A Federated Simulation Environment for Hybrid Systems

Saurabh Gayen
Eric J. Tyson
Mark A. Franklin
Roger D. Chamberlain

Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, Roger 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, pp. 198-207.

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

A Federated Simulation Environment for Hybrid Systems[1]

Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, and Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
{sg3,etyson,jbf,roger}@wustl.edu

Abstract

Hybrid computing systems consisting of multiple platform types (e.g., general purpose processors, FPGAs etc.) are increasingly being used to achieve higher performance and lower costs than can be obtained with homogeneous systems (e.g., processor clusters). Different platforms have different languages and simulators associated with them. Auto-Pipe has been developed as a toolset to reduce the complexity inherent in deploying an application to a diverse resource set. In Auto-Pipe, applications are expressed using the data flow coordination language X*, which describes the application in terms of interactions between functional blocks.*

As part of the Auto-Pipe system, X-Sim has been developed as a federated distributed simulator that can be used to conveniently and efficiently simulate applications. After a short introduction to Auto-Pipe and the X *Language, this paper considers issues involved with total system simulation of an application mapped to a hybrid resource set. The paper then demonstrates the use of X-Sim with a real-time signal processing application employed in the VERITAS gamma-ray astronomy project.*

1. Introduction

Hybrid (or heterogeneous) computing systems consisting of a multitude of different platform types (e.g., general-purpose processors, FPGAs, chip multiprocessors) are often the target for high-performance applications. This is generally motivated by a desire to leverage the unique strengths of each platform so that higher performance (or lower cost for the same performance) can be achieved. Developing applications to run on such a diverse set of platforms, however, is difficult.

Coordinating the multiple development languages and design environments associated with very different hardware platforms is awkward and error-prone. While simulation is an essential tool in the development methodology of some individual platforms (e.g., FPGAs), simulating a complete hybrid system is complicated by the need to coordinate disparate compilers and simulators, often with very different interfaces, options, and fidelities. In addition, implementing communication mechanisms to deliver data between different devices also increases development time.

Auto-Pipe [4] is a toolset developed to address these issues. Applications are expressed in Auto-Pipe through use of a data flow coordination language called *X*, which describes the interactions of functional blocks that comprise the application. The functional blocks themselves are expressed in the native language(s) associated with the hardware platforms on which they are to be deployed. For example, an FFT block to be deployed on an FPGA could be authored in a hardware description language (HDL), such as VHDL or Verilog, while a file I/O block to be deployed on a general-purpose processor could be authored in C. Data communication between blocks deployed on distinct platforms is automatically provided by the system, removing the need for the application developer to manually address this requirement.

X-Sim is the simulation component within Auto-Pipe. It uses platform-specific native simulation tools and direct execution capabilities to simulate the entire hybrid system. This helps in establishing functional correctness and gathering system-wide performance information. Key features of X-Sim are:

- Integration into the Auto-Pipe design flow.
- Integration of multiple, potentially very different simulators into a single federated simulation.
- Automation of the system simulation by coordinating individual simulator runs.
- Collection of performance data associated with resources and tasks.
- Fast simulation of parallel applications, with efficient

[1]This work is supported by NSF grant CCF-0427794.

re-simulations of moderately altered resource mappings.

Beyond simulation, the Auto-Pipe system also supports the deployment of the application onto a physical hybrid system for execution, and in the future will support automated application performance optimization.

In the remaining six sections, this paper describes the X-Sim federated simulation capability within the Auto-Pipe framework. Section 2 presents a brief introduction to the Auto-Pipe system and the X-Language. Section 3 then describes how X-Sim works, and how it fits into the Auto-Pipe system. In Section 4, an example hybrid system design problem and its *X* language representation are presented. The problem is taken from VERITAS [12], a high-energy gamma-ray astronomy experiment that involves the analysis of cosmic rays striking the Earth's atmosphere. Section 5 presents the results for application performance generated by using X-Sim, while Section 6 focuses on parallel performance of the simulation itself. The paper concludes in Section 7 with a discussion of the current status of the Auto-Pipe project and future work to be done.

2. Auto-Pipe Toolset

Auto-Pipe is a performance-oriented development environment for hybrid systems. It concentrates on applications that are represented as general data flow graphs and is especially useful in dealing with streaming applications placed on pipelined architectures. In Auto-Pipe, applications are expressed in the *X* language [11] as dataflow graphs. In these graphs, individual computational tasks called *blocks* are connected with interconnections called *edges* indicating the type and flow of data between blocks. An example application is illustrated in Figure 1. Here, blocks A through E have the indicated pipeline structure, enabling concurrent execution of blocks C and D.

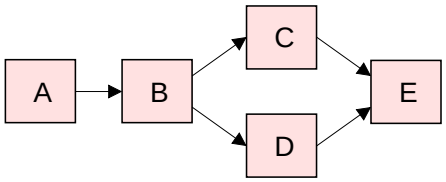

Figure 1. Sample application dataflow graph.

The actual *implementations* of the blocks are written in various languages for any subset of the available platforms (e.g., C for general-purpose processors, HDL for FPGAs, assembly for network processors and DSPs). Auto-Pipe provides an extensible infrastructure for supporting a wide variety of computation and interconnection devices, simulators, and native languages.

While it is possible for an application developer to both design the top-level application in *X* as well as implement the constituent blocks in the platform native languages, the separation of design responsibilities into top-level application development and block implementation also provides opportunities for block reuse. A well designed library of blocks, along with implementations of each block on more than one hardware platform type, supports both the ability to quickly develop applications (improving programmer productivity) and the ability to easily investigate alternative mappings of blocks to platforms.

The Auto-Pipe environment includes an *X* language compiler, called X-Com, and the X-Sim federated simulation environment. These components are the basis of the archetypical Auto-Pipe design flow depicted in Figure 2.

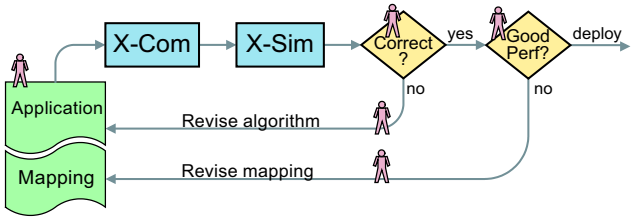

Figure 2. Design flow under Auto-Pipe.

In the Auto-Pipe design flow, X-Com performs compilation of the user-provided application code, supplemented with library code to perform execution profiling, inter-block connections, and high-performance inter-resource communications. X-Sim provides both functional simulation to determine application correctness and performance simulation to profile individual components of the application.

Simulation is an essential tool in the development process when using the types of computational platforms present in hybrid systems. For example, FPGA designs are regularly simulated prior to deployment onto the physical chip. By executing these simulations within the Auto-Pipe infrastructure, the individual block simulations are executing "in context," meaning their inputs are more readily representative of the actual data to be present once deployed. The simulation executions test a block's individual correctness as well as the correctness of its interactions with other blocks.

Once the designer is convinced of the functional correctness of the application, X-Sim also facilitates investigation of the performance of the system, both at the individual block and system-wide levels. Tools are provided for the user to calculate performance metrics like average execution time, average arrival rate, etc., for individual blocks from simulation traces. Work is currently underway to automate this analysis for the entire system.

In this context, the user is provided with performance data ranging from mean values to full distributions of tim-

ing parameters. The results of performance simulation may be used to investigate the implications of alternative mappings of blocks to computational resources, or alternatively to tune individual block implementations.

Currently, X-Com and X-Sim are operational and support a variety of computation platforms including native execution on general-purpose processors, basic simulation on such processors, hardware deployment on FPGAs, and simulation of HDL-composed hardware in ModelSim [9]. Processor resources support communication over shared memory or TCP/IP, FPGAs support communication over PCI-X bus, and all resources support a file-based simulation interconnect used by X-Sim.

3. The X-Sim Environment

Simulation is a critical precursor to deployment. This is even more so in the case when an application is being mapped to a hybrid resource set. Different resources have different simulators associated with them, so each resource's simulator must be run in the correct order. Data must be passed correctly between the simulators at the right time, taking into account simulated communication delays. Managing all this manually is a complex, error-prone task. X-Sim uses the application description provided by the user to the Auto-Pipe system to automate the simulation process.

A key feature of X-Sim is that it integrates multiple simulators into a federated simulation. The X-Sim infrastructure is open-ended to allow support for a range of simulators, from low-level discrete-event and cycle-accurate simulators to performance estimates from emulators and native execution. X-Sim supports the ModelSim [9] hardware logic simulator and native execution in POSIX environments. Support is currently being planned for processor simulators such as SimpleScalar [1].

X-Sim is a federated distributed simulator, similar to other projects like Distributed Interactive Simulation (DIS) [7], Aggregate Level Simulation Protocol (ALSP) [13], and the High Level Architecture [2, 3, 8], where federate simulators are treated as black boxes and few restrictions are applied on their internal operational mechanisms. Individual federate simulators need only conform to the interface specifications to become interoperable. Once the user specifies the application using the *X* language, no additional API or simulation requirements need to be satisfied for X-Sim to run.

X-Sim uses a system of standardized data and timestamp files across the entire application to coordinate multiple simulators. When data is produced by a simulation run, it is stored in a data trace file. Additionally, timestamps for when that data was produced are stored in a T_{out} file. The format for the timestamp file is stored in a file header and depends on the individual simulator. For example, a native machine simulation would store the system clock as the timestamps, and record the machine clock frequency in the header format section. A ModelSim simulation, on the other hand, would store simulation times as the timestamps, and the resolution (e.g. ns) in the file header. Timestamps are stored as binary 64 bit values. Each individual simulator records times in its native time format with X-Sim converting between the different formats.

There are two other types of timestamp files, T_{avl} files and T_{in} files. While T_{out} files are associated with output ports, T_{avl} and T_{in} files are associated with input ports. T_{avl} timestamps represent the time that data became *available* at an input port. X-Sim generates T_{avl} from T_{out} by simulating communication delay between hardware devices. T_{in} timestamps represent the time that data was *consumed* at an input port. The distinction between these files will become more apparent by considering the example shown in Figure 3.

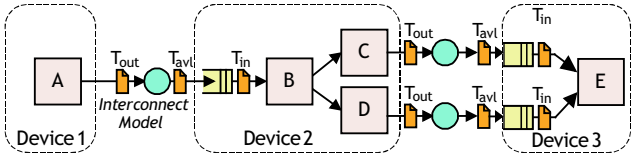

Figure 3. Flow of data in an X-Sim simulation.

In Figure 3, Block A is mapped to Device 1, Block E is mapped to Device 3, and the remaining blocks are mapped to Device 2. X-Sim first runs the simulator associated with Device 1. Note that the simulation for Device 1 is run in its entirety. This generates a data trace file containing the entire set of data produced, as well as an associated T_{out} file which records the time that each datum is produced. An implication of this practice is that X-Sim only supports feedforward data flow graphs, feedback is not allowed. Luckily, most parallel pipelined scientific applications that are the primary target for Auto-Pipe and X-Sim do not require feedback support.

Next, X-Sim simulates the transfer of data from Device 1 to Device 2 by using an interconnect model. It reads T_{out}, which represents the time data was produced at Block A's output port, applies the interconnect model to simulate communication delay, and writes T_{avl} which represents the time data is available at Block B's input port. The interconnect model can be a simple constant delay model, a probabilistic delay model, or even a data-dependent delay model. A constant delay model would simply add a constant delay to T_{out} values to generate T_{avl} values. A probabilistic delay model might sample from a given probability distribution to generate communication delays. Experimental runs on the actual communication links can be used to provide parameters for the probability distributions. Data-dependent delay

models can look at the data available in the trace files to determine transfer delay over the simulated communication link.

Once the T_{avl} file for Block B's input has been created, X-Sim is ready to run the simulation for Device 2. X-Sim makes a datum available to the Device 2 simulator only when the corresponding T_{avl} time has been reached. The Device 2 simulator runs the implementation including Blocks B, C and D until all the data available at Block B has been consumed. A T_{in} timestamp file is created which records the times that data was actually consumed by the simulator. For instance, a datum might have been available at time 5, but the simulator might have only been ready to start working on it at time 7. In that case, the corresponding values for that datum in the T_{avl} and T_{in} files are 5 and 7 respectively. As the Device 2 simulator processes the data, it creates data files and T_{out} timestamp files for Block C's output port as well as Block D's output port. It is important to note here that no data or timestamp files are generated for edges internal to Device 2. Data transfer between Block B and Blocks C and D is managed internally by the Device 2 simulator, and X-Sim does not create trace files for those edges[1].

Once again, communication models are run to generate T_{avl} files from T_{out} files. At this point, data and T_{avl} files are available for both of Block E's input ports. X-Sim then runs the simulation for Device 3. As before, data is only provided to the input ports when the T_{avl} timestamps allow it.

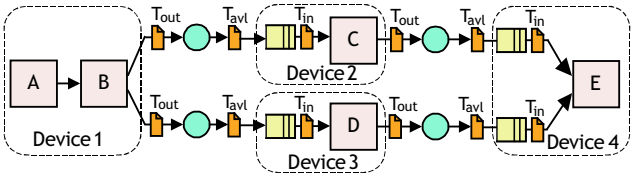

Figure 4. Parallel flow of data within an X-Sim simulation.

Let us now consider simulation of the alternative mapping presented in Figure 4. In this mapping, blocks A and B are mapped to Device 1, C and D are mapped to two separate devices 2 and 3, and block E is mapped to Device 4. X-Sim first runs the simulation for Device 1. T_{out} and data trace files are created for both the outputs of Block B, since the outgoing edges are not hidden within a device. At this point, the dataflow becomes parallel. Simulations for Devices 2 and 3 are both dependent only on the simulation for Device 1 and not on each other, so they can run concurrently. This feature is particularly useful in the case when lengthy hardware device simulations must be run and there are no direct dependencies between those simulations. In the Figure 4 mapping, T_{out} and data trace files can be transferred to two parallel machines that will run the simulations for Devices 2 and 3. Those machines will apply the communication models on the data, run the simulation for the corresponding device, and generate output trace files. Finally, X-Sim will run the simulation for Device 4, dependent on the times and data specified in the input trace files to Block E.

X-Sim analyzes the algorithm dataflow and resource mappings provided by the user to automatically determine the dependencies between individual simulation runs. For instance, in the previous example X-Sim is able to determine that simulations for Devices 2 and 3 are only dependent on the simulation for Device 1. X-Sim creates a Makefile that contains command line calls to run the individual simulations as well as calls to the communication model simulator. The Makefile also maintains the dependency relations between the simulations. The user simply needs to run "make sim" to run the entire federated simulation. X-Sim simulates the entire hybrid system, managing data transfer, timing standardization, communication, and individual device simulations.

In Figure 4, if all the individual simulations were to be run on the same machine, X-Sim would first run the Device 1 simulation, then run the simulations for Devices 2 and 3 in any order, and finally run the simulation for Device 4, all on the same machine. If multiple machines were available to run simulations, then X-Sim would first run the Device 1 simulation on one machine, then run the simulations for Devices 2 and 3 in *parallel* on two separate machines, and then finally run the Device 4 simulation on a single machine. The gains from parallel simulation execution will be explored more in Section 6.

X-Sim creates data files at each edge entering and exiting a device, which is very useful in checking total system correctness. Even if individual blocks have been checked for correctness, it is important to make sure that the entire system as a whole runs correctly. The time-stamped data trace files provide a way to debug inter-device connections. They can be examined to find at what time, and with what data element, any errors occurred. Individual device simulations can be run by themselves using these trace files to focus debugging efforts on the malfunctioning device.

After system correctness has been verified, the user can consider system performance analysis. The timestamp trace files are examined to generate timing results. Basic measurements from the trace files include the mean and variance of service time distributions associated with devices. These measurements can easily be aggregated to determine throughput and latency figures for the individual devices and the system as a whole. Analysis of this data can guide

[1]These traces can be obtained by mapping Blocks B, C, and D to separate devices and executing the simulation.

a user in creating a more advantageous mapping of blocks to devices. Currently the user must manually examine the system performance results and use them to generate a new mapping. Future plans include developing an X-Opt utility that can re-map the application automatically, analyze the trace files and repeat the whole process until satisfactory performance is achieved.

4. Example Application

The capabilities of X-Sim are illustrated in the following sections through the simulation and analysis of a scientific application: high-energy gamma-ray event parameterization.

There are thousands of scientifically interesting astronomical objects detectable from Earth which emit gamma-rays and other cosmic rays at very high energies. These sources may include pulsars, supernovae, neutron star collisions, and supermassive black holes. In order to detect and better understand these sources, astrophysicists use ground- and space-based gamma-ray detectors.

Unfortunately, detecting a wide range of gamma-rays directly is infeasible. Scientists in ground-based experiments can, however, make use of a byproduct of incident gamma-rays using the *imaging atmospheric Cherenkov technique*. This technique involves detecting faint showers of high-frequency visible light called Cherenkov radiation that are produced when high-energy gamma-rays strike the atmosphere. These showers strike the ground in specific patterns (such as hollow and filled ellipses) which indicate the origin of the shower and the type of particle which created it.

This technique is used by a variety of high-energy gamma-ray experiments [6, 10, 12]. In this paper, we focus on the VERITAS gamma-ray experiment, located in Arizona, USA, for which the gamma-ray parameterization application is the key algorithm that converts raw sensor data to meaningful scientific information.

The VERITAS experiment consists of four large (12 meter-wide) parabolic dishes covered in mirrors which reflect incoming light to an array of 500 photomultiplier tube sensors. These sensors are sampled by digital-to-analog converters at rates exceeding 500 MHz. Processing is required to turn the individual digitized sensor waveforms into a parameterized image. The processing involves two primary stages: signal processing to clean the analog signals received by the sensors, and image processing of the resulting cleaned image to detect patterns correlated to gamma-ray events.

Signal processing is performed on the input waveforms to deconvolve or "de-smear" the signal defects introduced by the analog electronics. This processing then compensates for various time delays and signal offsets and measures the resulting charge seen on each pixel.

The image processing finds a set of moments across the entire field of pixel charges. These moments are then input to a set of statistical equations which find parameters describing the most likely shape of the image seen on the pixel array. These parameters, a tremendous data reduction from the original waveform data, are then stored for later use in non-computationally-intensive statistical analysis to explore the nature of the original gamma-ray source.

Figure 5 depicts a simple hybrid architecture consisting of a cluster of dual general-purpose processor (GPP) computers, each connecting an FPGA to their system bus. Such a system can be readily assembled by inserting a PCI-capable FPGA development board (for which there are many dozens of options [5]) into a common off-the-shelf computer. Many companies (e.g. SGI, Cray) also sell and support similar systems and applications for those systems.

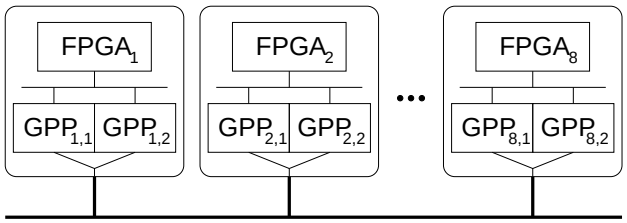

Figure 5. Hybrid architecture used in this example. Two general-purpose processors and one FPGA are available at each node.

This architecture is a good target for improving the performance of the gamma-ray event parameterization application beyond what could be achieved with a general-purpose processor solution alone. Performance improvement is contingent on the "good" mapping of application tasks to the computational components of the system, to take advantage of both task-level parallelism and the relative strengths of different platforms (e.g., FPGAs excel at many signal processing tasks).

Figure 6 demonstrates one potential reasonable mapping, hereafter called "Mapping A", of the gamma-ray application onto the above architecture. In Mapping A, the two processors in one system, $GPP_{1,1}$ and $GPP_{1,2}$, provide the data source ("Read Event") and sink ("Store Result"), respectively. Additionally, one signal processing pipeline is placed on each of four more processors ($GPP_{2,1}$ through $GPP_{5,1}$), each consuming pixel waveform data from the source and providing the resulting charge measurement to $GPP_{1,2}$.

FPGAs provide excellent performance for many signal processing applications due to the high degree of parallelism that they can support for very regular operations (such as vector operations and Fourier transforms). Figure 7 demonstrates "Mapping B", a mapping that introduces the

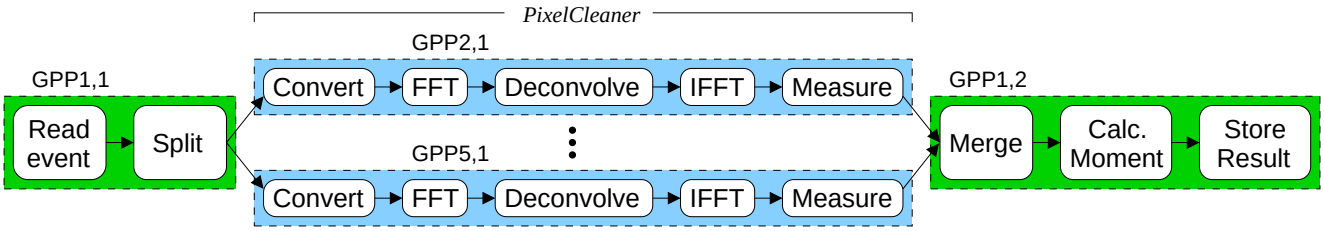

Figure 6. Mapping A, using four signal pipelines on four processors.

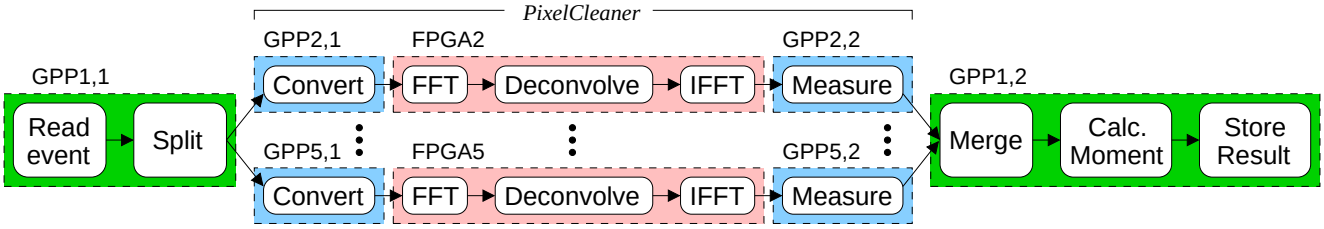

Figure 7. Mapping B, additionally using four FPGAs.

FPGA devices in each computer system to accelerate the signal processing pipelines. The FFT, IFFT, and Deconvolve operations of each pipeline are placed on the FPGAs. Additionally, the remaining operations (Convert and Measure) are divided onto different resources. This is done so that pipeline parallelism can be preserved across all three computational resources in the pipeline. $GPP_{1,1}$ and $GPP_{1,2}$ retain the same mapping as before.

An example of the signal-processing portion of the computation, as expressed in the *X* language, is provided in Figure 8. The Deconvolve is performed by an "AMultiply" block with appropriate coefficients, since a deconvolution is simply an element-wise multiplication in the frequency domain. The code demonstrates the instancing and configuration of library (e.g., FFT) and user-supplied (e.g., MeasureCharge) blocks within a compound block (PixelCleaner). The last two lines show the connection of the instanced blocks with the inputs and outputs to form a pipeline within the PixelCleaner block itself. PixelCleaner then becomes another block which can be similarly instanced in higher level blocks. This hierarchical approach is a valuable feature for high-level dataflow application design.

```
block PixelCleaner {
  input  InBytes VIN;
  output FLOAT64 VOUT;

  Convert c;
  FFT  ft(width=FFTSAMPLES);
  // Deconvolve with array-array multiply
  AMultiply dcv(coeffs=DECONV_XFORM);
  IFFT it(width=FFTSAMPLES);
  MeasureCharge mc;

  VIN -> c  -> ft -> dcv
      -> it -> mc -> VOUT;
};
```

Figure 8. *X* language code for the signal-processing subpipeline of the application.

5. Performance of Example Application

X-Sim generates comprehensive timestamp trace files from simulating a particular mapping of an application to a set of resources. These timing results can be analyzed to get an understanding of device performance, identify bottlenecks in the entire system, and to eventually generate a more efficient mapping. In this section, Mapping A and Mapping B from Section 4 are analyzed to illustrate the use of simulation results.

X-Sim generates a set of timestamps for every edge connecting two devices. In Mapping A, a set of timestamp files are thus generated for the edges between the Split and the Converts, and the edges between the Measures and the Merge. Execution times for devices $GPP_{2,1}$ through $GPP_{5,1}$ can be calculated by subtracting each device's T_{in} time from its T_{out} time. Execution times for the end devices ($GPP_{1,1}$ and $GPP_{1,2}$) are a bit trickier because currently there are no timestamp files for the very beginning and very end of the simulation. $GPP_{1,1}$ does not need to wait for any upstream block so the effective T_{avl} for all the data is zero.

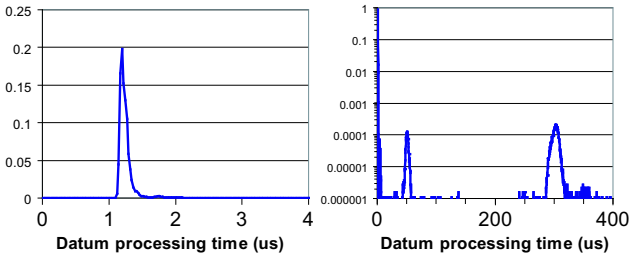

Figure 9. Histograms showing processing time of $GPP_{1,1}$. Note log scale on right graph.

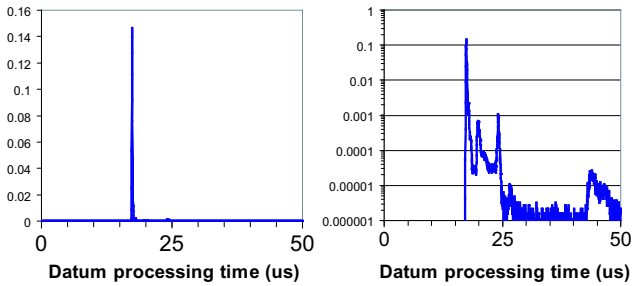

Figure 10. Histograms showing processing time of $GPP_{2,1}$. Note log scale on right graph.

The execution time for the device is then just the value of the last T_{out}. $GPP_{1,2}$ takes advantage of the profiler built into executables generated by X-Com. This feature keeps track of each block's cumulative execution time.

Mapping A was simulated using a dataset of 15,000 events. Mean and standard deviation of the execution time for the different devices are listed in Table 1. Only the mean is listed for $GPP_{1,2}$ because the built-in profiling utility only reports the mean. As can be seen from the data, the standard deviation is small across the board, indicating low variability in execution times.

Table 1. Performance statistics for Mapping A.

Resource	Mean (μsec)	Std. Dev. (μsec)
$GPP_{1,1}$	3.8	.04
$GPP_{2,1}$	17.7	.03
$GPP_{3,1}$	17.6	.02
$GPP_{4,1}$	17.7	.05
$GPP_{5,1}$	17.6	.04
$GPP_{1,2}$	9.25	—

Using the timestamps collected by X-Sim, we are able to construct histograms showing the distribution of execution times. Figure 9 and Figure 10 show histograms of execution times for $GPP_{1,1}$ and $GPP_{2,1}$ from simulating Mapping A. In Figure 9, note that there is a classic spike representing the bulk of execution times for $GPP_{1,1}$, with smaller bumps at higher times representing rarer longer execution times. Figure 10 shows the same thing for $GPP_{2,1}$. Comparing the two histograms, the bulk of executions for $GPP_{1,1}$ take about 1.2 μs and the bulk of executions for $GPP_{2,1}$ take about 17 μs.

The first diagram for $GPP_{1,1}$ shows how almost all the executions happen around the 1.2 μs mark. The second diagram is the first reproduced with a logarithmic y-axis and a longer x-axis. This is useful to see outlier behavior. Two additional spikes can be seen in this view, one occurring at the 50 μs mark and another occurring at the 300 μs mark. $GPP_{1,1}$ is performing disk access to read data files and so experiences cache-miss and memory-miss delays. These are possible explanations for the outliers.

The first diagram for $GPP_{2,1}$ shows the bulk of executions happening around the 17 μs mark. The second diagram is again a logarithmic y-axis version. We can see a couple of bumps around 20 μs and 25 μs that might represent context switches, and a much smaller (two orders of magnitude smaller) bump at about 45 μs. There were no extreme outliers found for $GPP_{2,1}$ so the x-axis was not expanded.

Mapping B differs from Mapping A in that the middle PixelCleaner stage is split across two processor cores and an FPGA. Table 2 shows the mean and standard deviation values for execution times across Mapping B.

Table 2. Performance statistics for Mapping B.

Resource	Mean (μsec)	Std. Dev. (μsec)
$GPP_{1,1}$	3.8	.17
$GPP_{2,1}$	.69	.006
$FPGA_2$	4.5	.005
$GPP_{2,2}$	.54	.095
$GPP_{3,1}$	.60	.006
$FPGA_3$	4.5	.005
$GPP_{3,2}$	.56	.055
$GPP_{4,1}$	.59	.006
$FPGA_4$	4.6	.004
$GPP_{4,2}$	.52	.026
$GPP_{5,1}$	.60	.006
$FPGA_5$	4.5	.005
$GPP_{5,2}$	.55	.031
$GPP_{1,2}$	9.25	—

For Mapping B, simulation was done on only 50 events. This was due to practical considerations of the length of simulation runs when using hardware simulators. This much smaller sample of events is still representative of hybrid performance because FPGA hardware does not have the same issues with OS overhead and disk access that a software implementation does. $\text{GPP}_{1,1}$ is mapped to software same as in Mapping A, and shows the same average performance. However, it shows a higher variability and standard deviation in this experiment because the number of events is much lower.

The end-to-end latency and overall throughput for the mappings can be determined directly using empirical results from the simulation. However, an alternative approach is taken here, where latency and throughput values are calculated by characterizing the execution times of the constituent blocks using their means and analytically composing the overall performance figures. This general analytic approach is more suitable to automatic re-mapping and optimization within the larger Auto-Pipe framework.

Let $d_{i,j}$ represent the execution time delay for $\text{GPP}_{i,j}$, calculated by subtracting the T_{in} from the T_{out} for $\text{GPP}_{i,j}$ as mentioned before. Let c_1 represent the communication time between two processors, taken to be a constant 5 μs delay in this example, a typical latency value for cluster communication. Then, the equations for latency and throughput for Mapping A are given below.

$$\begin{aligned} \text{latency} &= \text{sum of sequential stage delays} \\ &= d_{1,1} + c_1 + \max_{2 \le i \le 5} d_{i,1} + c_1 + d_{1,2} \\ \text{throughput} &= 1/\text{maximum individual stage delay} \\ &= 1/\max(d_{1,1}, d_{2,1}, d_{3,1}, d_{4,1}, d_{5,1}, d_{1,2}, c_1) \end{aligned}$$

Applying these equations to the simulation trace data, we get that Mapping A has a latency of 40.75 μs and a throughput of 56,500 events/s. For Mapping B, the equations for latency and throughput are given below, with f_i representing the execution time delay for FPGA_i, and c_2 representing a constant delay of 1 μs delay, derived from data transfer size, PCI-X bandwidth, and processor-FPGA communication overhead.

$$\begin{aligned} \text{latency} &= \text{sum of sequential stage delays} \\ &= d_{1,1} + c_1 \\ &\quad + \max_{2 \le i \le 5}(d_{i,1} + c_2 + f_i + c_2 + d_{i,2}) \\ &\quad + c_1 + d_{1,2} \\ \text{throughput} &= 1/\text{maximum individual stage delay} \\ &= 1/\max(d_{1,1}, c_1, \max_{2 \le i \le 5}(d_{i,1}, f_i, d_{i,2}), d_{1,2}) \end{aligned}$$

These show that Mapping B has a latency of 30.78 μs and a throughput of 108,100 events/s. This represents a decrease in latency by 25% and an increase in throughput by a factor of two. Clearly, the use of FPGAs is beneficial for this application. For all these calculations, a constant delay communication model has been used. A probabilistic model could be used to generate more realistic communication delays.

This section has shown a basic analysis of Mappings A and B using the timestamps collected by X-Sim. Utilities to process the time stamp files to generate model parameters and histograms are provided and are useful to help the user understand system performance.

6. Parallel Simulation Performance

As indicated earlier, Auto-Pipe and the *X* language permit specification and simulation of dataflow described systems. A key feature of such systems is the representation of parallel computational structures and their subsequent mapping onto multiple resources that may execute their tasks in parallel. This provides opportunities for exploiting parallelism and thus obtaining improved performance in the deployed hybrid system. It can also be used in improving simulation performance during the system correctness and performance analysis phase of a project. Thus, if parallel resource structures are present in the resource mapped system (e.g., Figures 6 and 7) then straightforward parallel simulation techniques can be used to reduce simulation times

Consider, for example, Figure 7 where there are both serial and parallel structures and two types of resources: FPGAs and GPPs (assume that all GPPs are identical). Representing the simulation time of each resource execution as $T.Resource$, a standard sequential simulation of Mapping B would take time $T.Sim.MapB.seq$ as shown below.

$$\begin{aligned} T.Sim.Map.B.seq &= T.GPP_{1,1} \\ &\quad + \sum_{i=2}^{5}(T.GPP_{i,1} + T.FPGA_i + T.GPP_{i,2}) \\ &\quad + T.GPP_{1,2} \end{aligned}$$

If the elements of the PixelCleaner pipeline are simulated in parallel then the overall simulation time becomes:

$$\begin{aligned} T.Sim.Map.B.par &= T.GPP_{1,1} \\ &\quad + \max_{2 \le i \le 5}(T.GPP_{i,1} + T.FPGA_i + T.GPP_{i,2}) \\ &\quad + T.GPP_{1,2} \end{aligned}$$

In this second expression, a max term has replaced the earlier summation term and thus, in general, the parallel

simulation time will be less, and often substantially less, than the sequential simulation time.

Similar expressions can be developed for any dataflow graph where there is no feedback. In this example, for Mapping B, the X-Sim uses ModelSim to simulate the VHDL implementation description and this represents a simulation performance bottleneck. By providing facilities (e.g., standard compute cluster) for parallel simulation of the Pixel-Cleaner pipelines, simulation times may be considerably reduced. To demonstrate these ideas, a series of experiments were performed with resource mappings A and B and with one, two or four processors available for simulation. The mappings are labeled MapA or MapB, and the number of simulation processors are labeled as Sim.1, Sim.2, or Sim.4. The individual experiments, including the assignments of simulated resources to simulation processors, are listed below:

- MapA.Sim.1 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$ to $GPP_{5,1}$, $GPP_{1,2}$

- MapA.Sim.2 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$, $GPP_{3,1}$, $GPP_{1,2}$

  Sim.Proc.2 $\leftarrow$ $GPP_{4,1}$, $GPP_{5,1}$

- MapA.Sim.4 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$, $GPP_{1,2}$

  Sim.Proc.2 $\leftarrow$ $GPP_{3,1}$

  Sim.Proc.3 $\leftarrow$ $GPP_{4,1}$

  Sim.Proc.4 $\leftarrow$ $GPP_{5,1}$

- MapB.Sim.1 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$ to $GPP_{5,1}$,
  $FPGA_2$ to $FPGA_5$,
  $GPP_{2,2}$ to $GPP_{5,2}$, $GPP_{1,2}$

- MapB.Sim.2 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$, $GPP_{3.1}$, $FPGA_2$,
  $FPGA_3$, $GPP_{2,2}$, $GPP_{3,2}$, $GPP_{1.2}$

  Sim.Proc.2 $\leftarrow$ $GPP_{4,1}$, $GPP_{5,1}$, $FPGA_4$, $FPGA_5$,
  $GPP_{4,2}$, $GPP_{5,2}$

- MapB.Sim.4 Experiment

  Sim.Proc.1 $\leftarrow$ $GPP_{1,1}$, $GPP_{2,1}$, $FPGA_2$, $GPP_{2,2}$,
  $GPP_{1,2}$

  Sim.Proc.2 $\leftarrow$ $GPP_{3,1}$, $FPGA_3$, $GPP_{3,2}$

  Sim.Proc.3 $\leftarrow$ $GPP_{4,1}$, $FPGA_4$, $GPP_{4,2}$

  Sim.Proc.4 $\leftarrow$ $GPP_{5,1}$, $FPGA_5$, $GPP_{5,2}$

Table 3. Parallel simulation performance.

Experiment	No. of Proc.	Simulation Time (sec)	Speedup
MapA.Sim.1	1	204	1.0
MapA.Sim.2	2	156	1.3
MapA.Sim.4	4	115	1.8
MapB.Sim.1	1	8,495	1.0
MapB.Sim.2	2	4,344	1.9
MapB.Sim.4	4	2,193	3.9

The results of the experiments are shown in Table 3.

Note that the times listed for Mapping B include hardware simulations and so are much higher across the board than the times for the pure software Mapping A simulations.

The speedups associated with simulation performance are distinctly different for the two resource mappings. In Mapping A, the tasks are mapped to processors and no FPGAs are involved. Due to this, the relative simulation complexity of the stages and the times associated with each of the pipeline stages are similar. Additionally the number of parallel pipelines is small. Thus, since only the middle stage supports concurrent execution, the amount of simulation speedup that can be achieved through parallelism is limited. This is shown in the speedup results of 1.3 and 1.8 with 2 and 4 processors respectively.

In Mapping B, where FPGAs are utilized, the time associated with FPGA simulation constitutes the major simulation performance bottleneck and parallelizing this simulation component yields almost a linear reduction in simulation time. Thus, with two processors the speedup is almost 2, and with four processors the speedup is nearly 4.

For complex hybrid systems that are to be simulated on processor clusters, determining the simulation assignments for efficient parallel simulation is an interesting problem in its own right. Our future work includes developing methods to automate this process.

7. Conclusions

This paper has introduced X-Sim, a federated simulation environment for the Auto-Pipe toolset, has described its capabilities, and has illustrated its operation on a hybrid systems design problem. Such hybrid systems employ multiple technologies to achieve high performance and are frequently used in embedded pipelined computer designs. X-Sim supports the infrastructure of Auto-Pipe and enables the simulation of applications distributed onto complex hybrid architectures. Such simulations are necessary in verifying correctness and in optimizing system design to achieve performance goals. The paper considers a particular pipeline that is employed in processing gamma ray signals

and illustrates the sort of simulation-derived, time-stamped data and associated distributions that can be obtained for each system device. Additionally, the design can be simulated in a parallel fashion, significantly improving simulation performance in many situations. For example, FPGA simulation is a bottleneck for the gamma ray pipeline application. In this case, our experiments demonstrated a near linear speedup of the simulation when multiple processors were employed to simulate parallel FPGAs.

X-Sim currently supports the ModelSim simulator for detailed hardware simulation. It also supports native execution for performance estimates of off-the-shelf computer systems. Furthermore, X-Sim is also easily extended to support more simulator and emulator environments and performance models as more device types and development platforms are integrated into the Auto-Pipe toolset. Current short term goals are to improve the ease of use of the Auto-Pipe system and to develop a set of processing modules that will eventually form a computational module library. Such a library will have implementations of various processing functions both in standard languages such as C and C++, and also in hardware description languages. This will permit designers to more easily examine design tradeoffs between the use of GPPs and FPGAs within a proposed implementation. Additionally, as they become available, detailed cycle accurate GPP simulators will be integrated into the set of simulation targets.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[2] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The Department of Defense High Level Architecture. In *Proceedings of the 1997 Winter Simulation Conference*, pages 142–149, 1997.

[3] J. S. Dahmann, F. Kuhl, and R. Weatherly. Standards for simulation: As simple as possible but not simpler the high level architecture for simulation. *SIMULATION*, 71(6):378–387, 1998.

[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] P. Freiden. FPGA boards and systems at FPGA-FAQ.ORG. `www.fpga-faq.com/FPGA_Boards.shtml`.

[6] W. Hofmann. Status of high energy stereoscopic sys. (HESS) project. In *27th Int'l Cosmic Ray Conf.*, 2001.

[7] IEEE. *IEEE 1278.1-1995, Standard for Distributed Interactive Simulation – Application Protocols*. New York, NY, 1995.

[8] F. Kuhl, R. Weatherly, and J. S. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.

[9] Mentor Graphics Corp. ModelSim. `www.model.com`.

[10] A. Moralejo. The MAGIC telescope for gamma-ray astronomy above 30 GeV. *Memorie delle Societa Astronomica Italiana*, 75:232, 2004.

[11] E. Tyson. X language specification draft. Technical Report WUCSE-2005-47, Dept. of Computer Science and Engineering, Washington University in St. Louis, 2005.

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

[13] A. L. Wilson and R. Weatherly. The aggregate level simulation protocol: An evolving system. In *Proceedings of the 1994 Winter Simulation Conference*, pages 781–787, 1994.