being measured is paramount to the utility of the system. Unfortunately, not all performance metrics require the same amount of data to report. A simple mean execution metric may only take 4 bytes to report per frame; a histogram of blocking probabilities may require on the order of kilobytes or more per frame. As a result, the choice of an appropriate frame size depends not only on the variability of the application executing on the system but also on the choice and resolution of performance metric(s) collected. Again, we would prefer to not impact the program's behavior by using only the extra available system bandwidth for performance monitoring. This raises the question of how does one choose between the quality of the temporal information and the quality of the measured performance metric(s) with the goal of providing the "best" performance information to the developer.

To develop a quantitative analysis of the performance monitoring system, let's explore the implications of these trade-offs using an example. Consider a streaming block $K$ that has one input stream and one output stream. $K$ has the property that for each input there is some probability $0 \leq P_K \leq 1$ that it will generate an output. Block $K$ is representative of many real-world applications, such as filters, that do not have fixed input-output relations. This example closely mimics the behavior of a block we developed for accelerating BLAST in our previous work [10].

A reasonable performance metric to observe is $K$'s *filter strength*, $S_K$, defined as the number of input datums consumed for each output datum generated. $S_K$ is a desirable metric to know since it establishes the input to output data volume ratio for $K$. Also, it may be represented by an average, a histogram, or some empirically-informed model. One might want to know $S_K$ to determine the bandwidth requirements for both the inbound and outbound links connecting the computing resource that is executing $K$ to its upstream and downstream neighbors.

Let $T_f$ be the period (in seconds) of a performance frame. Let us also assume $S_K$ is characterized by collecting a histogram $H_{K,i}$ of observed values during each frame $i$. Each histogram $H_{K,i}$ has two parameters, $N_i$, the number of bins in $H_{K,i}$, and $W_i$, the width of each bin.

Now let us examine the bounds on the error inherent in $H_{K,i}$. Let $R$ be the range of the data in the measurements. Then the maximum quantization error of $H_{K,i}$ is defined as $E_{q,i} = R_i/W_i$. Similarly, the maximum error due to aggregating temporal performance data (assuming a fixed frame size) is simply the period of the frame, $E_{f,i} = T_f$.

We propose that, at a minimum, these error bounds accompany each point on the performance graphs. Let's assume that the data from an execution of our example application is presented as frame times on the x-axis and filter strength on the y-axis as shown in Figure 4. On the x-axis, $E_{f,i}$ are shown as horizontal error bars. These bars show the maximum amount of uncertainty in time that the user will expect from the chosen frame size. On the y-axis, $E_{q,i}$ are shown as vertical error bars. The vertical bars show the upper bound on the errors due to the aggregation of individual statistics within a frame.

The two uncertainty measures can be conceptualized as a rectangle bounding the error both in time and value. Resource constraints of the system such as buffering capacity or outbound bandwidth create a minimum area of the rectangle. Under this constraint, an automatic choice of $T_f$ that minimizes the perimeter of the rectangle (the sum of the two uncertainties) is appropriate. However, in some situations the user may benefit by decreasing the uncertainty of one axis and trading-off accuracy on the other axis. In this case, the area of this rectangle is still at least as large, the shape is simply altered. Providing direct user control is clearly required.

## 4. Conclusions

In this paper, we have presented the current and future capabilities of the Auto-Pipe application development environment focused on performance understanding of stream-

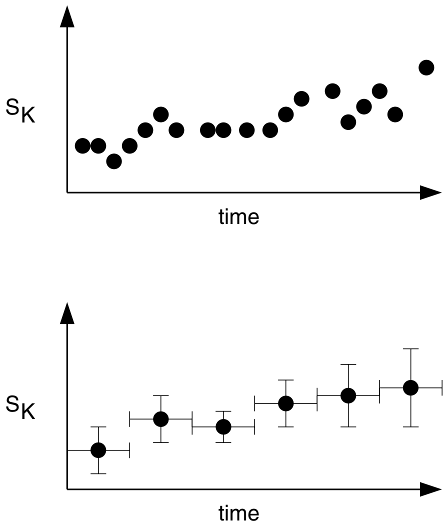

**Figure 4. Example execution trace and resulting performance graph after framing.**

ing applications executing on hybrid systems. In the X-Sim simulator, observability is unconstrained by available resources and complete timestamped data traces are collected for each physical data communication path. When one deploys the application on the actual hybrid system, resource constraints such as available buffer memory and/or outbound communication bandwidth limit the total volume of performance data that can be reported. In this case, the system enables flexible configuration of the uncertainty inherent in the reported performance metrics. By reasoning rigorously about the uncertainty present in both the temporal domain and measured metrics, the trade-offs inherent in configuring the performance monitoring system can be better understood.

## References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[2] R. D. Chamberlain, E. J. Tyson, S. Gayen, M. A. Franklin, J. Buhler, P. Crowley, and J. Buckley. Application development on hybrid systems. In *Proc. of Supercomputing (SC07)*, Nov. 2007.

[3] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proc. of 15th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 33–42, Sept. 2006.

[4] 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.

[5] 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*, June 2007.

[6] M. Genden, R. Raghavan, M. Riley, J. Spannaus, and T. Chen. Real-time performance monitoring and debug features of the first generation Cell processor. In *Proc. of 1st Workshop on Tools and Compilers for Hardware Acceleration*, Sept. 2006.

[7] M. T. Heath, A. D. Malony, and D. T. Rover. The visual display of parallel performance data. *IEEE Computer*, 28(11):21–28, Nov. 1995.

[8] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood, and R. Cytron. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, Feb. 2006.

[9] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. of Research and Development*, 49(4/5):589–604, 2005.

[10] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 49:101–121, 2007.

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

[12] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept./Oct. 2007.