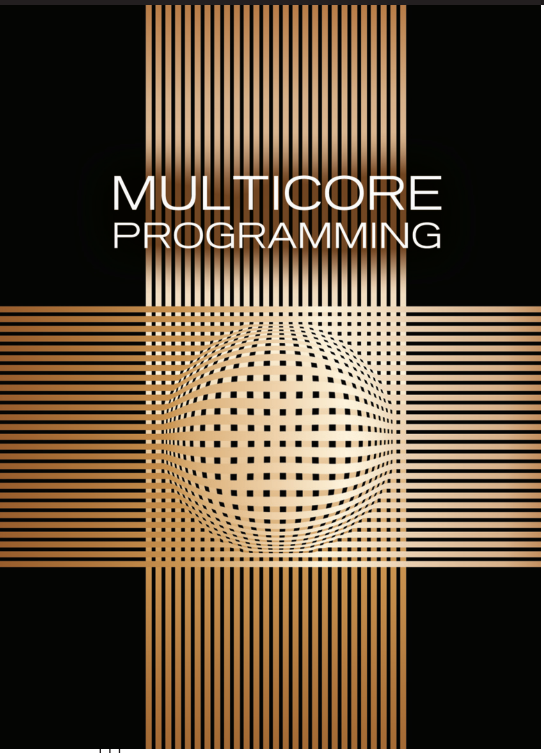

AUTO-PIPE: STREAMING APPLICATIONS ON ARCHITECTURALLY DIVERSE SYSTEMS

Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, and E.F. Berkley Shands, *Washington University in St. Louis*
Naveen Singla, *Exegy Inc.*

Auto-Pipe, an application development environment for streaming applications executing on architecturally diverse computing platforms, supports the flexible mapping and automatic delivery of application components between computational resources.

The recognition and subsequent exploitation of streaming data semantics in applications can dramatically simplify the development process. Using the streaming data paradigm, the expression of available parallelism is clarified, the opportunities to inadvertently introduce races and synchronization errors are fewer, and ensuring correct execution is therefore easier. Brook[1] and StreamIt[2] are examples of languages that support the direct expression of streaming data semantics. Computation blocks, called *kernels* in Brook and *filters* in StreamIt, communicate via explicitly defined edges that move data between blocks in a fixed topology specified at compile time. While not every application is amenable to streaming data formulation, a large number of applications fall into this class.

In recent years, several computational resource types have matured to the point that they can materially benefit high-performance applications. These resources include multicore general-purpose processors, reconfigurable hardware, 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, and performance improvements of one to two orders of magnitude have been reported.[3,4]

Systems built out of these resources are architecturally diverse, and while constructing hardware prototypes that include diverse compute resources is straightforward, application development for such systems is quite difficult for several reasons:

- In most cases, each compute resource has its own language, development environment and tools, runtime environment, and debugging aids.
- The intellectual task of describing the computation is often quite different for each compute resource. For example, developers typically use task-level threads to program general-purpose processors and chip multiprocessors, while they program reconfigurable hardware at the register transfer level.

- Delivering data between these disparate environments is a significant task in its own right.

The result is that, while substantial performance gains are achievable using diverse systems, these gains are only achievable with enormous effort. Our aim is to simplify the development and deployment of streaming applications onto diverse systems. This includes representation of such applications and the available compute resources, mapping of application components onto the resources, providing a mechanism for evaluation of application performance, and, finally, deployment onto the diverse system and application execution.

To tackle these design issues, we constructed Auto-Pipe,[5] a development environment for streaming applications executing on architecturally diverse computing platforms. Our approach involves the use of a coordination language to specify streaming data communications between compute blocks combined with native languages and toolsets for the development of the compute blocks themselves. The environment supports evaluating application performance early in the design cycle, mapping of compute blocks to computational resources, and providing 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 (for example, application pipeline stages) and the existence of implementations (potentially for 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 to 3). These stages might represent, for example, 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 a pair of candidate mappings, with application stage 2 mapped either to platform 1 or 2.

While the figure illustrates a particular design question, a full design problem presents many such questions. For example, what technology should be used for compute platform 1 (for example, processor core or reconfigurable logic)? How does this choice impact the mapping question for application stage 2? The Auto-Pipe application development environment helps developers answer these questions, while keeping them cognizant of the performance implications of their design decisions.

ARCHITECTURALLY DIVERSE SYSTEMS

Architecturally diverse (or hybrid) computing systems incorporate two or more distinct computational resource types (or platforms) including the following:

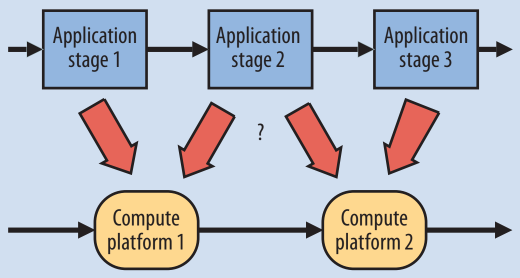

Figure 1. Mapping an application to an architecture. The application's three pipelined stages are mapped to two compute platforms. Application stage 1 is mapped to compute platform 1, application stage 3 is mapped to compute platform 2, and there is a question as to whether application stage 2 should be mapped to compute platform 1 or 2.

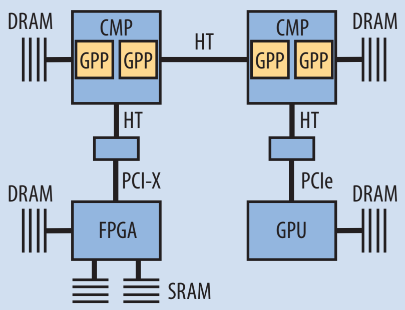

Figure 2. Example of an architecturally diverse system architecture. Two chip multiprocessors (CMPs) are interconnected with a HyperTransport (HT) link. Additional HT links are used to connect to an FPGA through a PCI-X bus and a graphics processing unit (GPU) via a PCIe bus. In this system, the memory attached to the two CMPs is cache-coherent and shared across the four GPP cores, while the memories attached to the FPGA and the GPU each form separate address spaces.

- homogeneous, multicore, general-purpose processors (GPPs)—for example, x86 processors;
- heterogeneous multicore processors, which provide processors of varying capability within a single chip—for example, the IBM Cell processor and network processors;
- graphics processing units (GPUs); traditionally aimed at visual rendering, these processors are now being used for a wide variety of purposes;
- reconfigurable hardware in the form of field-programmable gate arrays (FPGAs); and
- digital signal processors (DSPs) or other application-specific instruction processors (ASIPs)—processors for which the instruction set or architecture has been optimized for an individual application or class of applications.

Figure 2 shows an example of an architecturally diverse system constructed using dual-core AMD Opterons, an off-the-shelf graphics card, and an FPGA card. This is but one example out of many ways in which such a system can be constructed.

While constructing this type of system is fairly straightforward, it is difficult to develop applications that can effectively exploit its capabilities. Distinct computational resource types typically have their own languages for describing applications. For example:

- multicore processors—C/C++ with thread-based or message-passing parallelism for homogeneous cores, specialized constructs (often including native assembly language) for heterogeneous cores;
- GPUs—stream programming languages such as Brook or APIs such as CUDA;
- FPGAs—hardware description languages such as Verilog, VHDL, and SystemC; and
- DSPs and ASIPs—C/C++ and assembly language.

Explicit coordination languages have been used in directing execution of software modules and in enhancing software reusability.

Associated with each of these languages is a distinct toolset that includes compilers (or synthesizers), runtime environments, and debugging aids. Note that there is little support in these languages and toolsets for enabling data delivery between different types of computational resources. Auto-Pipe's focus is on enabling designers to develop high-performance applications that run correctly despite the above limitations.

AUTHORING STREAMING APPLICATIONS ON DIVERSE SYSTEMS

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

Our approach is to take advantage of these relatively efficient, robust, and well-entrenched languages by designing a coordination language called X that is capable of connecting task kernels—written in traditional languages—in a data streaming manner. Each kernel or block may have several platform-specific implementations (for example, ANSI C, CUDA, VHDL, and so forth). All implementations of a given block are required to provide the same interface and streaming data semantics, thus ensuring correctness regardless of the block-to-resource mapping. Each supported language has a specific API and syntax for specifying the particular data streaming interface employed by the block, such as input ports (including data type), output ports, and configuration parameters.

Coordination languages

Coordination languages have been developed in several contexts. Edward A. Lee argued that coordination languages represent a better mechanism for reasoning about concurrency than traditional thread-based approaches.[6] Both Brook[1] and StreamIt[2] are languages tailored to streaming applications for homogeneous compute platforms where coordination is inherent in the language definition. Explicit coordination languages have been used in directing execution of software modules and in enhancing software reusability.[7]

David Gelernter and Nicholas Carriero discussed the inherent separation between computation and coordination (they used the term "synchronization") and the advantages associated with explicitly separating the two.[8] This separation is present in their Linda language.[9]

Our X language follows these ideas in many ways in that it permits representation of algorithms in terms of coordination of blocks that communicate with each other where the computation language associated with the blocks is separate and may be one of a host of languages. However, in the case of X, these blocks may be mapped onto diverse compute resources.

X language benefits

There are several benefits to authoring applications using this approach. First, it is possible to build a library of blocks that can be (re-)used to enable application development either entirely (or at least primarily) in the coordination language without requiring implementation of individual blocks. This is analogous to the use of numerical libraries such as BLAS for authoring scientific applications. Base solvers are typically not recoded, but application developers call them from the appropriate libraries. This also follows the rationale behind much of the work in the software-only domain referenced above.

Second, X provides the underlying structure so that the application developer doesn't need to code data movement and synchronization between blocks. The X language permits specification of data movement between blocks at a high level, thus relieving the application programmer from tedious coding requirements. The X runtime infrastructure automatically delivers block A's output to block B's input. This delivery is independent of whether block A and block B are deployed on a common resource or distinct resources, independent of whether block A and block B have a common memory subsystem or must use other data delivery mechanisms (for example, a network), and independent of whether block A and block B are even executing on the same type of computing component.

Third, with explicit knowledge of the algorithm decomposition known to the system, expressing the mapping of blocks to compute resources for deployment and execution proceeds naturally.

Fourth, reasoning about the correctness of streaming applications is fairly straightforward, thus diminishing the chances of programming errors (either design or implementation errors) that are difficult to detect and debug. Contrast this with the complexity of correcting a synchronization error due to a missing lock in a shared-memory program.

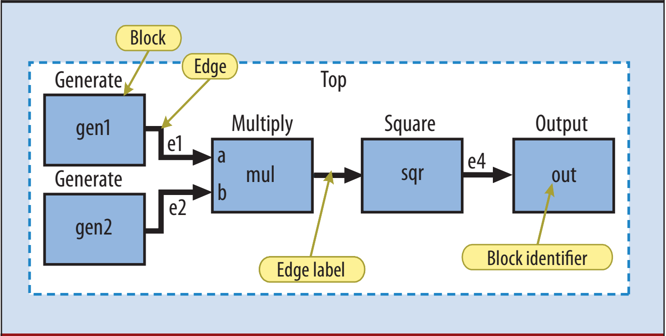

Figure 3. Example X application showing streaming data coordination. Block identifiers are shown within the blocks, and block types are shown above the blocks. Optional edge labels are shown next to the edges.

Example streaming application

Figure 3 illustrates the use of the X coordination language with an example streaming application. The figure defines a compound block Top constructed from basic blocks of type Generate, Multiply, Square, and Output. Figure 4 shows the X code that describes Top. Associated with each basic block is a set of block implementations. The definition of Top comprises two main portions: The first specifies the computational blocks to be used, while the second specifies the connections between the blocks. Each block references an implementation that supports block execution on a computational resource type and is coded in the language appropriate for that resource (note that certain languages may execute on multiple resource types). While this example uses simple functions such as multiply and square as basic blocks, blocks are more typically course-grained computations such as filters, FFTs, matching, and so on.

The Top portion of the X code also describes application block communications. Edges (->) in the application description convey the delivery of a data stream from an output port on an upstream block to an input port on a downstream block. At the application level, these can be considered to be strongly typed FIFO channels that preserve order between data elements. A directed acyclic graph can formally represent the overall topology.

The X language also provides for specification of the physical architecture of the diverse system being considered as well as the data types (integer, floating, and so on) being communicated. A library of classes of computational resources and the interconnect resources that transmit data between devices is provided. Users can then describe their architecturally diverse systems (not shown here), both real and hypothetical, in terms of instances of the resource classes. After specifying an application as a set of connected blocks, their implementations, and their interconnections, the user specifies the deployment/mapping by noting the placement of each block on a computational resource (last two lines of Figure 4).

```
// algorithm description
block Top {        // instantiate block types
Generate gen1, gen2; // data generation blocks
Multiply mul;      // multiply block
Square sqr;      // square block
Output out;      // output block

e1: gen1 -> mul.a; // gen1’s output connected to input port “a” of mul
e2: gen2 -> mul.b; // gen2’s output connected to input port “b” of mul
e3: mul -> sqr;    // output of mul connected to input of sqr
e4: sqr -> out;    // output of sqr connected to input of out
};

// application mapping
map proc[1] = {gen1, gen2, mul}; // gen1, gen2, and mul mapped to proc[1]
map proc[2] = {sqr, out};  // sqr and out are mapped to proc[2]
```

Figure 4. Example X description. Each of the blocks within Top is instantiated, and then their interconnections are specified. Data types of input ports, output ports, and stream edges are given in the definitions of the block types (not shown in the figure). Blocks gen1, gen2, and mul are mapped to one processor and the remaining blocks to a second processor.

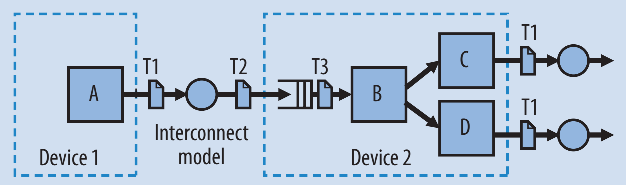

Figure 5. Data flow within an X-Sim simulation. Block A is mapped to device 1, while blocks B, C, and D are mapped to device 2. Time stamps labeled T1 keep track of when the application block outputs data from a device onto an interconnect. T2 time stamps indicate when the data has been transmitted across the interconnect and is available to the receiving device. T3 time stamps record the time at which the application block on the receiving device consumes the data.

X-Com and X-Dep

The X-Com compiler parses the X language descriptions of applications, diverse systems, and their mapping to create a set of source files that can be compiled for each device in the system. These source files, compiled with their respective platform-specific tools (for example, C compiler, HDL synthesizer, and so on), fully implement the entire application as a distributed set of executables (for example, one program per processor, one bit file per FPGA). The X-Dep tool further automates this step by generating a compilation and deployment script to perform the final linking steps and deploy the application to the physical hardware devices or simulations (or emulations) of devices.

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, application developers can use our X-Sim federated simulation environment[10] to verify functional correctness of the application and estimate performance on the specified computational resources. X-Sim provides an environment that seamlessly combines multiple simulators into one federated execution to simulate applications expressed in X. 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.

Figure 5 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 data flow, with interdevice communication captured and stored in trace data files, and intradevice edges determined using the simulated native communication methods (wires in an FPGA, function calls on a processor core).

To profile application performance, X-Sim keeps track of the data values and when the data enters and exits individual block-level simulators. X-Sim maintains multiple timestamp files (T1, T2, T3) for every interconnect. Interconnect models are used on all interdevice communications to simulate data transmission. These communication models may be as simple as a fixed delay model, or may be arbitrarily complex, trace-driven, discrete-event simulation models developed from first principals or through use of the trace data. By maintaining these time stamps, 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. However, by default, time stamps are only kept for the data entering and exiting blocks that connect to distinct computational resources.

An analysis component obtains basic and advanced performance measurements using the time stamps. Basic measurements include the mean and variance of service time distributions associated with devices. The user can aggregate these measurements to determine throughput and latency figures for the individual devices and the system as a whole.

As with any simulation, it is often impractical to execute voluminous data sets in the simulator. This is particularly true when saving complete traces of data streams between blocks. As a result, it is incumbent upon the user to judiciously choose an appropriate input data subset that is reasonably characteristic of the overall input data set.

APPLICATION DEPLOYMENT

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,
- instantiation of FIFO buffers on X block interconnection ports, and
- providing 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 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.

If two X blocks are mapped to the same processor core, the generated interconnection code directly invokes the downstream block upon data output from an upstream block. When two X blocks are mapped to distinct processor cores, the interblock communication mechanism depends upon the underlying infrastructure available. If the two cores have a common memory subsystem (as is the case if they are on the same chip), the Auto-Pipe runtime system uses a shared memory buffer to move data from the upstream block's output port to the downstream block's input port. If the two cores do not have common memory (for example, if they are connected via a local-area network), the runtime system invokes socket-level interprocess communication.

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. For FPGA-to-FPGA communication, a simple FIFO is instantiated on the FPGA. For FPGA-to-processor communication, the FPGA is positioned on the PCI-X bus. 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 is moved across the PCI-X bus via DMA back into processor memory, where the C wrapper invokes the downstream block.

While the above describes the deployment of an application to target hardware, the X-Dep tool also has responsibility for deploying the application to the X-Sim simulation environment. In this case, the generated wrappers use the file system 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 time stamp files. In the simulator, the actual block execution is dependent on the type of computational resource being modeled. For FPGAs, we use the Modelsim simulator, and for processors we use direct execution on an individual representative processor core.

COMPUTATIONAL FINANCE APPLICATION

To demonstrate the mapping, performance evaluation, and execution of applications in the Auto-Pipe environment, we present a computational finance application that expands the compute platforms to include graphics engines. Since the Auto-Pipe environment is currently being expanded to include graphics engines, we developed this application using a combination of Auto-Pipe and CUDA.

An important application in computational finance is the calculation of value at risk (VAR). The VAR is an indicator of the risk associated with a portfolio of financial instruments. It is defined as the maximum loss that is not exceeded with a given probability over a specified period of time. The probability is specified as a confidence level. The two confidence levels frequently used in practice are 95 percent and 99 percent. For example, a VAR of $10,000 at the 95 percent confidence level indicates that the probability that the losses will exceed $10,000 is less than 0.05.

The application calculates the VAR by estimating a portfolio's value at the end of a specified time. Since stochastic processes drive the underlying models for pricing financial instruments, a distribution for the value of the portfolio is obtained at the end of the specified time. The standard Black-Scholes model for price dynamics of financial instruments (for example, stocks) is used.[11]

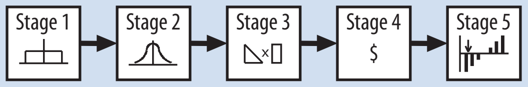

Figure 6. Computation pipeline for financial Monte Carlo simulation.

The Monte Carlo approach to VAR calculation involves simulation of the portfolio's value at the end of the specified time.[12] The differences between the value of the starting portfolio and the simulated future portfolios provide estimates of the profit and loss (P&L) over the specified time. The VAR then is simply the appropriate value of the sorted P&L estimates.

Simulating the values of the components of a portfolio under the Black-Scholes model requires generating correlated Gaussian random numbers and propagating them forward under the model. The VAR can then be calculated from the resulting distribution. Figure 6 shows the functional pipeline for this simulation.

The pipeline stages are as follows:

- *Stage 1:* Uniform pseudorandom number generation—the Mersenne twister[13] is used to generate random numbers that are uniformly distributed between 0 and MAXINT ($2^{32} - 1$).
- *Stage 2:* The uniformly distributed random numbers are transformed into a Gaussian (normal) distribution with $\mu = 0$ and $\sigma^2 = 1$.
- *Stage 3:* The vector of independent normally distributed random numbers is transformed into a vector of correlated random numbers reflecting the correlations between individual financial instruments (for example, stocks) in the portfolio. This is accomplished by multiplying the vector by a lower triangular matrix. This lower triangular matrix is obtained by the Cholesky factorization of the specified correlation matrix.
- *Stage 4:* The correlated Gaussian random numbers are used to generate random walks according to the Black-Scholes model. The portfolio's values and the P&L values are also calculated in this stage.
- *Stage 5:* The P&L values are aggregated and sorted to obtain the VAR.

With the exception of stage 5, each of these stages can be executed in a data-parallel manner. While the Auto-Pipe development environment does not yet directly support mapping of blocks to a GPU, we deployed this application

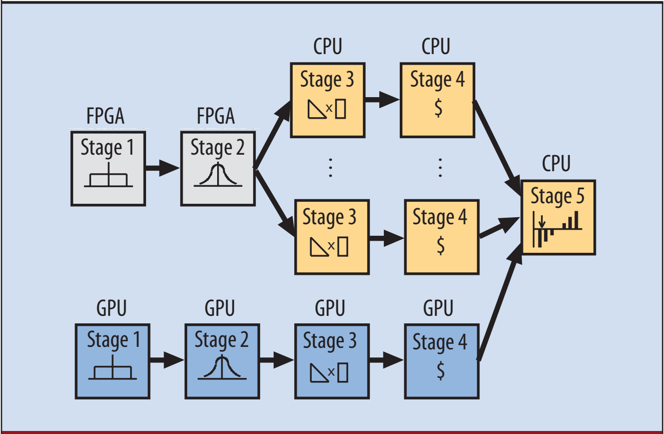

Figure 7. Financial Monte Carlo simulation deployed on eight Opteron processor cores, a Xilinx Virtex-4 FPGA, and an Nvidia GTX 260 GPU. The FPGA and CPU portions of the implementation are in Auto-Pipe, and the GPU portion is coded using CUDA.

to a set of processor cores, an FPGA, and a GPU using a combination of Auto-Pipe and CUDA. Figure 7 illustrates the highest performing mapping.

We considered a portfolio of 1,024 stocks and nominal values for the parameters in the Black-Scholes model. These parameters are uniquely available from the data stream provided by an Exegy XTP ticker plant (www.exegy.com). Obtaining a single value of the portfolio at the end of the specified time requires 1,024 random walks, one for each stock. This then constitutes a "trial" for the Monte Carlo simulation. We simulated the portfolio 2^{20} times, resulting in 2^{30} random walks.

System performance is measured in terms of random walks per second. On an individual processor, the Monte Carlo simulation can execute 450,000 walks/s. Executing the configuration of Figure 7, the Monte Carlo simulation executed 81 million walks/s. This represents a speedup of 180×. To our knowledge, this was the first use of both GPUs and FPGAs in the acceleration of an individual application.

While the Auto-Pipe development environment does not yet directly support mapping of algorithm blocks to the GPU, we are currently using the lessons learned in this application development to expand Auto-Pipe to explicitly include graphics engines as deployment targets.

Architecturally diverse systems can improve streaming application performance by orders of magnitude, albeit with enormous programmer effort. To simplify the programming of such systems, we have constructed the Auto-Pipe application development environment, which 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 diverse systems is to exploit the achievable performance gains. Our goal is to enable the application developer to observe the performance implications of design choices and to reduce application development time.

In addition to the computational finance application, Auto-Pipe has been used to implement applications ranging from cryptography[5] to astrophysics.[14] Auto-Pipe currently supports applications deployed on chip multiprocessors and FPGAs, and we are expanding its scope to include graphics engines. In addition, a block library is under development. In the future, we plan to investigate the incorporation of analog computation (for example, via field-programmable analog arrays[15]) into the application. C

Acknowledgments

This research has been supported in part by National Science Foundation grants CCF-0427794, CNS-0720667, CNS-0751212, CNS-0905368, and DGE-0538541. M.A. Franklin and R.D. Chamberlain are principals in Exegy Inc. The authors also thank Nvidia for its support.

References

1. I. Buck et al., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Trans. Graphics*, Aug. 2004, pp. 777-786.
2. W. Thies, M. Karczmarek, and S.P. Amarasinghe, "StreamIt: A Language for Streaming Applications," *Proc. 11th Int'l Conf. Compiler Construction*, Springer-Verlag, 2001, pp. 179-196.
3. T. El-Ghazawi et al., "The Promise of High-Performance Reconfigurable Computing," *Computer*, Feb. 2008, pp. 69-76.
4. J.D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, 2007, pp. 80-113.
5. M.A. Franklin et al., "Auto-Pipe and the X Language: A Pipeline Design Tool and Description Language," *Proc. Int'l Parallel and Distributed Processing Symp.*, IEEE CS Press, 2006, pp. 1-10.
6. E.A. Lee, "The Problem with Threads," *Computer*, May 2006, pp. 33-42.
7. R. Prieto-Diaz and J.M. Neighbors, "Module Interconnection Languages," *J. Systems and Software*, Nov. 1986, pp. 307-334.
8. D. Gelernter and N. Carriero, "Coordination Languages and Their Significance," *Comm. ACM*, Feb. 1992, pp. 97-107.
9. N. Carrier and D. Gelernter, "Linda in Context," *Comm. ACM*, Apr. 1989, pp. 444-458.
10. S. Gayen et al., "A Federated Simulation Environment for Hybrid Systems," *Proc. 21st Int'l Workshop Principles of Advanced and Distributed Simulation*, ACM Press, 2007, pp. 198-207.
11. P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer, 2004.

12. N. Singla et al., "Financial Monte Carlo Simulation on Architecturally Diverse Systems," *Proc. Workshop High-Performance Computational Finance*, IEEE CS Press, Nov. 2008, pp. 1-7.
13. M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Trans. Modeling and Computer Simulation*, Jan. 1998, pp. 3-30.
14. E.J. Tyson et al., "Acceleration of Atmospheric Cherenkov Telescope Signal Processing to Real-Time Speed with the Auto-Pipe Design System," *Nuclear Instruments and Methods in Physics Research A*, Oct. 2008, pp. 474-479.
15. P. Hasler and T.S. Lande, "Overview of Floating-Gate Devices, Circuits, and Systems," *IEEE Trans. Circuits and Systems II, Analog and Digital Signal Processing*, Jan. 2001, pp. 1-3.

Roger D. Chamberlain *is an associate professor in the Department of Computer Science and Engineering at Washington University in St. Louis. He received a DSc in computer science from Washington University. Contact him at roger@wustl.edu.*

Mark A. Franklin *is the Urbauer Professor of Engineering in the Department of Computer Science and Engineering at Washington University in St. Louis. He received a PhD in electrical engineering from Carnegie Mellon University. Contact him at jbf@wustl.edu.*

Eric J. Tyson *is a PhD candidate in the Department of Computer Science and Engineering at Washington University in St. Louis. He is currently employed as an ASIC design engineer at Nvidia. He received an MS in computer engineering from Washington University. Contact him at etyson@wustl.edu.*

James H. Buckley *is a professor in the Department of Physics at Washington University in St. Louis. He received a PhD in physics from the University of Chicago. Contact him at buckley@wustl.edu.*

Jeremy Buhler *is an associate professor in the Department of Computer Science and Engineering at Washington University in St. Louis. He received a PhD in computer science from the University of Washington-Seattle. Contact him at jbuhler@wustl.edu.*

Greg Galloway *is an MS candidate in the Department of Computer Science and Engineering at Washington University in St. Louis. He received a BS in electrical engineering from Washington University. Contact him at ggalloway@wustl.edu.*

Saurabh Gayen *is a chipset validation engineer at Intel. He received an MS in computer engineering from Washington University in St. Louis, where he contributed to this research. Contact him at saurabh.gayen@intel.com.*

Michael Hall *is a PhD candidate in the Department of Computer Science and Engineering at Washington University in St. Louis. He received an MS in electrical engineering from Southern Illinois University-Edwardsville. Contact him at mhall24@wustl.edu.*

E.F. Berkley Shands *is a senior research associate in the Department of Computer Science and Engineering at Washington University in St. Louis. He received an MS in computer science from Washington University. Contact him at berkley@wustl.edu.*

Naveen Singla *is a quantitative analyst at Exegy Inc. He received a DSc in electrical engineering from Washington University in St. Louis. Contact him at nsingla@exegy.com.*