Application Development on Hybrid Systems

Roger D. Chamberlain
roger@wustl.edu

Eric J. Tyson
etyson@wustl.edu

Saurabh Gayen
gayen@wustl.edu

Mark A. Franklin
jbf@wustl.edu

Jeremy Buhler
jbuhler@wustl.edu

Patrick Crowley
pcrowley@wustl.edu

Dept. of Computer Science and Engineering
Washington University
St. Louis, Missouri

James H. Buckley
Dept. of Physics
Washington University
St. Louis, Missouri
buckley@wuphys.wustl.edu

ABSTRACT

Hybrid systems consisting of a multitude of different computing device types are interesting targets for high-performance applications. Chip multiprocessors, FPGAs, DSPs, and GPUs can be readily put together into a hybrid system; however, it is not at all clear that one can effectively deploy applications on such a system. Coordinating multiple languages, especially very different languages like hardware and software languages, is awkward and error prone. Additionally, implementing communication mechanisms between different device types unnecessarily increases development time. This is compounded by the fact that the application developer, to be effective, needs performance data about the application early in the design cycle. We describe an application development environment specifically targeted at hybrid systems, supporting data-flow semantics between application kernels deployed on a variety of device types. A specific feature of the development environment is the availability of performance estimates (via simulation) prior to actual deployment on a physical system.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*heterogeneous (hybrid) systems*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Astronomy*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC07 November 10-16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

General Terms

Design, Experimentation, Measurement, Performance

Keywords

hybrid systems, performance modeling, hardware/software co-design, gamma ray astronomy

1. INTRODUCTION

In recent years, a number of basic computational resource types have matured to the point that they can materially benefit high-performance applications. These resources include multi-core general-purpose processors, reconfigurable hardware (e.g., FPGAs), graphics processors, digital signal processors, and other application-specific processors. In many cases, the performance gains associated with these specialized computational resources are quite significant, improving application performance by one to two orders of magnitude or more.

We refer to systems built out of these resources as *hybrid* systems, and while it is reasonable to construct a hardware prototype that includes a number of hybrid compute resources, application development for such a system is quite difficult. This is true for a number of reasons.

- In most cases, each compute resource has its own language, development environment and tools, run time environment, and debugging aids.
- The intellectual task of describing the computation is often quite different for each compute resource. For example, general-purpose processors and chip multiprocessors are typically programmed using task-level parallel threads, while reconfigurable hardware is programmed at the register-transfer level.
- Delivering data between these disparate environments is a significant task in its own right.

The result is that while significant performance gains are achievable using hybrid systems, it is only with enormous

programming effort. Our aim is to simplify the development and deployment of applications onto these systems.

The above difficulties are merely the beginning of the problem, however. Given that performance is often the primary motivating factor when using hybrid computational platforms, the lack of consideration given to evaluating performance in the vast majority of application development environments is regrettable. Performance evaluation can involve empirical measurements of constructed systems, or performance estimates for candidate systems where direct measurements are not possible. We take the position that performance evaluation is a primary design consideration, second only to correctness, and that it should be richly supported in the development environment.

We are designing a development environment for applications executing on hybrid computing platforms. Our solution involves the use of a coordination language to specify dataflow style interconnections between compute blocks, and native languages and tool sets for the development of the compute blocks themselves. The environment supports the evaluation of application performance early in the design cycle, (semi-)automated mapping of compute blocks to computational resources, and direct support for block to block communication both within and between computational resources.

Given the decomposition of an application into a set of interconnected compute blocks (e.g., application pipeline stages), and the existence of implementations (potentially on more than one type of compute platform) of each compute block, Figure 1 illustrates one of the design questions that the development environment intends to address. Across the top of the figure is an application that consists of three pipelined computational stages ($1 \rightarrow 3$). These stages, for example, might represent application modules expressed both in C and in VHDL. Across the bottom of the figure is a pair of computing resources (compute platforms 1 and 2). The figure illustrates application stage 1 being mapped to compute platform 1, application stage 3 being mapped to compute platform 2, and a question as to whether application stage 2 should be mapped to compute platform 1 or 2.

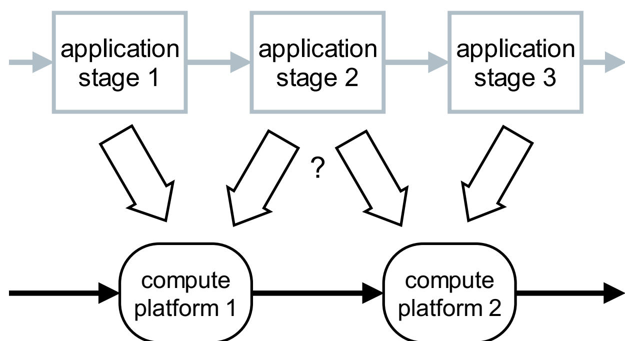

Figure 1: Mapping application to architecture.

When the time required to execute each application stage on each compute platform is well known, and when the communications costs for moving data between compute platforms is also well understood, it is a straightforward task to perform the above mapping optimally. When the two assumptions above do not hold, however, the choice of appropriate mapping needs to be guided by performance models that *can* incorporate compute time for application stages and communication costs between stages.

While the figure illustrates a particular design question, a full design problem has many such questions. For example, what technology should be used for compute platform 1 (e.g., processor or reconfigurable logic)? How does this choice impact the mapping question for application stage 2? We have built an application development environment that helps developers answer exactly such questions, while keeping them cognizant of the performance implications of their design decisions.

2. HYBRID SYSTEMS

For the purposes of this paper, we use the phrase *hybrid system* to mean a computing system that incorporates two or more distinct computational resource types. Candidate computational resources (or *platforms*) include the following:

- general-purpose processors such as x86 processors from AMD or Intel
- homogeneous, multi-core versions of general-purpose processors; while AMD and Intel are shipping dual-core x86 processors, Sun has eight-core SPARCs in production
- heterogeneous, multi-core processors, which provide processors of varying capability within a single chip; they have their roots in network processors (e.g., Intel IXP) and are expanding their scope of usage with the introduction of the IBM Cell processor
- reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs)
- Digital Signal Processors (DSPs) or other Application Specific Instruction Processors (ASIPs); processors for which the instruction set and/or architecture have been optimized for an individual application or class of applications
- Graphics Processing Units (GPUs); traditionally aimed at visual rendering, these processors are now being used for a wide variety of purposes.

There are a wide variety of architectural options available to the designer when constructing a hybrid system. Figure 2 illustrates an example configuration in which an FPGA is attached to one processor via the local PCI bus, a network processor with 8 engines is attached to another processor via its local PCI bus, and a dual-core processor is attached to the overall system via a local network (e.g., Ethernet or Infiniband).

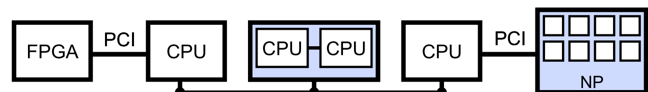

Figure 2: Example hybrid system architecture.

While it is fairly straightforward to build a hybrid system of the type described above, it is yet another matter to develop applications that can effectively exploit the capabilities of such a system. Distinct computational resource types typically have their own languages for describing applications. For example,

- general-purpose processors – sequential, procedural languages such as C/C++, Java, etc.;
- multi-core processors – thread-based parallelism for homogeneous cores, specialized constructs (often including native assembly language) for heterogeneous cores;
- FPGAs – hardware description languages such as Verilog, VHDL, and SystemC;
- DSPs and ASIPs – C/C++ and assembly language; and
- GPUs – stream programming languages such as Brook [3] or APIs such as OpenGL.

Associated with each of these languages is a distinct tool set that includes: 1) compilers (or synthesizers) that transform the source into executable form; 2) run time environments that provide services such as scheduling and resource management; and 3) debugging aids that enable the developer to observe various portions of the state of the computation. In addition, there is little support in any of the above languages or tool sets to enable data delivery between one type of computational resource and another. Enabling designers to develop high performance applications that run correctly, in spite of the above limitations, is the focus of our research.

Even with the above obstacles to efficient use of application developer time, performance gains have been demonstrated on real applications executing on hybrid systems, e.g., see [1, 2, 5, 6, 7, 9, 12, 13, 14, 16, 17, 20, 21, 24]. Example applications that can benefit from hybrid system architectures include signal processing, biosequence search, encryption, text search, etc. The performance gains achievable in hybrid systems come from a variety of opportunities.

- fine-grained parallelism – While traditional, single-core, general-purpose processors have fairly limited computational parallelism available, the other types of computational resources available in hybrid systems offer significant, on-chip parallelism.
- functional specialization – When specialized hardware is well matched to the application, performance is typically much greater than when executing on general-purpose hardware. While DSPs and GPUs provide this benefit for a class of applications, FPGAs have the added benefit that the hardware itself can be altered to match the application, further increasing the number of applications that can see performance gains.
- distinct compute resource types – With more than one computational resource type in a system, it need not be the case that the entire application be deployed on any one resource. This has the clear advantage that portions of the application can be deployed on the portions of the system to which they are best suited.

While the first two points above rely on on-chip technology, and are therefore primarily the domain of the chip manufacturers, the strength of the latter point depends heavily on the ability to efficiently move data between computational resources (i.e., the system must be tightly connected). This tight connection must be both at the architectural level and the application level. High-bandwidth, low-latency data delivery mechanisms need to be available and function effectively.

Most existing hybrid systems are research machines; however, there are a number of systems (and components) that are now available commercially. Homogeneous, multi-core, general-purpose processors are now ubiquitous. Example systems with FPGAs include the Cray XD1 (`www.cray.com`) and SGI RASC (`www.sgi.com`). Example FPGA add-on boards connect to the host via PCI (typically PCI-X), such as offerings from Annapolis Micro Systems (`www.annapmicro.com`) and Nallatech (`www.nallatech.com`), or via HyperTransport in a processor slot, such as offerings from DRC (`www.drccomputer.com`) and XtremeData (`www.xtremedatainc.com`). Mercury Computer (`www.mc.com`) makes combined general-purpose processor (both PowerPC and Cell) and DSP systems. Programmable GPUs are available from both NVIDIA (`www.nvidia.com`) and ATI Technologies (`www.ati.com`) and primarily connect via PCI (PCIe).

LabView [18] is a proprietary application development environment from National Instruments, targeted towards hybrid systems such as the ones described above. It is popular among scientists and engineers because it provides a graphical language “G” that can be used to design dataflow graphs. This graphical programming environment is a major selling point of LabView, but drawbacks include the system’s proprietary nature, as well as a lack of support for simulation, performance analysis, and performance optimization.

Ptolemy [19] is another hybrid application development system, with code generation capabilities for C, C++, VHDL (incomplete), and DSP assembly (for the TI-320 series). It is more focused towards embedded systems, and is now fairly dated, with support for the system only existing on the Solaris and HP-UX operating systems.

3. APPLICATION AUTHORING FOR HYBRID SYSTEMS

There are many possible approaches to the problem of expressing applications to be deployed across hybrid systems. While it is possible to express arbitrary hybrid applications using a single language, such an approach would likely be awkward, make inefficient use of the unique resources of each platform, and lack the robustness and user base of the language types that have succeeded in their respective fields (e.g., procedural languages on processors, structural HDLs on FPGAs, or stream languages on GPUs).

Our approach is to take advantage of these relatively efficient, robust, and well-entrenched languages by designing a coordination language called X which is capable of connecting task kernels—written in traditional languages—in a dataflow manner. Each kernel, called a *block*, can have any number of platform-specific implementations of varying specificity, such as an ANSI C implementation, or a C implementation using MMX instructions for Intel compatible processors, or a VHDL implementation for FPGAs. All implementations of a single block are required to provide the same interface and data flow semantics in order to ensure correctness regardless of the block-to-resource mapping. Each language supported has a specific API and syntax for specifying the particular dataflow interface employed by the block, such as input ports, output ports, and configuration values.

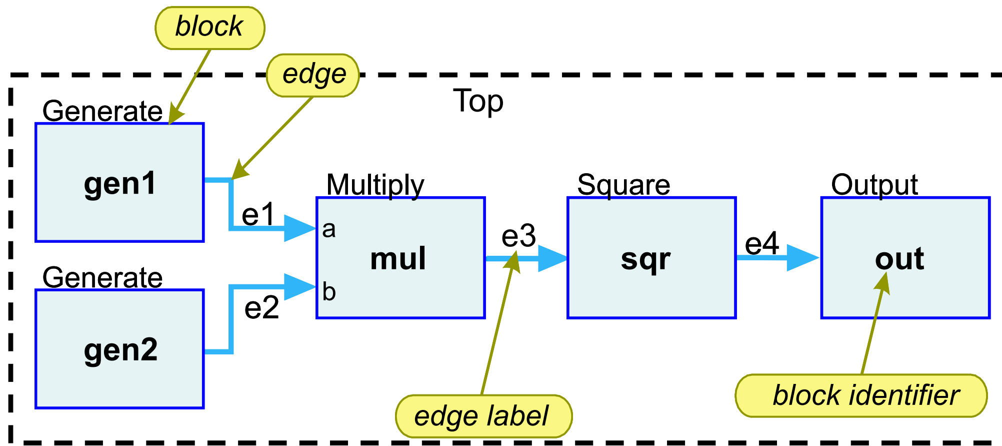

Figure 3: Example *X* data flow graph.

Figure 3 illustrates a compound block `Top` constructed from basic blocks of type `Generate`, `Multiply`, `Square`, and `Output`. The *X* code that describes `Top` is shown in Figure 4. Each of the blocks within `Top` is instantiated, and then their interconnections are specified. While this example uses simple functions such as multiply and square, the true target of blocks are more course-grained computations such as filters, FFTs, and the like.

```
block Top {
    Generate gen1, gen2;
    Multiply mul;
    Square sqr;
    Output out;

e1: gen1 -> mul.a;
e2: gen2 -> mul.b;
e3: mul  -> sqr;
e4: sqr  -> out;
};
```

Figure 4: Example *X* description.

The *X* language also provides language structures to express the hybrid systems themselves. We provide a library of classes of computational resources and the interconnect resources that transmit data between devices. Users can then describe their hybrid systems, both real and hypothetical, in terms of instances of the resource classes.

Once the user has specified an application as a set of connected blocks, and the computing system as a set of interconnected computation resources, the deployment pattern is specified simply by mapping each block to a computation resource.

The *X-Com* compiler tool parses *X* language descriptions of applications, hybrid systems, and their mapping to create a set of compilable source files for each device in the system. These source files, compiled with their respective platform-specific tools (e.g., C compiler, HDL synthesizer, etc.), fully implement the entire application as a distributed set of executables (e.g., one program per processor, one bitfile per FPGA). A second tool called *X-Dep* further automates this step by generating a compilation and deployment script to perform the final linking steps and deploy the application to real hardware devices or simulations (or emulations) of devices.

When deployed to simulation devices, the *X-Sim* federated simulation environment (described further in Section 4) is used to capture detailed performance information useful in understanding the performance of the system. Even when deployed to real hardware, however, the user can choose to capture performance statistics that can be used to further fine-tune the mapping and other application parameters, or to improve the simulation of hypothetical systems.

Figure 5 illustrates the archetypical steps taken in the development, functional simulation, and performance modeling of an application using the *X* language and associated tools. An application description, system description, and mapping (all in *X*) are compiled with *X-Com* targeting the simulator. *X-Sim* is used to execute the simulation and the results are checked for correctness, revising the algorithm as needed. The simulation results are then examined to assess performance, and the mapping can be revised as desired to improve performance. Once the user is satisfied with the mapping, the application is compiled targeting the physical hybrid system.

Further details on the *X* language and compiler may be found in [8] and [22]. The *X-Sim* simulator is described in [10].

4. PERFORMANCE MODELING

Given an application description in the *X* language, a set of block implementations on various computational resources, and a mapping of blocks to resources, the *X-Sim* federated simulation environment can be used to verify functional correctness of the application and estimate performance on the specified computational resources. Key features of *X-Sim* are:

- integration of multiple, potentially very different, simulators into a single federated simulation, and
- automation of the system simulation by coordinating individual simulator runs.

X-Sim provides an environment in which multiple simulators are seamlessly combined into simulate applications

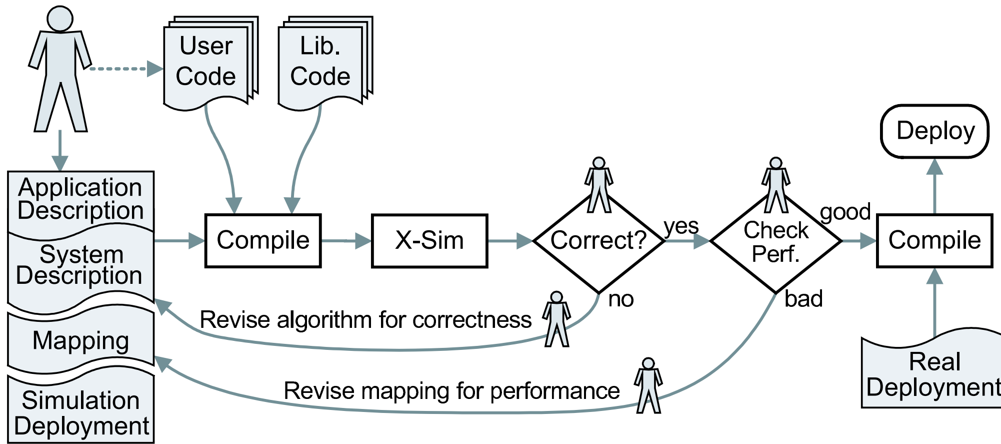

Figure 5: Application development flow.

expressed in X mapped to hybrid systems. The *X-Sim* infrastructure is open-ended to allow support for a range of individual simulators, from low-level, discrete-event and cycle-accurate simulators to rough estimates from analytic models.

X-Sim generates Makefile scripts to automatically compile and simulate applications. The scripts use data dependencies derived from the application description to simulate individual components in order. In acyclic pipelined systems, this allows one to run simulations concurrently on multiple machines, subject to the precedence constraints in the application itself.

Interconnects between computational resources are simulated using an extensible library of communication models. If a more complex communication simulator is available, that may be used instead.

Figure 6 shows how an application with four blocks (A, B, C, D) distributed across two devices looks when simulated with *X-Sim*. Directed arrows depict the flow of data, with inter-device communication using trace data files and intra-device edges using the native communication methods (wires in an FPGA, function calls on a processor). To profile application performance, *X-Sim* keeps track of when data enters and exits individual simulators by maintaining multiple timestamp files (T1, T2, T3) for every interconnect. Interconnect models are used on all inter-device communications to simulate data transmission.

There are three types of timestamps. The first timestamp (T1 in Figure 6) keeps track of when data is output from a computational device onto an interconnect. The second timestamp (T2) indicates when the data has been transmitted across the interconnect and is available to the receiving device. The third timestamp (T3) records the time at which the receiving device consumed the data and started processing it. By maintaining these timestamps, *X-Sim* provides a time trace of all data transfers that occur between computational devices.

Multiple blocks may be mapped to the same computational resource. By default, timestamps are only kept for the data entering and exiting blocks which connect to interconnects.

When they are executed, each simulator generates entry and exit timestamps in their native format. Header information indicates the data format of the timestamps. After all the timestamps are collected, this format information is used to normalize them into a universal time domain. The execution time for a computation event can be calculated by subtracting the time input data entered a device from the time output data exited it. These times may then be used to generate characteristic distributions of execution times for each device.

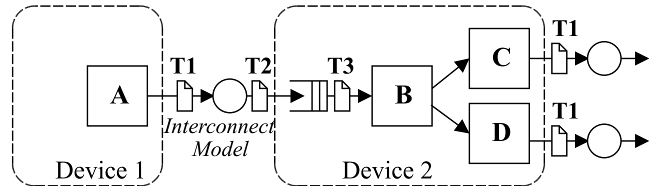

Figure 6: Flow of data within an *X-Sim* simulation.

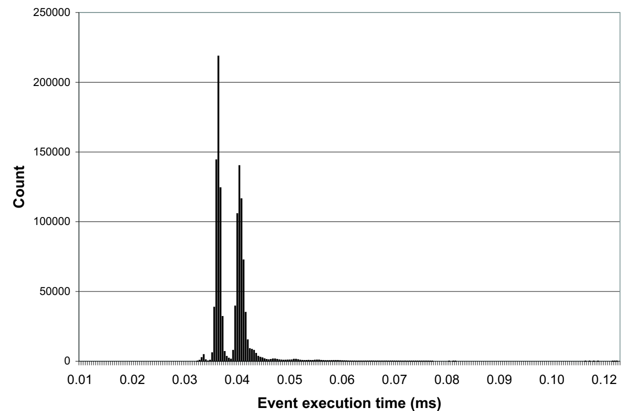

Figure 7: Histogram of event execution times for a sample application task mapped to a general-purpose processor.

For example, the histogram in Figure 7 shows the distribution of execution times measured using X-Sim for a sample application task that was mapped to a general purpose pro-

cessor. The execution times exhibit an interesting bimodal distribution in this case, reflecting the fact that execution times on general purpose processors are rarely truly deterministic, but rather reflect pertubations due to any number of potential causes, from cache misses to external interruptions.

An analysis component obtains basic and advanced performance measurements using the timestamps. Basic measurements include the mean and variance of service time distributions associated with devices. These measurements may be easily aggregated to determine throughput and latency figures for the individual devices and the system as a whole.

More advanced measurements may take the time distribution data and fit it to common analytic distributions (e.g., exponential, gamma, Gaussian) for the development of queuing models. The raw data may also be used in trace-driven analyses. A user may take these simple and complex performance analyses and solve for various performance parameters using analytic methods. Using this capability, alternative mappings may be explored to improve performance.

Currently, *X-Sim* requires that the devices in a simulation be organized in an acyclic pipeline. This is because all intermediate data and timestamp files must be completed before they may be used by the next stage. An advantage of this approach is that the simulator data dependencies are well understood *a priori*. This allows independent simulations to run in parallel. By exposing the data dependencies in the simulation Makefile and using the file system for communication, *X-Sim* is able to easily distribute simulations of large systems across many real computing resources to speed up the simulation task.

5. APPLICATION DEPLOYMENT

Returning to the development flow diagram of Figure 5, once the developer is satisfied with the simulation results, the *X-Dep* tool is then used to deploy the application on the target hardware. Key features of *X-Dep* are:

- physical instantiation of the X blocks onto the computational resources to which they have been assigned via the mapping;
- interconnection of ports on X blocks with FIFO buffers; and
- providing the communications support between X blocks that are assigned to distinct resources.

In effect, *X-Dep* transforms the X language description of the application, machine description, and mapping into a physical system executing the user's program. It does this by providing wrappers for each block that are tailored to the specifics of how the block is mapped. The generated wrapper code provides the input data to each input port, accepts output from the output ports, and moves data as required across interconnect resources for delivery between blocks.

When the block implementations are mapped to a processor, the C code that is generated makes use of the GLib [11] low-level portability and utility library. GLib data structures are used for memory management and queuing to facilitate the underlying data flow organization. GLib data types are used for safety and robust management of nontrivial data structures such as arrays. In particular, the `GArray` structure is used for both constant and variable size arrays.

If two X blocks are mapped to the same processor, the generated interconnection code invokes the downstream block on a `send()` from an upstream block. When two X blocks are mapped to distinct processors, socket-level interprocess communication is invoked.

For X blocks that are mapped to an FPGA, there exist generated wrappers that reside both on the FPGA itself and the processor to which the FPGA is physically attached. In the prototype system currently in use, the FPGA is positioned on the PCI-X bus. Using techniques described in [4], data destined for input ports on blocks mapped to the FPGA are moved across the PCI-X bus via a DMA transfer to physical FIFOs on the FPGA directly wired to the input ports of the X block's implementation. Correspondingly, data from output ports are moved across the PCI-X bus via DMA back into processor memory, where the C wrapper either invokes the downstream block (if it is mapped to that processor) or delivers the data to the appropriate computational resource using the above-mentioned socket-level communication mechanisms.

While the above describes the deploying of an application to target hardware, the *X-Dep* tool also has responsibility for deploying the application to the *X-Sim* simulation environment as well. In this case, the generated wrappers use the filesystem to manage the data into and out of ports, reading data from trace files for input ports and writing data to trace files from output ports. These wrappers also create the timestamp files described in the previous section.

6. APPLICATION MAPPING AND PERFORMANCE EVALUATION

To demonstrate the mapping, performance evaluation, and execution of an application onto a hybrid system, we examine a scientific application called "high-energy gamma ray event parametrization." In this section we describe this application which has been implemented in the X language, present a set of different mappings of the application onto software and hardware resources, and evaluate the mappings' performance both in the simulator and deployed on the actual hardware.

6.1 VERITAS Application

A common experiment in high-energy astrophysics is the examination and characterization of gamma rays generated by extraterrestrial sources. Astrophysicists believe these sources may include pulsars, supernovae, neutron star collisions, and supermassive black holes in galactic nuclei.

The event parametrization application is a computationally intensive step in the ground-based detection of stellar gamma ray sources. Gamma rays striking the atmosphere result in showers of thousands of photons called Cherenkov radiation. In astrophysics experiments such as VERITAS [23] and HESS [15], these photons are reflected by large (10–17 m) mirrors onto arrays of hundreds of photomultiplier tubes. The photomultiplier tubes transduce the Cherenkov photons, along with the unwanted diffuse background light, to high-voltage analog waveforms. These waveforms are then recorded by fast analog-to-digital converters at sampling rates surpassing 500 MHz. We concentrate on the signal processing that is performed on the digitized waveforms

to improve the signal-to-noise ratio of Cherenkov photons above background light, and the image processing that characterizes the resulting images to discover features indicative of gamma rays and other cosmic rays.

Our application, which has been configured for the VERITAS telescopes, is depicted in Figure 8. It consists of a configurable number (N in the figure) of signal processing pipelines which process each of the digitized waveforms from the 499 photomultiplier tubes. Each pipeline pads the input to a higher resolution, and performs two frequency-domain filters on the signal. The first filter, a low-pass, interpolates the signal to correct for the padding. The second filter performs a deconvolution to correct for signal "smear" introduced by the analog electronics and improve the signal-to-noise ratio of the individual high-energy photons over the background. After filtering and subtracting the background light level, the charges for each event are measured and compared to a threshold. From these charge results, the first and second moments of the telescope image are calculated. These moments are then used to determine parameters for the image corresponding to various shapes (e.g., filled and hollow ellipses) commonly generated by cosmic particles.

6.2 Performance Analysis Using X-Sim

For the experimental results in this section, the VERITAS pipeline computation of Figure 8 is configured to process a stream of events. Each event corresponds to a 499-pixel image, with each pixel waveform containing 24 samples from the original A/D converters. Processing a sample set of 5,000 events on an individual processor (a 2 GHz AMD Opteron) required 255 s. We have intentionally reduced the total compute requirements for the sample problem for illustration purposes. The production version processes millions of events, instead of a set of 5,000 events as used in our experiments.

X language blocks have been developed for the following kernels: zero pad, FFT, lowpass filter, deconvolution filter, IFFT, pulse area, leading edge, pulse width, threshold test, moment calculation, and image parameter calculation. All of the blocks have C implementations, and zero pad through IFFT also have VHDL implementations.

The VERITAS pipeline was described in *X*, and the performance of two distinct mappings were investigated using *X-Sim*. First, the pipeline was partitioned into three stages, illustrated by the horizontal lines in Figure 8. Second, stages 1 and 3 were mapped to general-purpose processors (the same type as the serial execution example above). Finally, state 2 was alternately mapped to a general-purpose processor (we call this mapping 1) and an FPGA (mapping 2). Table 1 summarizes these two mappings.

Table 1: Stage to resource mappings for VERITAS pipeline

	Mapping 1	Mapping 2
Stage 1	processor	processor
Stage 2	processor	FPGA
Stage 3	processor	processor

Mapping 1 was deployed on *X-Sim* and the following performance information was extracted:

- Total execution time is 165 s, representing a speedup of only 1.54 over the single processor case.
- Mean event execution time on stage 1 is 13.3 ms.
- Mean event execution time on stage 2 is 33.0 ms.
- Mean event execution time on stage 3 is 11.2 ms.

The mean event execution times given in the table represent the average time taken by each processor core in processing an event, but does not include any estimate for the time required for IPC data transfer between the cores. This was done deliberately to state only empirical performance measurements, and to exclude any subjective estimates for communication delays. The total execution time, 165 s, for the simulated mapping is consistent with taking the bottleneck time, 33.0 ms, and multiplying it by the number of events, 5,000.

The obvious disparity of event execution times on the three stages is clearly the primary cause of the limited performance gain; however, stage 2 is subsequently mapped to an FPGA, which will perform much better. Deploying mapping 2 on *X-Sim*, the following performance information results:

- Total execution time is now 66.5 s, representing a speedup of 3.8 over the single processor case.
- Mean event execution time on stage 1 is 13.3 ms.
- Mean event execution time on stage 2 is 7.9 ms
- Mean event execution time on stage 3 is 11.2 ms.

Here, the performance of stage 2 has been accelerated by a factor of 4.2 by deploying it on an FPGA. The overall performance, however, has only increased to the point where stages 1 and 3 are now the bottleneck.

Clearly, further performance gains rely on the ability to move some of the computational load off of stages 1 and 3. One approach to doing this would be to deploy some of the blocks currently mapped to the stage 3 processor onto the FPGA. This option is viable if an FPGA implementation is available for the blocks currently mapped to the stage 3 processor and there is available space on the FPGA itself. A second approach is to decompose stages 1 and/or 3, using a pair of processors, possibly mapping half of the pixel pipelines to one processor and the other half to another processor. For each of the above approaches, it is straightforward to examine the resulting performance implications using *X-Sim*.

6.3 Performance Results on the Hybrid System

Both configurations from section 6.2 were deployed to a hybrid system containing four processors and an FPGA development board. These devices were connected in the manner described in section 5.

Mapping 1 was deployed on the hybrid system and the following performance information was extracted:

- Total execution time is 260 s, representing a negligible slowdown over the single processor case. This is due to the introduction of significantly increased communication overhead, experienced strongly by stage 2.
- Mean event execution time on stage 1 is 15.9 ms.
- Mean event execution time on stage 2 is 52.0 ms.

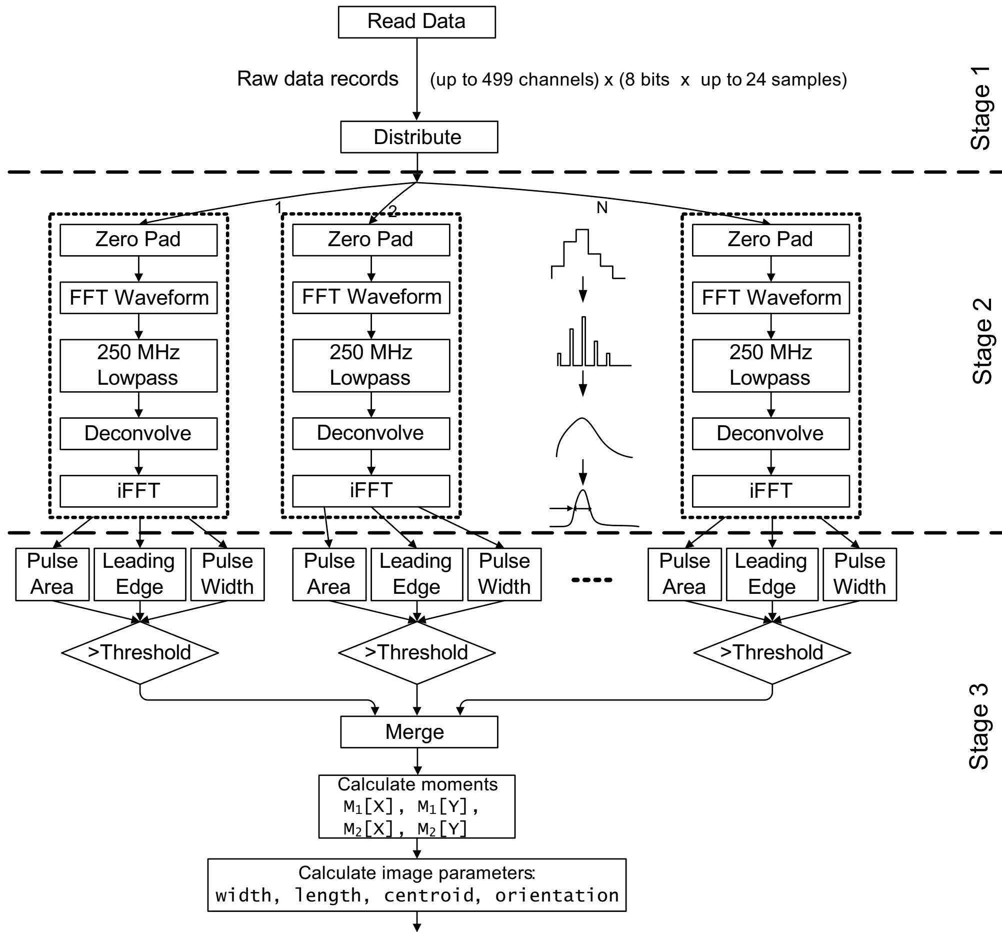

Figure 8: The VERITAS pipeline for Cherenkov image parametrization.

- Mean event execution time on stage 3 is 13.4 ms.

Each of these times are higher than the measurements for the simulated system, but a part of the discrepancy can be attributed to communication delays that were not accounted for in the simulation. However, the measured time for stage 2 when deployed is 52.0 ms, significantly higher than the simulated mapping's measurement of 33.0 ms. This is due to the small default size of the Linux IPC queues; these queues are used for the high-volume communication between processors and thus they fill up quickly. This causes the process to become blocked, which adds significantly to the measured processing delay. Future work will improve this communication link with a customizably large queue implemented in shared memory.

Deploying mapping 2 on the hybrid system, the following performance information results:

- Total execution time is now 133 s, representing a speedup of 1.9 over the single processor case.
- Mean event execution time on stage 1 is 24.1 ms.
- Mean event execution time on stage 3 is 15.9 ms.

Stage 2 (on the FPGA) was not configured to record performance information, however based on the other stages' results, it was not the bottleneck in the system. In this configuration, stage 1 has become the bottleneck.

The stage 3 time is 15.9 ms, slightly higher than the simulation prediction of 11.2 ms. Once again, communication delay can reasonably be blamed for a substantial portion of this. However, the stage 1 time for the deployed system is 24.1 ms, significantly higher than the simulated time of 13.3 ms. The probable reason for this is that the communication mechanism being used for communication over the PCI-X bus has a high overhead penalty for small data transfers.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Hybrid systems are capable of improving application performance by several orders of magnitude, albeit with enormous programmer effort. Our aim is to simplify the programming of such systems, and to that end we are building an application development environment that supports the flexible mapping of application components onto computational resources and the automatic delivery of data between these computational resources.

An important component of this development environment is the emphasis placed on performance assessment and evaluation. The major purpose for deploying applications on hybrid systems is to exploit the performance gains achievable, and enabling the application developer to observe the

performance implications of design choices is crucial to efficient use of developer time.

7.2 Current Status and Future Work

The current status of the system is as follows:

- *X-Com* The compiler is functional, supporting the transformation of X dataflow descriptions and C/C++ or HDL block implementations into the set of binaries and/or bitfiles needed for execution. Future work includes the expansion of supported native languages to include Brook for GPU support.

- *X-Dep* The deployment tool currently only supports deployment to physical processors, FPGAs, and to the simulator (including processor and FPGA simulations). We are currently expanding this to include GPUs connected via PCIe.

- *X-Sim* The federated simulator currently supports native execution of blocks implemented in C/C++ on a traditional processor and Modelsim simulation of blocks implemented in HDL targeted at an FPGA. Future work includes a processor simulator (e.g., Simplescalar) and a wider set of communication models.

- *Overall* Verification and Validation (V&V) of the performance predictions made by the federated simulator are currently underway. While we are confident of simulation correctness (verification), clearly improvement is needed in the area of performance values (simulation predictions and empirical execution time).

Our vision is that the development environment be accessible by a variety of users, not just in the computer science community but in the wider scientific community as well, enabling the effective exploitation of hybrid computer architectures across a large application set. To enable this vision to become a reality, we must ensure that:

1. the system can generate fully executable images on a number of physical hybrid systems,

2. the performance model predictions are both understandable and reliable,

3. a large collection of usable blocks are implemented on more than one computational resource,

4. a collection of developers are enabled to design additional block implementations, and

5. the entire system is well supported and well documented.

We are doing all we can to ensure all of the above takes place, enabling the use of hybrid systems on many real-world problems.

8. ACKNOWLEDGMENTS

This research has been supported in part by National Science Foundation grant CCF-0427794.

9. REFERENCES

[1] L. Atieno, J. Allen, D. Goeckel, and R. Tessier. An adaptive Reed-Solomon errors-and-erasures decoder. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 150–158, 2006.

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

[3] 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.

[4] R. D. Chamberlain, B. Shands, and J. White. Achieving real data throughput for an FPGA co-processor. In *Proc. of 1st Workshop on Building Block Engine Architectures for Computers and Networks*, Oct. 2004.

[5] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 213–222, Feb. 2004.

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

[7] M. A. Franklin, R. D. Chamberlain, M. Henrichs, B. Shands, and J. White. An architecture for fast processing of large unstructured data sets. In *Proc. of IEEE 22nd Int'l Conf. on Computer Design*, pages 280–287, Oct. 2004.

[8] 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.

[9] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, pages 3–14, 2005.

[10] 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*, pages 198–207, June 2007.

[11] The Glib Team. GLib Reference Manual. `http://developer.gnome.org/doc/API/glib`.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of SIGMOD Int'l Conf. on Management of Data*, pages 325–336, 2006.

[13] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 162–170, Feb. 2004.

[14] M. C. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt. Single pass, BLAST-like, approximate

string matching on FPGAs. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 217–226, 2006.

[15] W. Hofmann, for the H.E.S.S. Collaboration. Status of the high energy stereoscopic system (H.E.S.S.) project. In *Proc. of 27th Int'l Cosmic Ray Conf.*, pages 2785–2788, 2001.

[16] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A streaming HMMer-search implementation. In *Proc. of ACM/IEEE Conf. on Supercomputing*, pages 11–19, 2005.

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

[18] National Instruments. Labview. `http://www.ni.com/labview`.

[19] The Ptolemy Team. The Ptolemy Kernel – Supporting Heterogeneous Design. *RASSP Digest Newsletter*, 2(1):14–17, Apr. 1995.

[20] R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna. Hardware/software approach to molecular dynamics on reconfigurable computers. In *Proc. of 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 23–34, 2006.

[21] J. L. Tripp, H. S. Mortveit, A. A. Hansson, and M. Gokhale. Metropolitan road traffic simulation on FPGAs. In *Proc. of 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 117–126, 2005.

[22] E. J. Tyson. Auto-pipe and the X language: A toolset and language for the simulation, analysis, and synthesis of heterogeneous pipelined architectures. Master's thesis, Washington University in St. Louis, Department of Computer Science and Engineering, 2006.

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

[24] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee. Overview of the FREEDOM compiler for mapping DSP software to FPGAs. In *Proc. of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 37–46, 2004.