WASHINGTON UNIVERSITY

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

X-SIM AND X-EVAL:

TOOLS FOR SIMULATION AND ANALYSIS OF HETEROGENEOUS PIPELINED ARCHITECTURES

by

Saurabh Gayen

Prepared under the direction of Professor Mark A. Franklin

A thesis presented to the School of Engineering and Applied Science
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

May 2008

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

X-SIM AND X-EVAL:

TOOLS FOR SIMULATION AND ANALYSIS OF HETEROGENEOUS PIPELINED ARCHITECTURES

by

Saurabh Gayen

ADVISOR: Professor Mark A. Franklin

May 2008

Saint Louis, Missouri

Heterogeneous 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 both the performance analysis and in the deployment of 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 simulation system that can be used to conveniently and efficiently simulate applications. X-Eval has been developed as a post simulation analysis tool, as part of an effort to optimize the performance of the application.

This thesis presents an overview of the Auto-Pipe system, descriptions of X-Sim and X-Eval, and sample applications to illustrate the Auto-Pipe development cycle with an emphasis on the simulation and analysis aspects.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank members of the Storage Based Supercomputing (SBS) group for providing an excellent framework within which to develop the Auto-Pipe toolset. In particular, I would like to thank my advisor, Mark Franklin, and Roger Chamberlain for their guidance and help in developing X-Sim and X-Eval, and in writing this thesis. A special mention also goes out to Eric Tyson, with whom I've spent countless hours designing, discussing, and developing the Auto-Pipe toolset.

I would also like to express my gratitude to my friends and family for their support and encouragement during the pursuit of my degree.

Finally, thanks to the National Science Foundation, who through grant CCF-0427794, have funded the research done in the SBS group.

Saurabh Gayen

Washington University in Saint Louis
May 2008

Chapter 1

Introduction

This thesis presents X-Sim and X-Eval, tools for simulation and analysis of applications within the Auto-Pipe [9, 20] framework. Auto-Pipe is a development environment that enables high performance applications, particularly streaming applications, to be developed for a diverse set of target computational platforms and connection topologies.

Within this environment, X-Sim provides a mechanism to simulate the entire application, both for correctness checking as well as for performance profiling. It produces a series of traces that record events that happened during the simulation. X-Eval serves as a post-simulation analysis tool that uses these traces to calculate performance measurements and create performance models of system elements. A tool planned for the future, tentatively named X-Opt, will use results and models generated by X-Sim and X-Eval to create more optimal mappings of the application to the target processing architecture.

This introductory chapter describes the background of high performance streaming problems and how Auto-Pipe was developed to aid the development process. The motivation for simulation and performance analysis within this framework is investigated, and the goals for X-Sim and X-Eval within Auto-Pipe are described. Related work to both X-Sim and X-Eval is discussed and compared. Finally, the outline for the rest of the thesis is presented.

1.1 Background

Heterogeneous (or hybrid) processing architectures consisting of a multitude of different platform types (e.g., general-purpose processors, FPGAs, chip multiprocessors) are often the target for high-performance applications[1]. 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.

Target applications that we consider are predominantly streaming scientific computation applications. These applications typically involve a large amount of data flowing sequentially through a pipeline of computational stages, with each stage performing an incremental computation. For example, an application currently being developed using the Auto-Pipe system is VERITAS [22], the Very Energetic Radiation Imaging Telescope Array System. VERITAS is an astrophysics project where data gathered from an array of telescopes is streamed through a processing architecture to analyze it. The data is generated by sensing Cherenkov radiation produced by gamma rays striking the atmosphere.

Streaming scientific applications like VERITAS can be represented using an acyclic dataflow graph. Figure 1.1 shows a dataflow graph for a simple example streaming application (not VERITAS) that serves as an illustrative example.

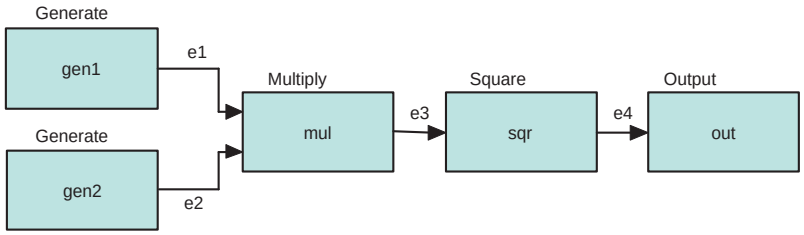


Figure 1.1: Sample dataflow graph

The algorithm represented by this dataflow graph consists of five *tasks*, shown in the graph as *blocks*. A *task* may represent any computational process, from as simple as an addition to as arbitrarily complex as required. A *block* is the representation of a computational task in a dataflow graph. Blocks

[1]Background text is similar to text in [10].

are connected to each other by *edges*. *Edges* are dataflow graph representations of communication channels between tasks. They can be thought of as infinite FIFOs. The two blocks on the left, `gen1` and `gen2`, represent *data sources* because they have no incoming edges. *Data sources* are blocks that produce data without getting an input from other blocks in the dataflow graph. Sample data generation methods include reading a file or database, reading a sensor, or using a random number generator. Data from the two generating blocks are sent over edges `e1` and `e2` to the `mul` (multiply) block. After being processed by the `mul` block, data is streamed through the *sqr* (square root) and the *out* blocks. The *out* block represents a *data sink*, a block which has one or more input ports but no output ports. Results can be stored by the sink block in data files or a database, or simply be output to screen. The presented dataflow graph is acyclic because there is no path by which data can re-enter a block it has already left. Many high performance scientific computations can either be represented by such an acyclic dataflow graph, or their data intensive computation part can be reduced to such a representation.

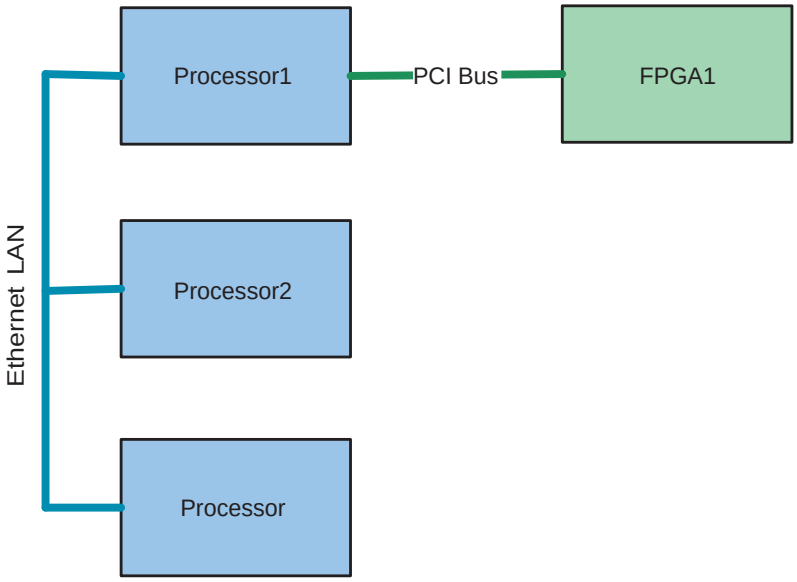


Figure 1.2: Sample processing architecture

Let us now consider a sample processing architecture that the given example application can be deployed to. Figure 1.2 shows three processors connected to each other using an Ethernet LAN. The first processor also has an FPGA (Field Programmable Gate Array) linked to it using a PCI bus.

A general processing architecture consists of multiple *computational resources* (CRs) connected together. A CR is a device that can execute tasks given by an application description. Examples of CRs include different types of general purpose processors (GPPs), field programmable gate arrays (FPGAs), digital signal processors (DSPs), network processors (NPs), graphical processing units (GPUs), as well as others. Support for GPPs and FPGAs is already built into Auto-Pipe. The Auto-Pipe system is extensible so that in the future other CRs can also be supported. In the processing architecture given in Figure 1.2, the instantiated CRs are three Intel x86 processors and one Virtex 2 FPGA.

The CRs in a processing architecture are connected using *interconnect resources* (IRs). An IR is a communication mechanism between two CRs. For example, in Figure 1.2, the Ethernet LAN connecting the processors, and the PCI bus connecting the first processor to an FPGA are examples of IRs.

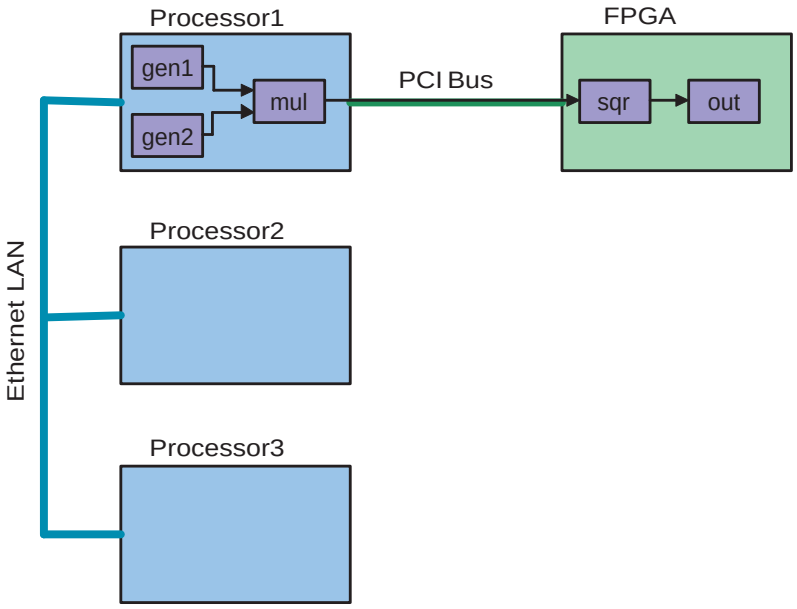


Figure 1.3: Sample mapping

Mapping an algorithm to an architecture is the process of assigning each block to a CR, and each application edge to an IR. Consider the problem of mapping the algorithm in Figure 1.1 to the architecture in Figure 1.2. A possible mapping is presented in Figure 1.3.

In this mapping, the two `gen` blocks and the `mul` block are mapped to one processor. Data from the `mul` block passes over the PCI bus to the FPGA, where the `sqr` and `out` blocks are mapped. With this mapping, Auto-Pipe generates a single executable to be deployed on `Processor1`, and a single bit file to be deployed on the FPGA.

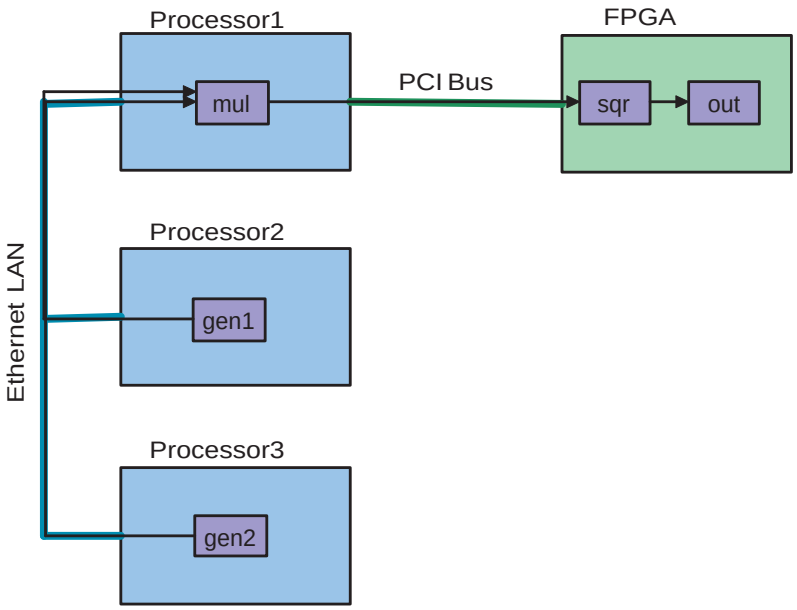


Figure 1.4: Alternate sample mapping

An alternate mapping is shown in Figure 1.4. In this mapping, the two `gen` blocks have been moved to other processors. This new mapping represents a more equal distribution of blocks to CRs, and exploits computational parallelism present in the algorithm. If the communication delay over the Ethernet IR is not too high, this mapping may have higher performance than that of Figure 1.3.

1.2 Motivation

Coordinating the multiple development languages and design environments associated with very different computational resources is awkward and error-prone[2]. Implementing and evaluating different mappings to optimize the performance of applications is a time consuming, complex task.

[2]Motivation text is similar to text in [10]

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, including X-Sim and X-Eval, were 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 might 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.
- Accelerating simulations by taking advantage of various simulation speedup techniques (see Chapter 5)

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.

1.3 X-Sim and X-Eval in the Auto-Pipe Design Flow

Auto-Pipe is a performance-oriented development environment for hybrid systems. Components of Auto-Pipe include an X language compiler called X-Com, the X-Sim federated simulation system, and the X-Eval analysis tool. These components form the standard design flow shown in Figure 1.5.

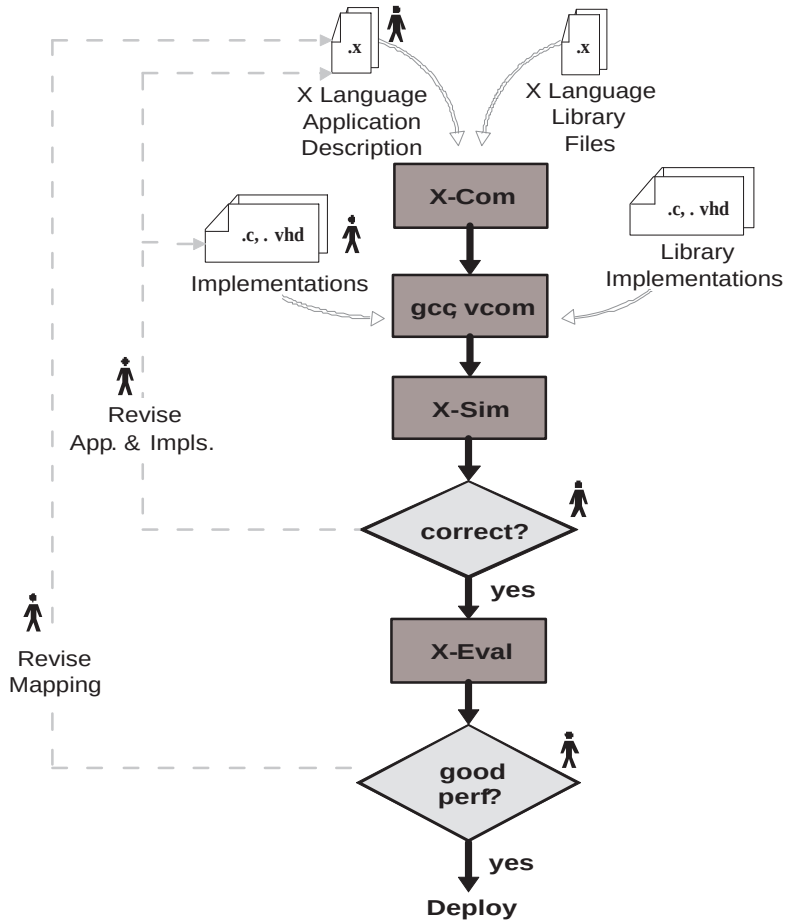


Figure 1.5: Design flow under Auto-Pipe

The human icon represents steps that are done manually, and boxes represent automated tools provided as part of Auto-Pipe for application development. Diamonds represent a choice that needs to be made, with yes and no answers leading to different next steps.

The design flow given in the figure starts on the top, with X Language files created by a user. In these files, the user describes the application algorithm, the target processing architecture, and an initial mapping of the algorithm to the architecture. These X Language files, along with various X libraries to manage communication and profiling, are compiled by X-Com to generate platform-specific source C and VHDL files. In turn, these X-Com generated files are combined with user-created C and VHDL implementation files, and compiled by platform-specific compilers (`gcc` and `vcom`) to create deployable executables for each CR.

In the development of any heterogeneous application, simulation is an essential tool. For example, FPGA designs are regularly simulated prior to deployment onto the physical chip. In the Auto-Pipe infrastructure, X-Sim simulates the entire application, automatically taking care of coordination between multiple simulators that are used to simulate different target CRs.

X-Sim produces a set of *traces* that provide a full history of all resource level events. These traces are recorded at edges mapped to IRs (Interconnect Resources), and at the start and end of processing of source and sink blocks. For example, Figure 1.6 shows where traces are recorded in the application introduced earlier.

Data trace files, represented by ‘D’ files in the figure, store the data that was transferred over a corresponding edge. Timestamp trace files, represented by ‘T’ files in the figure, store the exact times at which the transfers occurred. Both data and timestamp files are recorded for each edge in the dataflow graph. The data files can be used to do a functional check of the simulation, in addition to aiding debugging efforts. The timestamp files can be used to generate a history of execution times for each block that has both input edges as well as output edges. By executing simulations within the Auto-Pipe infrastructure, X-Sim runs individual block simulations in context, meaning their inputs are more readily representative of the actual data to be present on the deployed system. The simulation executions test a block’s individual correctness as well as the correctness of its interactions with other blocks. To allow timing analysis to be done for source and sink blocks, timestamps are

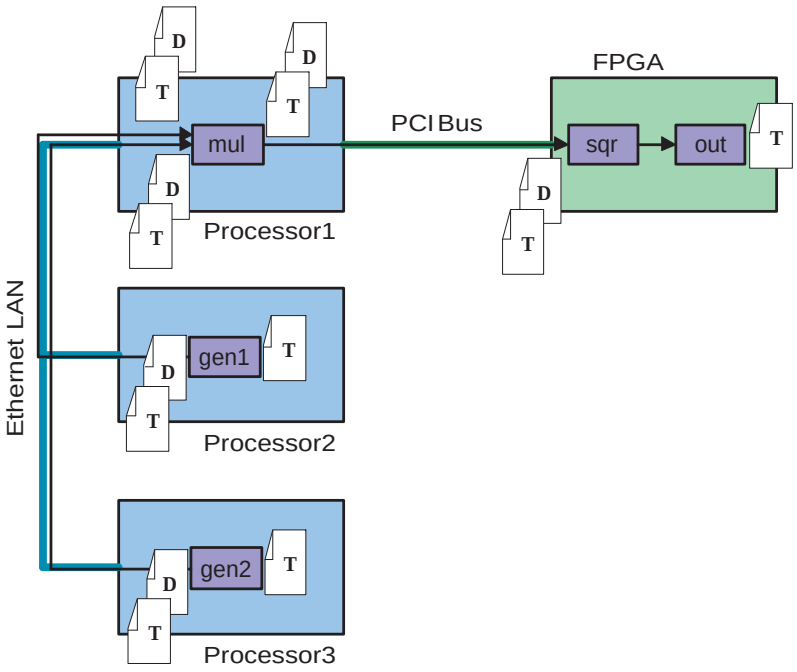


Figure 1.6: Traces recorded for sample mapping

also stored to record when source blocks *started* executing, and when sink blocks *ended* executing. Additional details of X-Sim's mechanism, including traces, are provided in Chapter 3.

Output from execution of the application simulation may be examined by the user to determine if the whole application executes correctly. If errors occur, the user can use data traces collected from the simulation run to narrow down problem areas and focus debugging efforts on the malfunctioning blocks. Once correctness has been established, the user can use X-Eval to analyze the performance of the application. For each block in the dataflow graph, X-Eval reads the input and output timestamp trace files. By subtracting the input times from the output times, execution times for each block can be calculated. How the analysis for different types of individual blocks is done, and how performance analysis is done for the application as a whole is described in more detail in Chapter 4.

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. A proposed tool for this has tentatively been named X-Opt. The augmented Auto-Pipe design flow including this new tool is shown in Figure 1.7.

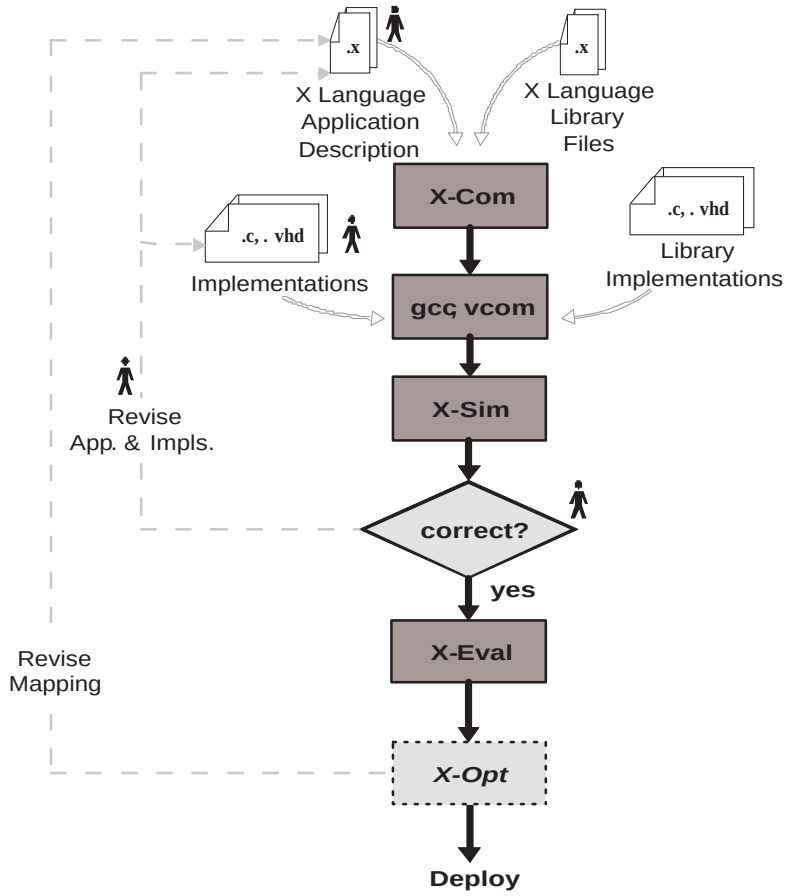


Figure 1.7: Augmented design flow under Auto-Pipe

Note that the difference in this design flow from the previous flow is that manual optimization has been replaced by X-Opt, an automated performance optimization tool. It is planned that X-Opt will use results from X-Eval to analyze the performance of individual blocks within the application, and will be able to determine more optimal mappings of blocks to resources. Additional details on X-Opt can be found in Section 2.6.

1.4 Related Work

X-Sim is a federated simulation system designed to model streaming applications mapped to a hybrid processing architecture. We will consider work related to X-Sim in three fields:

- federated simulation systems
- hybrid architecture targeted application development systems
- streaming application development systems

A *federated* simulation system is one in which multiple simulators, called *federate* simulators, run to simulate a single system. These federate simulators can be in different simulation languages, and even be of very different fidelities. Such systems are called *heterogeneous* federated simulation systems, and X-Sim is an example of such a system.

A popular standard for parallel federated simulation is the High Level Architecture (HLA) [5, 6, 14] developed by the Defense Modeling and Simulation Office [8]. The HLA is a general purpose architecture for coordinating a distributed set of simulators spread across a variety of computing platforms. Figure 1.8 shows the general structure of a HLA compliant federated simulation system.

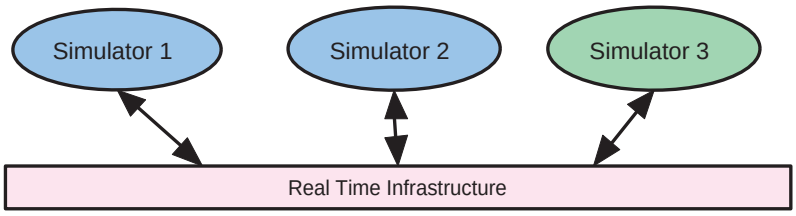


Figure 1.8: An HLA simulation system

Autonomous, heterogeneous federate simulations run in parallel with each other. These heterogeneous simulators need to be able to communicate with each other, but do not share a common native communication interface. HLA enforces an Interface Specification that federate Simulators must adhere to. By conforming to this Interface Specification, federate simulators can interact, through an underlying Run Time Infrastructure (RTI) [3] simulation backplane, with each other. The RTI is responsible for:

- construction and destruction of shared data objects
- time synchronization of simulation operation between federates
- coordinating access to shared data objects

X-Sim is similar to the HLA/RTI distributed federated simulation architecture. A target processing architecture includes many different CRs that need to be simulated. If the architecture is a heterogeneous mix, different types of simulators are needed to simulate the different components. For example, a native execution model can be used to simulate an application component mapped to a general purpose processor, while a ModelSim [15] simulation can be used to simulate a different component mapped to an FPGA. The equivalent to the RTI for X-Sim is a system of timestamp and data trace files.

The trace files produced by blocks mapped to all resources have a uniform format. This allows simulators for different CRs to be able to seamlessly communicate with each other. Auto-Pipe is an extensible system, designed to allow more CRs to be supported in the future. X-Sim is similarly extensible. A developer adding support for another CR type to Auto-Pipe must also add the trace generation code for the simulator associated with the new CR to X-Sim. This will allow the new simulator to communicate correctly with all the other federate simulators.

Let us now consider work related to X-Sim from a different perspective, looking at application development systems targeted towards hybrid, or heterogeneous, processing architectures.

The first system considered is Ptolemy II. Ptolemy II is “a set of Java packages supporting heterogeneous, concurrent modeling and design” [7]. It is a part of UC Berkeley’s Ptolemy project [18], which studies modeling, simulation, and design of concurrent computational systems, focusing on assembly of heterogeneous components. Ptolemy II has many of the same goals as X-Sim, particularly the analysis of heterogeneous processing architectures. However, there are also some key differences between the two systems. Ptolemy II is focused on the modeling of heterogeneous systems, with little attention given to actual deployment on a real processing architecture. In Ptolemy II, modeling is approached from a top-down *analytic* perspective. The whole application is first designed by representing different tasks (or blocks) by analytic models. More detailed functional implementations for the blocks are written only later. In contrast, X-Sim uses a bottom-up performance modeling approach where implementations are fully developed first before being profiled to create analytic models. The main reason for this difference is the different targets for the two design tools. The Ptolemy II system is targeted towards embedded systems, where *latency* results are

often crucial to the operation of an application. By designing a system from the beginning with required timing models, latency requirements can be satisfied. In contrast, X-Sim is targeted towards streaming applications, usually with large data processing requirements, where *throughput* is the most critical performance metric. Latency requirements may be considered later using performance modeling and revised mapping facilities. This allows ease of implementation and deployment to be given a higher priority, appropriate for a system like Auto-Pipe where ease of use for end-user scientists and engineers is an important goal.

LabView [12] is a popular proprietary heterogeneous application development environment from National Instruments [17]. The main feature of LabView that distinguishes it from other similar systems is that LabView has a graphical language "G" that is used to describe the application. G allows users to create graphical representations of tasks and link them by drawing wires. This makes the system very popular with scientists and engineers who do not have much background in conventional programming. Rudimentary functional simulation is supported in LabView, but there is no extensive mechanism to profile the timing performance of blocks.

We now shift the focus to streaming application development. GLU (Granular LUcid) [13] is an application development system for granular data-parallel programming, including for streaming applications, targeted towards general purpose computers. It uses a high level programming language called Lucid [2] to express implicitly parallel relationships between sequential functions (or tasks) that are implemented in C. Lucid does not simply structurally represent the edges between blocks like X does. Instead, it uses functional relations to express data dependencies between user-defined C functions. GLU is able to express functionally more complex relationships between blocks, but requires significantly more user effort in porting over an existing application to the GLU framework. Functional level debugging is provided by GLU, but only at the level of traces that verify the data dependencies between blocks. In terms of a dataflow graph, these traces can be used to check whether the data being produced on an edge by an upstream block matches the data being consumed by a downstream block. In Auto-Pipe, this level of functional checking is inherently provided by the X-Com compiler.

In addition to functional debugging, GLU also provides a mechanism for performance profiling of the application. Compiling GLU source code creates one or more *generator* executables, with each generator executable having one or more *worker* executables assigned to it. Generator executables are in charge of interpreting the dataflow relationships between blocks and running worker executables at the right time with the right data. Worker executables include the actual implementations of the dataflow graph blocks. GLU provides performance metrics for each generator and worker executable, including values for total runtime, total implementation execution time, and idle time. In addition to performance analysis, GLU is capable of dynamic load management as well as static load distributions, and of providing performance metrics for each generator and worker executable. X-Sim and X-Eval combine to provide a profiling mechanism for applications written under X, while X-Opt will provide the ability to determine an optimal static load distribution over the different available CRs. In GLU's case, the CRs are all GPPs, each capable of running any block because each block is a C implementation. This lets worker loads be dynamically distributed among CRs. In Auto-Pipe, the CRs are more heterogeneous, so work cannot easily be dynamically reallocated. The emphasis in Auto-Pipe is thus on determining the optimal static allocation of work to CRs (i.e. mapping the application to the heterogeneous architecture).

StreamIt [19] and StreamC [1] are two other high level languages that have been developed to facilitate the development of streaming applications on multi-core processors. StreamIt is platform independent but has primarily been built to support the RAW processor [21] developed at MIT. It has a legal Java syntax, and is used to represent the relationship between tasks, called 'filters' in this system. The simulator for the system is btl [16], a cycle-accurate simulator specific to the RAW processor. StreamC [1] was developed in conjunction with the Stanford Imagine processor, and is specifically aimed to this target. Two simulators are available for the Imagine processor. Idebug is a functional simulator that uses the Visual Studio debugging environment and is used to debug Imagine applications during development. ISim, on the other hand, is a cycle-accurate simulator for the Imagine Processor. It is used for performance profiling, and like Idebug, is a platform specific simulator.

Being a federated simulation language for streaming application, all the simulators mentioned for other streaming languages are appropriate targets for being run as federate simulators within X-Sim.

For example, a possible application might have an MIT RAW processor stream data to an Imagine processor. X-Sim can provide the means to run a federated simulation where simulators for the two different processors are run at the right times and with the right data to simulate running of the complete heterogeneous application.

1.5 Overview of Thesis

This thesis presents X-Sim and X-Eval, the simulation and analysis parts of the Auto-Pipe design environment. Chapter 2 presents the broader Auto-Pipe system, describing the different tools available to the developer. Chapter 3 presents X-Sim, describing its underlying theory as well as execution mechanism. Chapter 4 looks at X-Eval, describing how resultant traces from X-Sim runs are analyzed to understand the performance of an application. In Chapter 5, we look at various techniques that are used to speed up simulation runs. Chapter 6 looks at some sample applications to analyze both application performance as well as simulation running times. Finally, Chapter 7 summarizes the current state of X-Sim and X-Eval, and describes the future work planned for these tools.

Chapter 2

The Auto-Pipe System

The goal of this thesis is to present X-Sim and X-Eval. However, to get a good understanding of these simulation and analysis tools, it is important to understand the broader Auto-Pipe system and see how these two tools fit into the bigger picture.

The goal of the Auto-Pipe system is to take a description of an algorithm and a target architecture and to produce a deployable system that has high performance. This chapter will take a sample dataflow and processing architecture and use it to describe the different components of Auto-Pipe.

2.1 X: A Language for Describing Target Applications and Architectures

The X-Language forms the basis for the whole Auto-Pipe system. It is a coordination language that is used to describe the dataflow of an algorithm, the targeted processing architecture, and the mapping of the algorithm blocks to processing resources.

Consider the example `test1` algorithm whose dataflow is presented in Figure 2.1. Two data source blocks `gen1` and `gen2`, are shown on the left side of the dataflow. These are instances of the `GENERATE` block, a type of block that performs the function of generating numbers and sending them to an output port. At the dataflow interface level, each `GENERATE` block has a single output `y0`

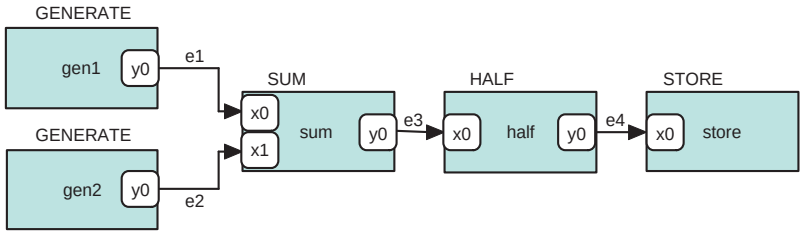


Figure 2.1: Algorithm dataflow for example `test1`

which is of type `FLOAT32`. Figure 2.2 shows the interface and the X description of the `GENERATE` block.

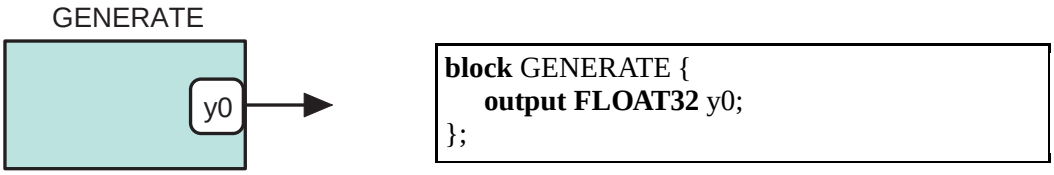


Figure 2.2: X Language code for the `GENERATE` block

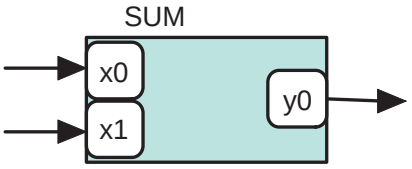
Blocks are the basic components that build up an application algorithm. They represent a single task within the algorithm. A block is named by a user-chosen identifier, in this case `GENERATE`. A block specification also has a list of input and output *ports* that indicate the types of data input and output from the block. In the case of the `GENERATE` block, there is only a single output port `y0` of type `FLOAT32`. This means that data produced by a `GENERATE` block on its `y0` port is of type `FLOAT32`.

X supports multiple types of data, with each of them conforming to the corresponding IEEE standard. The main types are:

- UNSIGNED8, UNSIGNED16, UNSIGNED32, UNSIGNED64
- SIGNED8, SIGNED16, SIGNED32, SIGNED64
- FLOAT32, FLOAT64, FLOAT128
- STRING

The X-Level description of the block only gives a structural description of the block. For example, for the GENERATE example, we know that the block produces a FLOAT32 on its single output port y0. The X code does not specify how that FLOAT32 is produced, or what it means semantically. The actual functioning of the block is given by its *implementation* source code that is written in a platform specific language such as C or VHDL. For the GENERATE block, the implementation code determines how data is produced on the y0 port. Implementation files will be considered in the next section, Section 2.2.

Consider another block, the SUM block, shown in Figure 2.3.




```
block SUM {
    input FLOAT32 x0, x1;
    output FLOAT32 y0;
};
```

Figure 2.3: X Language code for the SUM block

This block has two inputs x0 and x1, and a single output y0. In the case of the SUM block, the implementation determines how data that is consumed on the two input ports is processed to produce output on the single output port.

In this way, all the blocks in the dataflow for an algorithm are defined using the X Language. In addition to block definitions, X also provides a mechanism to instantiate blocks and connect them to each other to form a dataflow graph.

Figure 2.4 shows the X Language description of the entire algorithm shown in Figure 2.1. Block definitions of HALF and STORE follow the same pattern as that described for GENERATE and SUM. The top block does not have any inputs or outputs, and represents the whole application. Within top, both the blocks required for the algorithm and their dataflow interconnections are specified.

First, all the blocks in the application are instantiated. For example, line 23 creates two instances of the GENERATE block.

```
23      GENERATE gen1, gen2;
```

```
1  block GENERATE { //generate a number
2      output FLOAT32 y0;
3  };
4
5  block SUM { //add two numbers
6      input FLOAT32 x0, x1;
7      output FLOAT32 y0;
8  };
9
10 block HALF { //divide a number by 2
11     input FLOAT32 x0;
12     output FLOAT32 y0;
13 };
14
15 block STORE { //save the results
16     input FLOAT32 x0;
17 };
18
19 block top { //top level dataflow description
20     //no inputs or outputs for top block
21
22     //block instances:
23     GENERATE gen1, gen2;
24     SUM sum;
25     HALF half;
26     STORE store;
27
28     //edges
29     e1: gen1.y0 -> sum.x0;
30     e2: gen2.y0 -> sum.x1;
31     e3: sum.y0 -> half.x0;
32     e4: half.y0 -> store.x0;
33 };
34
35 use top;
```

Figure 2.4: `test1_algo.x`: Sample algorithm description

Once all the blocks for the dataflow have been instantiated, dataflow connections between the ports of the blocks are made. This is done by placing an arrow `->` between a block's output port and another block's input port. For example, line 29 makes a connection (labeled `e1`) between `gen1`'s output port `y0` and `sum`'s input port `x0`.

```
29      e1: gen1.y0 —> sum.x0;
```

Let us now consider a target architecture on which the algorithm is to be deployed, shown in Figure 2.5, consisting of three processors connected to each other via Ethernet LAN. Two of the processors are additionally linked to an FPGA over a PCI bus.

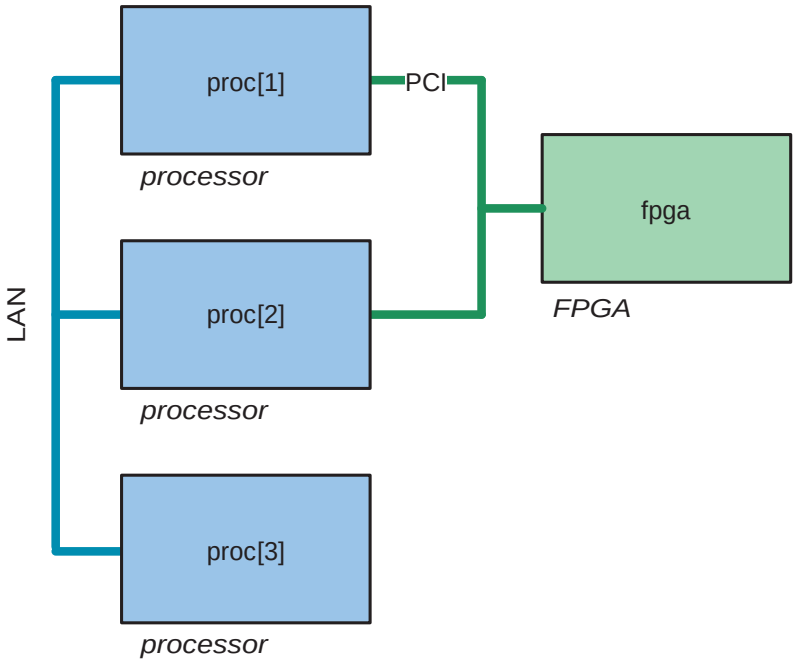


Figure 2.5: Processing architecture for example `test1`

Each separate node in a processing architecture is called a *computational resource (CR)*. In this example, there are four CRs, three processors and one FPGA. Each CR is an instance of a *platform*. Platforms are classes that represent the languages that can be used to implement an X block on a real world device. A CR instantiates a platform, giving definite values to platform configuration options such as whether MMX is supported (for Pentium platforms), which part number is being used (for an FPGA card) etc. Figure 2.6 shows the platform declarations for the given example.

```
1  //library code defines base HDL and C platforms
2  #include "std.x"
3
4  //---FPGA platform definition---
5  platform HDL;
6  platform HDL_Virtex4:HDL {
7      config STRING part; //FPGA part name
8  };
9
10 //---GPP platform definition---
11 platform C;
12 platform C_x86:C{
13     config UNSIGNED16 freq=1600; //default frequency in MHz
14 };
15 platform C_Pentium4:C_x86{
16     config STRING hasMMX = "yes";
17 };
```

Figure 2.6: X Language platform declarations for example `test1`

Base platforms such as `HDL` and `C` are defined inside the X library file `std.x`. Platforms can be extended by other platforms, as shown in Figure 2.6 line 6 where `HDL_Virtex4` is declared to be a subclass of `HDL`. Platforms can have *configs* defined inside their bodies, which give configuration options for the particular platform. In the case of `HDL_Virtex4`, the `part` config lets the X compiler X-Com know which part to use when the implementation is synthesized.

If no default value is given for a config, as is the case for the `part` config, the user must provide a value when instantiating the platform. Contrast this with the `hasMMX` config where the default value `"yes"` is given. The user does not need to provide a value for `hasMMX` when instantiating a `C_Pentium4`. The default value `"yes"` will automatically be assigned to the config.

Linktypes are classes used to define the interconnections between CRs. Linktypes are instantiated by Interconnect Resources (IRs), similar to how platforms are instantiated by Computational Resources (IRs). Figure 2.7 shows how linktypes are declared for the `test1` example. Similar to platforms, linktypes too can be organized in hierarchies. In this example, `bus_pci` is a subclass of the `bus` linktype.

```
1  //library code defines base bus and switch linktypes
2  #include "std.x"
3
4  //---PCI bus declaration---
5  linktype bus;
6  linktype bus_pci:bus{
7      config UNSIGNED16 freq=66, width=32;
8  };
9
10 //---Ethernet LAN declaration---
11 linktype switch;
12 linktype switch_ether:switch{
13     config bandwidth=1000; //bandwidth in Mbps
14 };
```

Figure 2.7: X Language linktype declarations for example `test1`

Once all the required platforms and linktypes are declared, they must be instantiated using CRs (for platforms) and IRs (for linktypes). Additionally, the topology of the processing architecture must also be described, as shown in Figure 2.8.

```
1  //---CR declarations---
2  resource fpga is HDL_Virtex4( part="XC4V8000");
3  resource proc[3] is C_Pentium4{
4      //these frequencies override the default of 1600MHz
5      (freq=3200), (freq=3200), (freq=2800)
6  };
7
8  //---IR declarations--
9  resource LAN is switch_ether(
10     { proc[1], proc[2], proc[3] }
11     //default bandwidth=1000 used
12 );
13 resource PCI is bus_pci{
14     { proc[1], proc[2], fpga}
15 };
```

Figure 2.8: X Language topology description for example `test1`

On line 2, `fpga` is declared as a CR instance of the `HDL_Virtex4` platform, with the config option part set to the name of the particular FPGA device. Line 3 shows the declaration of a whole array of CRs. Three processors are declared here, all of platform type `C_Pentium4`, and their config option frequencies are given in a list.

The IRs for the example are also presented in this code. This is where the topology of the target hardware architecture is described. Line 10 of the X code declares the IR `LAN` to be an instance of the platform `switch_ether`, and assigns the three `procs` declared earlier to be nodes on this IR.

```
10 resource LAN is switch_ether(
11     { proc[1], proc[2], proc[3] }
12     //default bandwidth=1000 config for LAN used
13 );
```

Similarly, line 14 declares `proc[1]`, `proc[2]`, and `fpga` to be nodes on the `PCI` IR, which in turn is an instance of the `bus_pci` linktype.

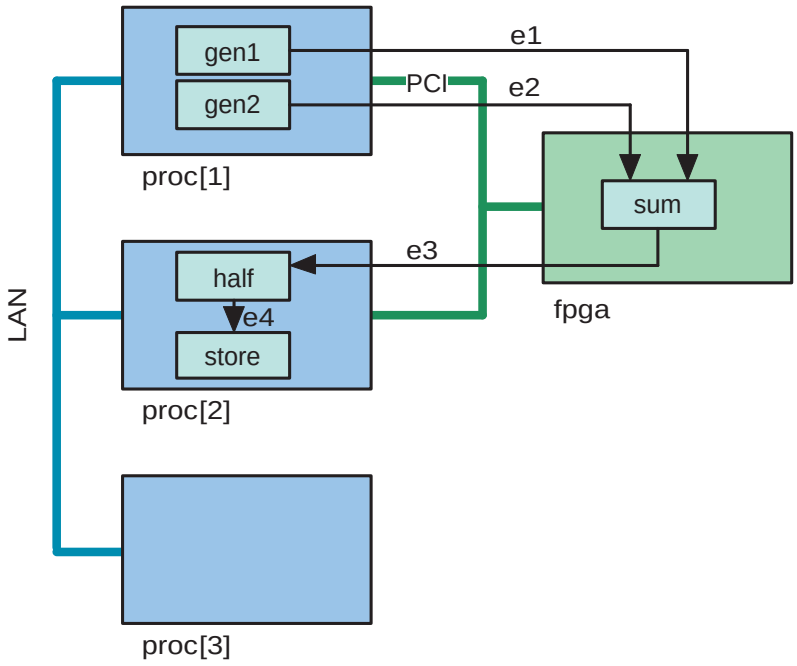
```
14 resource PCI is bus_pci{
15     { proc[1], proc[2], fpga}
16     //default freq=66 and width=32 configs for bus_pci used
17 };
```

So far, we have looked at X language descriptions of the algorithm dataflow, as well as the description of the targeted processing architecture. Now let us look at how the X language is used to describe mappings of dataflow blocks onto the processing architecture for the example `test1`. Consider the mapping shown in Figure 2.9.

In this mapping, the two `GENERATE` blocks, `gen1` and `gen2`, are mapped to `proc[1]`. Data from these blocks is transferred over PCI to the `fpga` CR where it is summed together, and then transferred back over PCI to `proc[2]`.

The mapping of blocks to CRs and edges to IRs is done simply by using the X Language keyword `map` as shown in the code snippet in Figure 2.9. Note that line 5, which maps dataflow edges to the `PCI` IR, could be omitted from the code and the X-Com compiler would be able to infer that mapping. This is because there is only one possible IR, `PCI`, that that can transfer data from `proc[1]` to `fpga`, or from `fpga` to `proc[2]`.

In this section we showed how the X language is used to describe the algorithm for an application, the target processing architecture, and the mapping of blocks and edges to CRs and IRs. We will now consider X-Com, the Auto-Pipe tool that compiles X language code to executables.




```
1 map { top.gen1, top.gen2 } to proc[1];
2 map { top.sum } to fpga;
3 map { top.half, top.store } to proc[2];
4
5 map { top.e1, top.e2, top.e3 } to PCI;
```

Figure 2.9: Mapping for example `test1`

2.2 X-Com: Compiling Heterogeneous Applications

X-Com is the compiler developed in conjunction with the X Language. Figure 2.10, modified from a figure in [20], shows the design flow of applications developed using X-Com.

X-Com takes in a list of X Language files that define the application. The command line call to run the X-Com executable `xcom` is given below:

```
xcom source1.x source2.x source3.x ...
```

These input files are parsed by X-Com in order (e.g. `source1.x` followed by `source2.x` followed by `source3.x`, and so on). Collectively, these X Language source code files represent

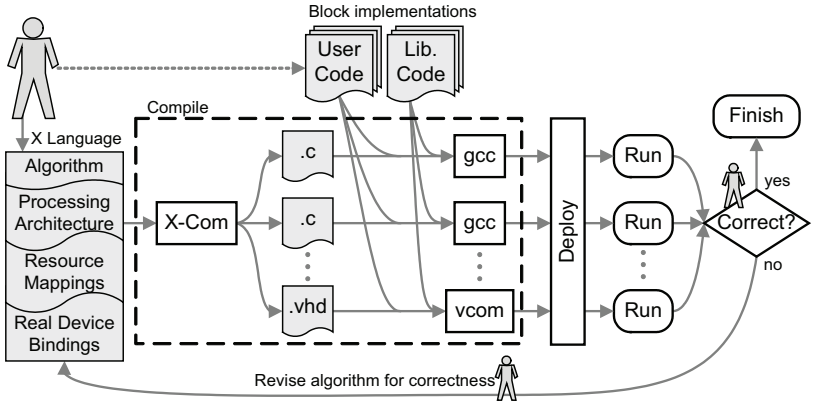


Figure 2.10: X-Com design flow

the application algorithm, processing architecture, mapping, and architecture device bindings. X-Com applies the C pre-processor to each source file before parsing, which allows an X Language file to include another X Language file using the `#include` directive. This allows for a cleaner file organization, and allows the user to avoid writing a long flat file for the entire application.

Another big benefit is that X Language files can be organized to be modular, allowing different architectures and mappings to be switched out simply by including a different file. Consider the example X application that was shown in Figure 2.9. Separate X Language files can be used to describe the algorithm, the architecture, and the mapping. `test1_algo.x` (Figure 2.4), `test1_arch.x` (Figure 2.11), and `test1_mapping.x` (Figure 2.12) give the X Language descriptions of these parts of the application.

In `test1_algo.x` (Figure 2.4), the block types used in `test1` are defined in lines 1–17. The top level block `top` is then defined in lines 18–33 to describe the structure of the algorithm dataflow.

`test1_arch.x` (Figure 2.11) describes the processing architecture for the `test1` example. Line 1 of the file is a `#include` statement that includes `std.x`, an X Language library file. This file contains definitions for the standard CRs `HDL` and `C`, and IRs `bus` and `switch`. These base resources are then extended to define resources specific to this X application, `HDL_Virtex4` on line 5, `C_x86` on line 11, `C_Pentium4` on line 14, `bus_pci` on line 20, and `switch_ether` on line 26. Finally, the topology of the architecture is given at the bottom of the `test1_arch.x` file.

```
1  #include "std.x"
2
3  //---FPGA platform definition---
4  platform HDL;
5  platform HDL_Virtex4:HDL {
6      config STRING part; //FPGA part name
7  };
8
9  //---GPP platform definition---
10 platform C;
11 platform C_x86:C{
12     config UNSIGNED16 freq; //frequency in MHz
13 };
14 platform C_Pentium4:C_x86{
15     config STRING hasMMX = "yes";
16 };
17
18 //---PCI bus declaration---
19 linktype bus;
20 linktype bus_pci:bus{
21     config UNSIGNED16 freq=66, width=32;
22 };
23
24 //---Ethernet LAN declaration---
25 linktype switch;
26 linktype switch_ether:switch{
27     config bandwidth 1000; //bandwidth in Mbps
28 };
29
30 //---CR declarations---
31 resource fpga is HDL_Virtex4( part="FXC4V8000");
32 resource proc[4] is C_Pentium4{
33    (freq=3400), (freq=3400), (freq=2800)
34 };
35
36 //---IR declarations--
37 resource LAN is switch_ether(
38     { proc[1], proc[2], proc[3]}
39 );
40
41 resource PCI is bus_pci{
42     { proc[1], proc[2], fpga}
43 };
```

Figure 2.11: `test1_arch.x`: Sample architecture description

Now that the application algorithm and processing architecture have been described in two X Language files, the mapping of the algorithm blocks to architecture resources can be done in a third file, `test1_map.x` (Figure 2.12).

```
1 #include test1_algo.x
2 #include test1_arch.x
3
4 map { top.gen1, top.gen2 } to proc[1];
5 map { top.sum } to fpga;
6 map { top.half, top.store } to proc[2];
```

Figure 2.12: `test1_map.x`: Initial sample mapping description

Note that this file includes the two previous X Language files, `test1_algo.x` and `test1_arch.x`, on Lines 1 and 2. If an alternate mapping was required with the same algorithm and architecture, the only file that would need to be changed would be the mapping file. Consider the alternate mapping shown in Figure 2.13.

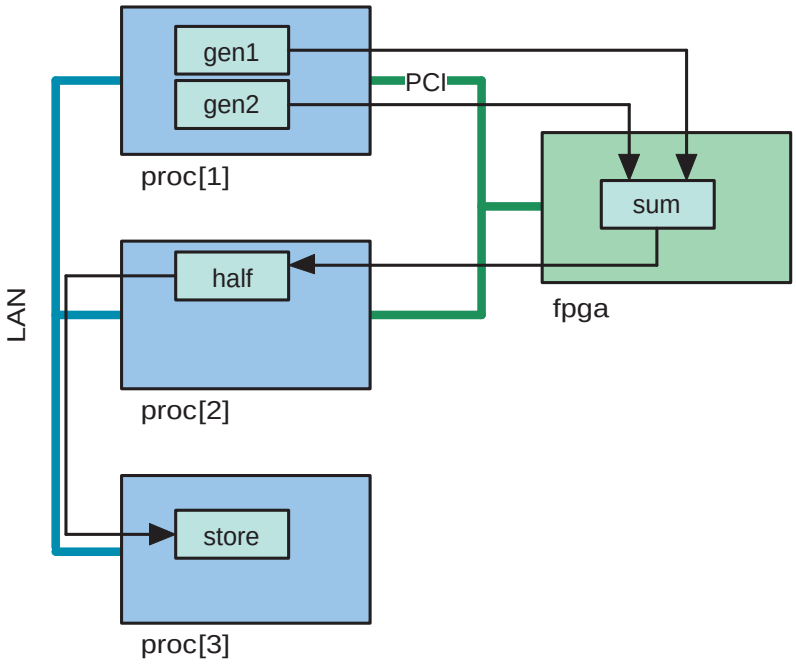


Figure 2.13: An alternate mapping for example `test1`

In this mapping, the `store` block has been moved from `proc[2]` to `proc[3]` to more evenly distribute the workload. The only change required to implement this new mapping would be to substitute file `test1_map.x` by the file `test1_map2.x` given in Figure 2.14.

```
1 #include test1_algo.x
2 #include test1_arch.x
3
4 map { top.gen1, top.gen2 } to proc[1];
5 map { top.sum } to fpga;
6 map { top.half } to proc[2];
7 map { top.store } to proc[3];
```

Figure 2.14: `test1_map2.x`: Alternate sample mapping description

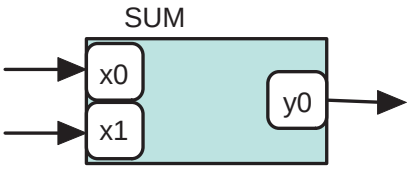
Note that edge mappings have been excluded in both the given mapping X Language files. In all cases, there was only a single possible IR option for edges to be mapped to. X-Com is able to figure out such implicit mappings of edges.

Going back to the X-Com design flow diagram (Figure 2.10) at the beginning of this section, let us consider the block implementation files shown at the top. These implementation files provide the functionality for the X Language specified blocks. Currently there are two major implementations for blocks: C implementations and VHDL implementations. We shall consider the APIs for both these languages.

2.2.1 C API and implementations for X blocks

To illustrate the implementation of a block implemented in C, let us take the `SUM` block from the `test1` example. The block level diagram for the `SUM` block and its corresponding X Language description are repeated in Figure 2.15.

From this description we know that the `SUM` block takes in two `FLOAT32`s, and outputs one `FLOAT32`. The C implementation for this block must match this block description. Figure 2.16 gives the generalized API for ALL X blocks, while Figure 2.17 gives the implementation skeleton for the `SUM` block.




```
block SUM {
    input FLOAT32 x0, x1;
    output FLOAT32 y0;
};
```

Figure 2.15: X Language code for the SUM block (same as Figure 2.3)

The C API for ALL X blocks consists of one data structure and four functions. It is important to note here that the data structure and four functions are only declared once for every type of block. In our `test1` example, there are two `GENERATE` block instances `gen1` and `gen2`. To signify these blocks, two instances of the data structure `X_GENERATE_data` are created. These data structures are then passed into the four functions shown before to "call the function" on `gen1` or `gen2`. In the case of `SUM`, there is only a single instance (called `sum`) in our example.

The data structure (Figure 2.17), in this case `X_SUM_data` for the `SUM` block, contains a clock to internally keep an aggregate of total time spent in running a block. This can be used to collect basic performance measurements. Its use is replaced in this thesis by X-Sim trace collection and X-Eval analysis which provide more detailed performance measurements, as will be seen in later chapters. Also contained in the data structure are pointers to the ports of the block, and a bit portmask that informs the block of whether data is available at each of its input ports. Finally, the data structure for an X block also contains pointers to a `send` function and a `release` function. These functions are generated by X-Com during compilation, and are used internally to send data downstream and to release upstream data respectively. Use of the `send` and `release` functions is illustrated later when the `X_SUM_push` function is considered.

Each of the implementation functions is now considered in turn. The first such function included in the C API for an X block is the `init` function. This function is called at the beginning of an X application run.

```
29  void X_SUM_init(struct X_SUM_data *d) {}
```

As shown in this code snippet, no initialization work is required for a `SUM` block. Other blocks may require some initialization work. For example, the initialization code for a `GENERATE` block might

```
1  //library file contains definitions for Xclock_t and portmask_t
2  #include "X.h"
3
4  struct X_<blockname>_data {
5      // internal clock for performance measurement
6      Xclock_t clock;
7
8      // pointer to send function for sending data to an output port
9      void (*send)(int);
10
11     // pointer to release function for releasing data on an input port
12     void (*release)(int,char);
13
14     // bitmask (array of chars) with info about data on input ports
15     // each bit corresponds to an input port
16     // 00000010 means x1 has data and x0 does not
17     // 1 char for up to 8 ports, so e.g. use portmask[3] for 18 input ports
18     portmask_t portmask[<number of port/8>];
19
20     //one line per port
21     <datatype> *<port>;
22
23     //add any internal state data that needs to be kept track of
24     //for e.g. GENERATE might have a counter variable stored here
25 };
26
27 //--- implementation functions for the X block ---//
28 // initialization function called on startup
29 void X_<blockname>_init(struct X_<blockname>_data *) { ... }
30
31 // destructor function called on termination
32 void X_<blockname>_destroy(struct X_<blockname>_data *) { ... }
33
34 // push function called when data is received on a port
35 void X_<blockname>_push(struct X_<blockname>_data *) { ... }
36
37 // go function for blocks that are sources of data
38 int X_<blockname>_go(struct X_<blockname>_data *) { ... }
```

Figure 2.16: C API for *all* X blocks

```
1  //library file contains definitions for Xclock_t and portmask_t
2  #include "std.x"
3
4  struct X_SUM_data {
5      // internal clock for performance measurement
6      Xclock_t clock;
7
8      // pointer to send function for sending data to an output port
9      void (*send)(int);
10
11     // pointer to release function for releasing data on an input port
12     void (*release)(int,char);
13
14     // bitmask (array of chars) with info about data on input ports
15     // each bit corresponds to an input port
16     // 00000010 means x1 has data and x0 does not
17     portmask_t portmask[1];
18
19     //list of ports
20     FLOAT32 *iport0;  // input port x0
21     FLOAT32 *iport1;  // input port x1
22     FLOAT32 *oport0;  // output port y0
23
24     //no internal variables required for SUM block
25 };
26
27 //--- implementation functions for SUM block ---//
28 // initialization function called on startup
29 void X_SUM_init(struct X_SUM_data *) {}
30
31 // destructor function called on termination
32 void X_SUM_destroy(struct X_SUM_data *) {}
33
34 // push function called when data is received on a port
35 void X_SUM_push(struct X_SUM_data *) {
36 ...
54
55 }
56
57 // go function for blocks that are sources of data
58 int X_SUM_go(struct X_SUM_data *) {
59     return 1;
60 }
```

Figure 2.17: C implementation for the `SUM` block

seed the random number generator and initialize the count variable to 3, as shown in the following code. The count variable keeps track of how many numbers still need to be generated.

```
1  void X_GENERATE_init(struct X_GENERATE_data *d) {
2      srand(time(NULL));
3      count = 3;
4  }
```

Note that for the implementation of a `GENERATE` block to work correctly, `stdlib.h` and `time.h` would need to be included for the C `srand()` and `time()` functions respectively. Since there are two `GENERATE` blocks, the data structure pointers for both `gen1` and `gen2` would be passed into the global function `X_GENERATE_init`.

In addition to the initialization function, a `go` function (line 58 of Figure 2.17) is also included in the C API, mainly as a way for data sources to create data and send it into the rest of the application. When a C binary is created and run for a processor, it calls the `go` functions of all the X blocks on that processor in a round robin schedule. For example, if `gen1`, `gen2`, and `sum` were all mapped to `proc[1]`, the binary for that processor could call the `go` functions for the blocks in the following order: `gen1`, `gen2`, `sum`, `gen1`, `gen2`, `sum`, `gen1` etc. However, each block is kept on the round robin schedule only until its `go` function returns a 1, at which point it is excluded from the schedule. A `SUM` block returns a 1 the first time its `go` function is called, as shown below.

```
58 int X_SUM_go(struct X_SUM_data *d) {
59     return 1;
60 }
```

On the other hand, a `GENERATE` block would keep generating a random number and sending it downstream `count` times, as shown in the code in Figure 2.18. After generating the last random number, a `GENERATE` block would return a 1 to signify that it should not be called anymore. The `count` variable was set to 3 in `X_GENERATE_init`, so `X_GENERATE_go` will generate three random numbers. The `go` function for a `GENERATE` block is shown in Figure 2.18.

In this case, the order of calling the `go` functions in the schedule would be: `gen1`, `gen2`, `sum`, `gen1`, `gen2`, `gen1`, `gen2` etc. The schedule would call `sum` once, and `gen1` and `gen2` three times each.

```
1  int X_GENERATE_go(struct X_GENERATE_data *d) {
2      if(d->count <= 0){ //check if more numbers need to be generated
3          return 1;
4      }
5
6      //allocate memory
7      d->oport0 = gmalloc(sizeof(FLOAT32));
8
9      //generate random number on output port y0
10     *(d->oport0) = 10*rand();
11
12     //send the data on oport0
13     send(0);
14
15     //update the count
16     d->count--;
17
18     if(d->count == 0){ //check if this was the last number
19         return 1;
20     }
21
22     //return 0 to signify that this go function should still be called
23     return 0;
24 }
```

Figure 2.18: `go` function for `GENERATE` block

Another function included in the C API is the `push` function. The push function in the C API of an X block is called whenever a new piece of data arrives at any of its input ports. The implementation code can then check the bitmask given by its portmask to see which input ports have data ready for processing. If all the data the block needs is ready, the block can process the input data and send it over its output ports using the `send` function. Once it has consumed the input data, it can use the `release` function to clear its `portmask`, and additionally free input data memory if required. Figure 2.19 shows the push function for `SUM`.

When multiple `SUM` blocks are present, there is a global push function for all SUM blocks, with the pointer `d` specifying which block instantiation the push function is being called on. In example `test1` there is only one SUM block, so `d` can only point to `sum`. For the function `X_GENERATE_push` (not shown here), the pointer argument `d` can point to either `gen1` or to `gen2`.

```
35 void X_SUM_push(struct X_SUM_data *d) {
36
37     //return data ready at both input ports x0 and x1
38     if(checkPortsUpTo(d->portmask,1)){
39
40         //y0 points to same data as x0
41         d->oport0 = d->iport0;
42
43         //data at y0 = data at x0 + data at x1
44         *(d->oport0) = *(d->iport0) + *(d->iport1);
45
46         //clear portmask bit for x0 but don't deallocate its memory
47         release(0,0);
48
49         //clear portmask bit for x1 and deallocate its memory
50         release(1,1);
51
52         //send the data on oport0
53         send(0);
54     }
55 }
```

Figure 2.19: Push function for `SUM` block

For this discussion, recall that `iport0` corresponds to input port `x0`, `iport1` corresponds to input port `x1`, and `oport0` corresponds to output port `y0`. Line 38 checks whether data is available on both the input ports `oport0` and `oport1`. Line 40 sets the data pointer for `y0` to the pointer for `x0`. On line 47, `release(0,0)` dequeues a data element from the FIFO for port `x0`, but does not free the memory for that data element because that memory is being used for port `y0`. The first `0` refers to the 0th port (`x0`), and the second `0` tells release to not deallocate memory. On line 50, `release(1,1)` dequeues an element from the `x1` port FIFO and deallocates the memory. Finally, the data on port `y0` is sent downstream on line 53.

The final C API function for all X blocks is the destructor function. This function is called for each block at the very end of an X application. Once again, in the case of the `SUM` block, the body of this function is empty, but other blocks might need end of execution operations to perform such as deallocating memory, closing files, etc.

```
32 void X_SUM_destroy(struct X_SUM_data *d) {}
```

2.2.2 VHDL block implementations

The other major implementation language currently supported by X-Com is VHDL. The skeleton VHDL API for the `SUM` block is given in Figure 2.20.

```
1  entity X_SUM is
2      port(
3          clk : in std_logic;
4          rst : in std_logic;
5
6          --input port x0
7          input_x0 : in X_unsigned32;
8          avail_x0 : in std_logic;
9          read_x0 : out std_logic;
10
11         --input port x1
12         input_x1 : in X_unsigned32;
13         avail_x1 : in std_logic;
14         read_x1 : out std_logic;
15
16         --output port y0
17         output_y0 : out X_unsigned32;
18         write_y0 : out std_logic;
19     );
20 end X_SUM;
21
22 architecture arch of X_SUM is
23 ...
29
30 end architecture arch;
```

Figure 2.20: VHDL implementation skeleton for `SUM` block

The code generation for HDL blocks automatically inserts FIFOs between blocks to manage data transfer. This allows any block to "pull" data from upstream FIFOs whenever it wants as long as data is available from upstream blocks and "push" data to downstream FIFOs whenever data produced by the block itself is ready.

Let us examine how the interfaces to input and output ports to FIFOs work. Consider first the input port `x0`, the code for which is given on lines 7–10. `input_x0` is the data line for the upstream FIFO for port `x0`. It is set to the first available data in the FIFO, and undefined when no data is

available. `avail_x0` goes high when data is available in the upstream FIFO. `read_x0` tells the upstream FIFO that this X block has consumed the first available piece of data. It causes the FIFO to discard the first data item and move to the next one if any more data is in the queue.

Consider now the output port `y0`, given on lines 17–19. `output_y0` is the output data line from the X block. The X block asserts `write_y0` high when it wants to write the data on `output_y0` to the downstream FIFO.

The code for the internal architecture of the VHDL block is given in Figure 2.21, illustrating how the input and output ports are used.

```
22 architecture arch of X_SUM is
23     signal all_rdy : std_logic;
24 begin
25     all_rdy <= avail_x0 and avail_x1;
26     read_x0 <= all_rdy;
27     read_x1 <= all_rdy;
28     output_y0 <= unsigned(input_x0) + unsigned(input_x1);
29     write_y0 <= all_rdy;
30 end architecture arch;
```

Figure 2.21: Architecture body for VHDL implementation of the `SUM` block

`all_rdy` is an internal signal, similar to the `portmask` variable from the C API, that is used to keep track of when all the required data is ready at input ports. Once this signal goes high, the `SUM` X block writes the sum of the two inputs to the downstream FIFO, and "reads" from the upstream FIFOs, letting them know that it has used the first pieces of data from both of them.

2.2.3 Compilation of Deployable Executables

We have shown how the input to X-Com is a set of user-created X Language files, supplemented by library files, that fully describes the application. From these inputs, X-Com creates a source file for each of the targeted CRs in the CR's specific language. In the next stage of compilation, CR specific compilers are used; the GNU C Compiler [11] for processor CR targets and the ModelSim [15] HDL

compiler for FPGA CR targets. These compilers take in X-Com generated source files and user-defined implementation files to create deployable executables for the targeted CRs. Library files that contain platform specific definitions and functions are also fed into the platform specific compilers. The whole process of compiling X Language code into CR-specific code, and then compiling the generated CR-specific code and user-defined implementation code into deployable executables is managed by a manually generated Makefile. Future work includes automating the creation of this compilation flow controlling Makefile.

2.3 X-Sim: Simulating Applications and Collecting Traces

When targeting a complex hybrid architecture, it is useful to run a simulation run first. Hardware implementations are simulated in ModelSim [15], an HDL simulation and verification tool, while software implementations may be simply run natively on a development processor. A trace file based system is used as the interconnect resource (IR) between all the simulation CRs.

To set up the simulation, the algorithm description file `test1_algo.x`, presented earlier in Figure 2.4, is used since the algorithm does not change when running a simulation. The target architecture, however, does change as shown in Figure 2.22.

Hardware implementations are now targeted to `VHDLSim`, a CR that represents ModelSim. The definition for `VHDLSim` is given in the library file `std.x`. Software implementations are now targeted to `C`, the base C language platform defined in `std.x`. Both of the IRs are declared to be of type `FileIO`. `FileIO` is the simulation trace file communication mechanism. It is defined in the library file `fileio.x`. Since the CR and IR names are the same as before, the mapping file `test1_map.x` shown previously in Figure 2.12 can still be used, with the only difference being that the alternate architecture file `test1_simarch.x` will be included instead of `test1_arch.x`. Additional details for `VHDLSim` and `C` platforms and the `FileIO` linktype are given in Chapter 3.

```
1  #include "std.x"
2  #include "fileio.x"
3
4  //definition for VHDLSim and C CR platforms given in included file std.x
5  //definition for FileIO IR platform given in included file fileio.x
6
7  //---CR declarations---
8  resource fpga is VHDLSim;
9  resource proc[3] is C{
10             (freq=3400), (freq=3400), (freq=2800)
11 };
12
13 //--IR declarations---
14 resource LAN is FileIO (
15             { proc[1], proc[2], proc[3]}
16 );
17 resource PCI is FileIO (
18             { proc[1], proc[2], fpga}
19 );
```

Figure 2.22: `test1_simarch.x`: Sample simulation architecture description

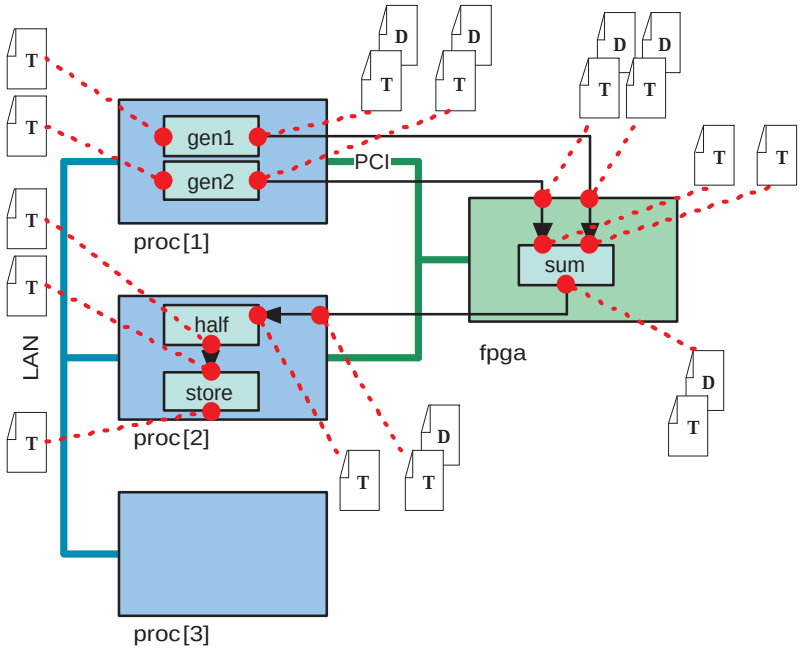


Figure 2.23: Trace capture points for `test1`

After running the simulation, trace files are generated that store details of communication between CRs as well as the start of execution of data source blocks and the end of execution of data sink blocks. For example, the data and timing trace files stored for the `test1` example is shown in Figure 2.23. 'D' files in the figure represent data trace files, and 'T' files represent timing trace files.

Data trace files can be examined by the user to check if they have the correct values. Once correctness has been established from the data trace files, the timing trace files are passed to X-Eval for performance analysis of the application. Refer to Chapter 3 for additional details on both data and timing trace files, as well as for more details about the mechanism of an X-Sim simulation run.

2.4 X-Eval: Analyzing Simulation Timing Traces from X-Sim

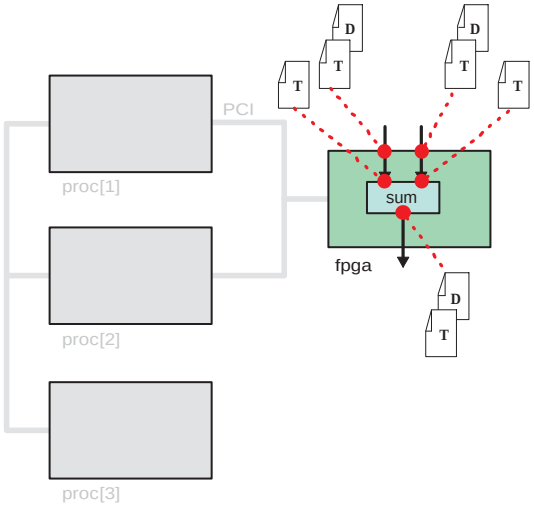


Figure 2.24: Trace capture points for `fpga` CR

Timing trace files are used by X-Eval to characterize the performance of CRs. Consider the `fpga` CR with the `SUM` block mapped to it, shown in Figure 2.24. The `SUM` block has two input ports `x0` and `x1`, and one output port `y0`. Timing trace files have been captured for each of these ports. X-Eval subtracts the input times from the output times to figure out the execution times for the CR. This history of execution times is then used to generate performance metrics such as the mean and variance of execution times. Further analytic modeling for specific CRs as well as for the entire

system can be done. This will be examined in more detailed in Chapter 4, along with more details of trace analysis methods and performance characterization.

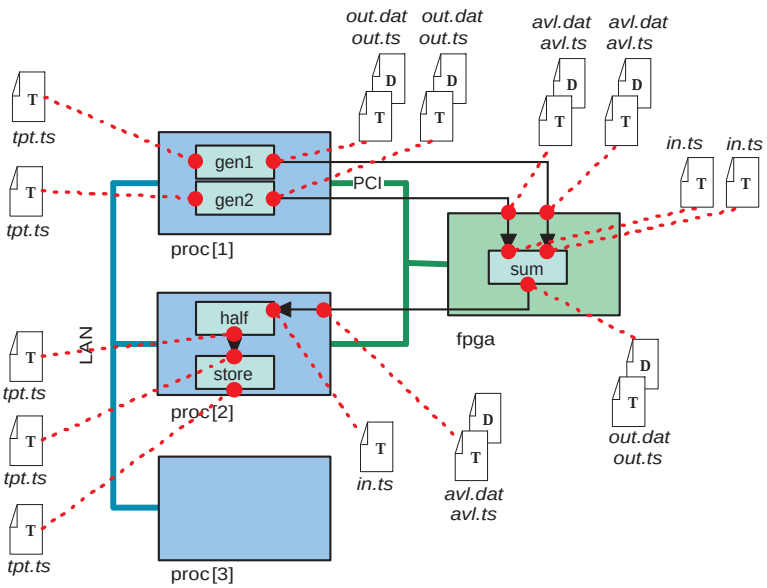
2.5 X-Opt: Performance Optimization

The analytic models generated by X-Eval can be used by X-Opt to analyze the performance of the entire system, and to identify system bottlenecks with the goal of creating more optimal mappings of the algorithm to the processing architecture. X-Opt has not been developed yet, so currently these mappings are generated manually. Mappings created manually or by X-Opt must be re-evaluated to determine performance behavior. X-Sim allows rapid evaluation of re-mappings by speeding up simulations using various techniques. One technique is to simulate independent CRs in parallel. Another is to use analytic models or trace timing data from previous simulation runs as a substitute for re-simulating a CR. Rapid re-simulation techniques allow re-mappings to be simulated and evaluated much faster, allowing X-Opt or the user to rapidly try out a large number of mappings. Simulation speedup techniques are discussed in more detail in Chapter 5.

Chapter 3

Simulation using X-Sim

Thus far, we have described Auto-Pipe as a holistic application development system, explaining how X-Sim and X-Eval fit into the wider system. This chapter delves in more detail into the X-Sim simulation tool, using example `test1` (Figure 3.1) introduced in the previous chapter to illustrate various aspects of this tool.




```
map { top.gen1, top.gen2 } to proc[1];
map { top.sum } to fpga;
map { top.half, top.store } to proc[2];
```

Figure 3.1: Traces gathered for sample `test1` mapping

The first section presents a step-by-step description of how X-Sim simulates the example mapping. The next section describes usage of the X-Sim tool, presenting a tutorial for how to set up an X-Sim simulation for an application developed under the Auto-Pipe system. The final section discusses the trace files generated by X-Sim.

3.1 An Example X-Sim Simulation Run

This section presents the X-Sim simulation run for the `test1` example. Figure 3.1 shows the traces that are created at the end of the entire simulation run. Note that the timing trace files on the left marked `tpt.ts` are `testpoint` timing trace files, generated only when the user specifies that they want trace data to be captured for those specific points in the algorithm. All the other trace files are generated by default because they store traces for communication that happens on algorithm edges mapped to interconnect resources (IRs). Details of how to capture arbitrary `testpoint` traces are given in the next section.

For the given `test1` mapping, three different computation resources (CRs) must be simulated, `proc[1]`, `fpga`, and `proc[2]`. Under X-Sim, a separate simulation is used for each CR. In this case X-Sim integrates the three simulations into a single federated simulation. The `proc[1]` and `proc[2]` CRs are both instances of the platform `C`, which represents a general purpose processor (GPP), while `fpga` is an instance of the platform `fpga`, a physical FPGA chip. To simulate a GPP, X-Sim may use a *native execution* model; that is, the deployable binary is run directly on the target GPP. Execution of the program on the native GPP in effect acts as a simulation.

If the target GPP (called the *physical target CR*) is unavailable, then a similar GPP (called the *simulation target CR*) can be used to replace the target GPP in the simulation. For example, a simulation target Pentium 4 with a 2.8GHz frequency might be used to simulate a physical target Pentium 4 with a 3.2GHz frequency. Functionally the simulation matches the end deployment exactly. Scaling the performance results (e.g., timing, latencies) can be done to roughly account for the difference in frequencies of the simulation and target machines.

ModelSim [15] is used to simulate the FPGA chip. It acts as the simulation target to the physical target FPGA chip. ModelSim is a function and timing model simulator for hardware designs authored in various HDLs (Hardware Description Languages), including VHDL and Verilog. It is primarily a GUI based simulation environment, using wave diagrams of signals to allow functional and timing debugging of hardware modules. However, in addition to the GUI, ModelSim also provides a command line interface. X-Sim generates scripts that uses the non-GUI text-based interface to run ModelSim.

X-Sim simulates the Interconnect Resources (IRs) `LAN` and `PCI` by modeling them with a file system based communication mechanism, `FileIO`. Any message sent on a target IR gets simulated by a write to a trace file, and correspondingly a message receipt from an IR gets simulated by a read from a trace file. Examples and details of trace files are given at the end of this chapter.

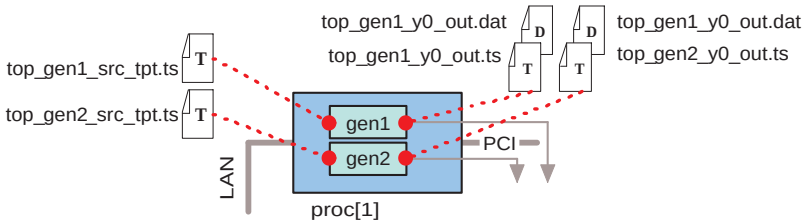


Figure 3.2: Simulation of `proc[1]`

For the given mapping being simulated, X-Sim does a series of federated simulations. Refer to Figure 3.2, which shows the first simulation done by X-Sim, where a complete native-execution simulation of `proc[1]` is run. The timing traces on the left end of the `GENERATE` blocks store a complete history of the starting times for the execution of these blocks. These `testpoint` traces are recorded at points specified by the user. The data and timing traces on the right end of the `GENERATE` blocks store all the data writes done to the `PCI` IR by `gen1` and `gen2` on their outputs. These output time traces are called `out.ts` timestamps. It is important to note here that the shown federate simulation is run *to completion*, so the trace files created store a complete record of timings and data produced over the entire simulation duration for this CR. The simulation for `proc[1]` must be finished and a complete record of output traces must be generated before the following simulations can be done.

The main reason to record the user-defined `tpt.ts` timing traces is to gather timing data for points in the algorithm that do not correspond to edges mapped to an IR. In this case, gathering the timing traces at the beginning of the `GENERATE` blocks allows the simulation to record start times for executions of the `proc[1]` CR. By looking at the `tpt.ts` and `out.ts` timing traces recorded for `proc[1]`, X-Eval can later reconstruct a history of the execution times for the CR. Currently no mechanism is provided for recording any *data* values at the arbitrary testpoints. Data recording is not necessary for timing analysis, which is the main purpose of the testpointing feature. However, it may be a useful debugging feature that can be incorporated in a future version of X-Sim.

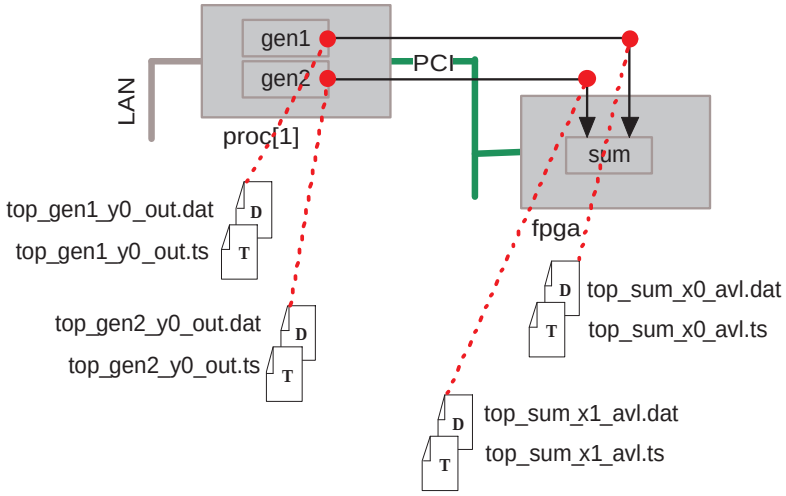


Figure 3.3: Simulation of communication between `proc[1]` and `fpga`

After the simulation for `proc[1]`, the communication modeling tool X-Model is used. X-Model reads the output (`out.ts`) timing trace files generated in the previous simulation step, adds a modeled communication delay to the times, and generates the `avl.ts` timing traces on the right of Figure 3.3. These timing traces represent the times data became available to the downstream CR `fpga`.

Complete traces of data and the times the data was available to `fpga` are now available for the third simulation step. ModelSim is now used to simulate `fpga` (see Figure 3.4). X-Sim reads the `avl.ts` timestamps to determine when it is allowed to read data from the input data traces. When ModelSim reads in the data, it stores the data input and associated times (`in.ts`) in a set of timing traces files (Figure 3.4). It is possible for the `in.ts` and `avl.ts` times for a piece of data to be

different if the data becomes available to be read, but the FPGA is busy processing a previous piece of data and thus cannot input the data that is available. In effect, the data that is available is kept in an input queue until the FPGA brings it in for processing. The difference between the `in.ts` and `avl.ts` times is thus the queue waiting time for a piece of data. After processing, ModelSim writes data and timing trace files on the output of `fpga` in the trace files labeled `out.dat` and `out.ts`.

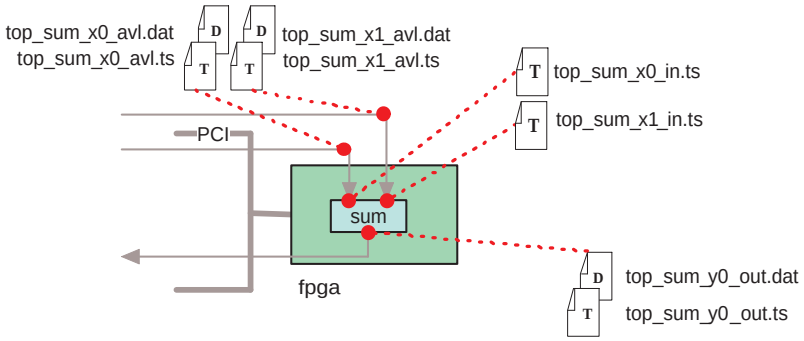


Figure 3.4: Simulation of `fpga`

At this point, X-Model is used to simulate communication of data from `fpga` to `proc[2]`, similar to its use in Step 2 of the X-Sim simulation. Figure 3.5 shows this process. The data and timing (including a communication model delay) are placed in the `avl.ts` timestamp files for use by the simulation for `proc[2]`.

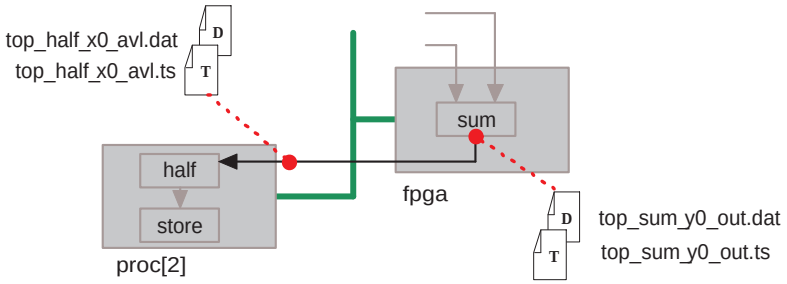


Figure 3.5: Simulation of communication between `fpga` and `proc[2]`

In the final step of the system simulation, as shown in Figure 3.6, a native execution simulation is done to simulate the operation of CR `proc[2]`. Timing trace files with `in.ts` timestamps are again generated, as well as `tpt.ts` timestamps. The next section details how to set up and run the X-Sim simulation example shown here.

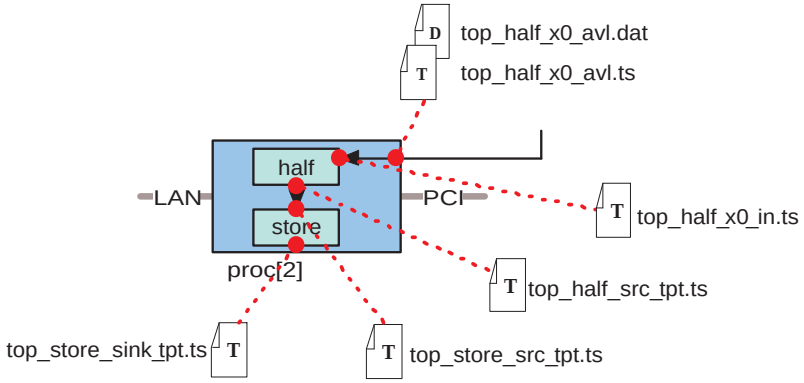


Figure 3.6: Simulation of `proc[2]`

3.2 Tutorial for Setting up a Simulation using X-Sim

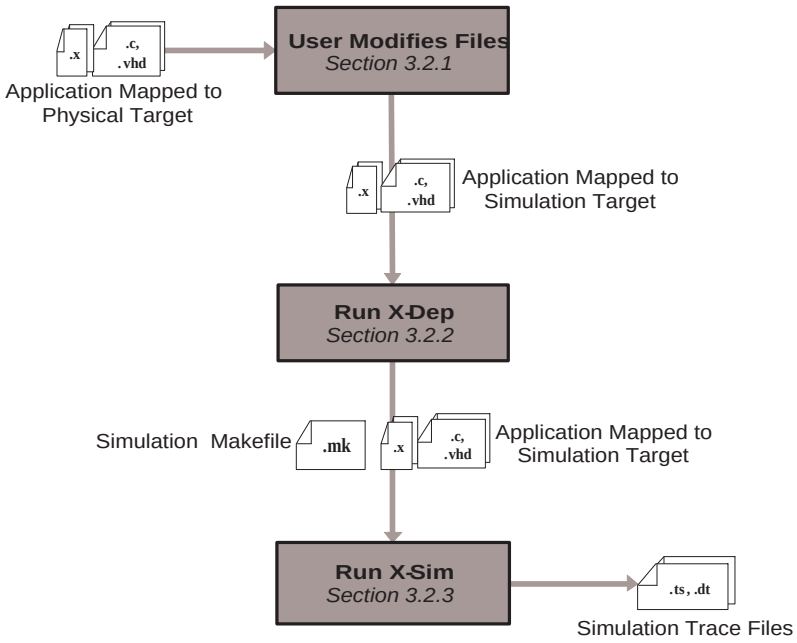


Figure 3.7: Design Flow for X-Sim

Figure 3.7 shows the high level steps involved in setting up a simulation for an application developed under Auto-Pipe. The input set of X Language files represents the description of an algorithm and a physical target architecture, as well as the mapping of the algorithm to the architecture. A *physical* target architecture in this context means an architecture that is comprised exclusively of real world devices such as general purpose processors and FPGA chips. This can be contrasted

with a *simulated* target architecture, which can include simulation targets such as ModelSim. The first step of the X-Sim design flow consists of modifying the X Language files to change the target architecture to a simulated one.

The second step uses the X-Dep tool to generate a Makefile which automates the running of X-Sim. As mentioned in the previous chapter, X-Dep is used to deploy to either a physical or simulation target. In the context of X-Sim, X-Dep is used to deploy the algorithm to a the simulation target. The Makefile created by X-Dep coordinates the running of the federated X-Sim simulation by keeping track of the dependencies between the different simulators.

In the third and final step of the X-Sim flow, the generated Makefile is used with the '`make`' to actually run the X-Sim simulation and generate trace data and timing files. We will continue to use the same mapping for `test1` that we have been using throughout this chapter. The following sections describe the individual steps of the X-Sim flow in more detail.

3.2.1 User Modifications to Files

```
map { top.gen1, top.gen2 } to proc[1];
map { top.sum } to fpga;
map { top.half, top.store } to proc[2];
```

Figure 3.8: Mapping for example `test1`

The X Language statements mapping blocks to CRs for our example is given in Figure 3.8. When deploying to a physical (i.e., non-simulation) target architecture, `proc[1]` and `proc[2]` are both instances of the platform `C`, which represents a general purpose processor, while `fpga` is an instance of the platform `fpga`, a physical FPGA chip. The IRs linking the `proc` and `fpga` CRs are `LAN`, an instance of the `switch_ether` linktype, and `PCI`, an instance of the `bus_pci` linktype. The original physical platforms are shown in Figure 3.9.

Under X-Sim, a separate federated simulator is used to simulate each CR, so in this case X-Sim would use three simulators (one each for `fpga`, proc[1], and proc[2]). This is shown in Figure 3.10. GPPs in X-Sim are simulated natively, so the platform for `proc[1]` and `proc[2]` remain the

```
//−−−CR declarations−−−
resource fpga is HDL_Virtex4( part="XC4V8000");
resource proc[3] is C_Pentium4{
   (freq=3400), (freq=3400), (freq=2800)
};

//−−−IR declarations−−
resource LAN is switch_ether(
     { proc[1], proc[2], proc[3] }
);
resource PCI is bus_pci{
     { proc[1], proc[2], fpga}
};
```

Figure 3.9: *Original* physical deployment targets for example `test1`

```
//−−−CR declarations−−−
resource fpga is VHDLSim; //simulates HDL_Virtex4
resource proc[3] is C_Pentium4{
   (freq=3400, xsim="true"),
   (freq=3400, xsim="true"),
   (freq=2800, xsim="true")
};

//−−−IR declarations−−
resource LAN is FileIO( //simulates switch_ether
     { proc[1], proc[2], proc[3] }
);
resource PCI is FileIO{ //simulates bus_pci
     { proc[1], proc[2], fpga}
};
```

Figure 3.10: Simulation platforms for example `test1`

same, `C_Pentium4`. The difference in the `C_Pentium4` simulation target CR is that the `xsim` option is set to `"true"`. This turns on C language macros for testpoints that are described later in this section. When deployment is done to a simulation target and the `xsim` option is set to `"true"`, the testpoint macros evaluate to a set of commands that cause X-Sim to record `tpt.ts` timing traces. When deployment is done instead to a physical target and the `xsim` option is set to `"false"`, the macros evaluate to blank statements (i.e., the macro is simply *compiled out*). Note that the default setting for the `xsim` option is *false*.

An FPGA platform is simulated by a ModelSim simulation under X-Sim. To specify this, the platform `HDL_Virtex4` is switched to `VHDLSim`, the X-Language platform corresponding to the ModelSim simulator. Similarly, the physical platforms `switch_ether` and `bus_pci` are replaced by the communication simulation platform `FileIO`.

By making these changes to the X Language files, the application has now been configured to run a simulation. In the default configuration, X-Sim captures data and timing trace files at all ports that send or receive data over the simulation IR (Interconnect Resource) `FileIO`. For our example, the automatically generated trace files are shown in Figure 3.11.

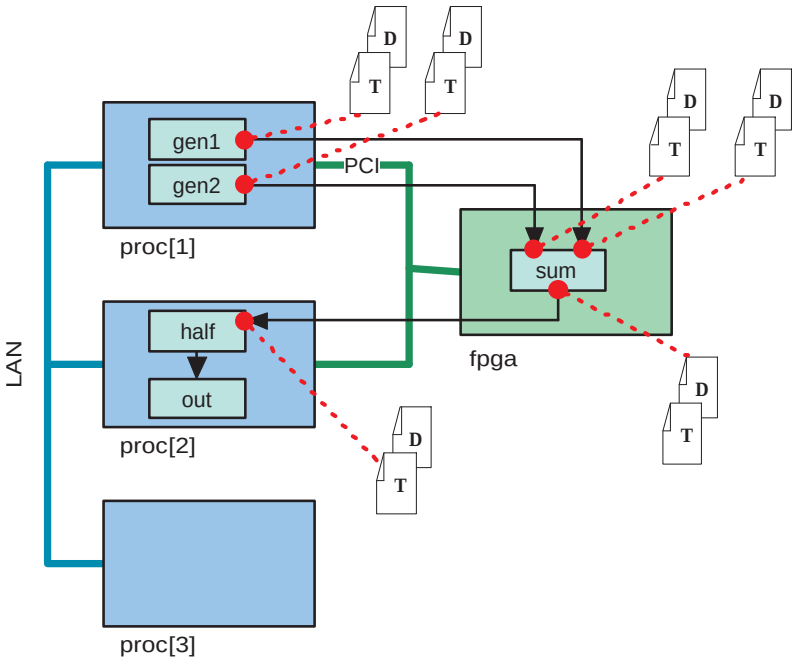


Figure 3.11: *Default* trace files for `test1` (i.e. without `tpt.ts` traces)

```
1 block GENERATE { //generate a number
2     misc {
3         testpoint "src";
4         filepath "temp/Xfileio";
5     };
6     output FLOAT32 y0;
7 };
```

Figure 3.12: X Language description of `GENERATE` block with a testpoint

In addition to the default traces that focus on the inputs and outputs of CRs, X-Sim also supports the ability to record traces at arbitrary points in blocks mapped to processors. To achieve this, testpoints are added to blocks. For example, to add a testpoint named `src` to the `GENERATE` block, the X Language code for the block would get lines 2-4 added to it, as shown in Figure 3.12. The main features on those lines is the specification of `"src"` as the name of the testpoint, and the specification of `"temp/Xfileio"` as the directory where trace files for `GENERATE` block will be stored. Testpoints are implemented under the X Language by making use of the `misc` feature of X Language blocks. The `misc` is a convenient feature of the X Language that allows additional block properties to be added on easily, in this case the testpoint property. To add more testpoints to the `GENERATE` block, simply add another line inside the `misc` code section that defines an additional testpoint. The timestamps for the new testpoint will be stored in a separate timing trace file inside the *filepath* directory. By adding the testpoint to the X Language specification, the X-Compiler knows to generate a function called `tpoint` that implements the ability to record timing traces for testpoints. Note that the generated code is not shown here.

The second change the user needs to do is to add a macro to the implementation header file for the block. Figure 3.13 shows the header file for the `GENERATE` block with the added testpoint macro on line 7. When compilation is done with an added `-DXSIM` option, this macro evaluates to a declaration of a pointer to the `tpoint` function.

The user is now able to record timestamps on the testpoint wherever they wish inside the implementation code. This is shown in Figure 3.14. Line 6 in this figure shows the user recording a timestamp on testpoint "src" before starting to generate a number. When the `xsim` option for the CR is set to

```
1  #include "X.h"
2
3  struct X_GENERATE_data {
4      Xclock_t clock;
5      void (*send)(int);
6      void (*release)(int,char);
7      TPOINTFUNC;
8      portmask_t portmask[1];
9
10     FLOAT32 *oport0; //output port y0
11
12     int count; //number of random numbers to generate
13 };
14
15 void X_GENERATE_init(struct X_genU32_data *d);
16 void X_GENERATE_destroy(struct X_genU32_data *d);
17 int X_GENERATE_push(int p, struct X_genU32_data *d);
18 int X_GENERATE_go(struct X_genU32_data *d);
```

Figure 3.13: Header file for `GENERATE` block with a testpoint

"true", the `TPOINT("src")` macro evaluates to a call to the `tpoint` function with the `"src"` testpoint name.

In a similar manner to the `"src"` testpoint for the `GENERATE` block, two testpoints `"src"` and `"sink"` were added to the `STORE` block. These testpoints record the start and end of processing for this block. Note that the testpoint `"src"` for a `STORE` block is completely separate from the testpoint `"src"` for a `GENERATE` block, similar to how the `y0` port for a `SUM` is completely separate from the `y0` port for a `GENERATE` block. The block definition for the testpointed `STORE` block is given in Figure 3.15, the header file is given in Figure 3.16, and the C implementation code is given in Figure 3.17.

Using testpoints allows the additional `tpt.ts` traces shown in Figure 3.1 to be recorded along with the default traces that get recorded for ports on IRs. Using the testpointing feature thus allows timestamps to be stored at any point in the C implementation code. This is necessary to be able to obtain start and end of processing `tpt.ts` timestamps, which in turn are required to be able to do performance analysis of resources containing source and sink blocks. Additionally, testpointing allows more in-depth timing results to be gathered for any point in block execution the user wants

```
1  int X_GENERATE_go(struct X_GENERATE_data *d) {
2      if(d->count <= 0){ //check if more numbers need to be generated
3          return 1;
4      }else{
5          //record a timestamp for testpoint "src" here
6          TPOINT("src");
7      }
8
9      //allocate memory
10     d->oport0 = gmalloc(sizeof(FLOAT32));
11
12     //generate random number on output port y0
13     *(d->oport0) = rand() % 10 + 1;
14
15     //send the data on oport0
16     send(0);
17
18     //update the count
19     d->count--;
20
21     if(d->count < 0){ //check if this was the last number
22         return 1;
23     }
24
25     //return 0 to signify that this go function should still be called
26     return 0;
27 }
```

Figure 3.14: `go` function for `GENERATE` block with a testpoint

```
1  block STORE { //generate a number
2      misc {
3          testpoint "src";
4          testpoint "sink";
5          filepath "temp/Xfileio";
6      };
7      input FLOAT32 x0;
8  };
```

Figure 3.15: X Language description of `STORE` block with testpoints

```
1  #include "X.h"
2
3  struct X_STORE_data {
4      Xclock_t clock;
5      void (*send)(int);
6      void (*release)(int,char);
7      TPOINTFUNC;
8      portmask_t portmask[1];
9
10     FLOAT32 *iport0; //port x0
11 };
12
13 void X_STORE_init(struct X_STORE_data *d);
14 void X_STORE_destroy(struct X_STORE_data *d);
15 int X_STORE_push(int p, struct X_STORE_data *d);
16 int X_STORE_go(struct X_STORE_data *d);
```

Figure 3.16: Header file for `STORE` block with testpoints

```
void X_STORE_push(struct X_STORE_data *d){
    //store a timestamp for start of execution
    TPOINT("src");

    //set result to value on input port
    float result = *d->iport0;

    //print result to a file
    fprintf(pfile,"Result%f\n",result);

    //release port and free associated memory
    release(0,1);

    //store a timestamp for end of execution
    TPOINT("sink");
}
```

Figure 3.17: `push` function for `STORE` with testpoints

to timestamp. Testpointing is currently only implemented for C implementations. A future revision of X-Sim will include support for testpointing inside VHDL implementations.

3.2.2 Running X-Dep to Create an X-Sim Makefile

After the user has modified the original X language and implementation input files, the next step is to create a Makefile that can be used to run the simulation. The tool used to automatically generate the Makefile is called X-Dep. X-Dep is run simply by calling the X-Dep binary with the name of the top level X Language file, and piping the result into a Makefile. For example, the following line writes the simulation Makefile `sim.mk` for the X application given by `test1.x`.

```
xdep test1.x > sim.mk
```

X-Dep uses the application description given in the X Language to analyze the dependency order of the resources. For example, for the `test1` example, X-Dep will analyze the application to figure out that the order of simulation executions should be to first run `proc[1]`, then to run X-Model to simulate communication from `proc[1]` to `fpga`, then to run the ModelSim simulation for `fpga` and so on. The main parts of the Makefile generated by running X-Dep are shown in Figure 3.18.

The comments at the start of the Makefile identify which X Language file was used. `Xsystem.mk` is an included file that contains the definition for `FILEIOPATH`, a variable that indicates where trace files are stored. The `.perf` files are used to keep track of which simulations have already been done. For example, after the first simulation for `proc[1]` is done, the file `proc_1_.perf` is generated to indicate that the first simulation has been completed.

3.2.3 Running X-Sim

Once the simulation Makefile has been created by X-Dep, all that remains is to actually run the X-Sim simulation. This is achieved by simply running the command "`make simulate`". When this command is run, the simulation Makefile finds that it needs three `.perf` files. These files form

```
1  simulate:  proc_1_.perf  fpga.perf  proc_2_.perf
2      echo  Simulation done.
3
4
5  proc_1_.perf:
6      proc_1_  >&  proc_1_.out
7      echo  Done  >  proc_1_.perf
8
9
10 fpga.perf:  proc_1_.perf
11     xmodel  -i  top_gen1_y0_out.ts  -1  freq=3.4e9 \
12             -o  top_sum_x0_avl.ts  -2  freq=1e9
13     mv  top_gen1_y0_out.dat  top_sum_x0_in.dat
14     xmodel  -i  top_gen2_y0_out.ts  -1  freq=3.4e9 \
15             -o  top_sum_x1_avl.ts  -2  freq=1e9
16     mv  top_gen2_y1_out.dat  top_sum_x1_in.dat
17     vsim  -c  SimModule  -do  "run; quit -f"
18     echo  Done  >  fpga.perf
19
20
21 proc_2_.perf:  fpga.perf
22     xmodel  -i  top_sum_y0_out.ts  -1  freq=1e9 \
23             -o  top_half_x0_avl.ts  -2  freq=3.4e9
24     mv  top_sum_y0_out.dat  top_half_x0_in.dat
25     proc_2_  >&  proc_2_.out
26     echo  Done  >  proc_2_.perf
27
28
29 clean:
30     rm  proc_1_.out  proc_1_.perf
31     rm  fpga.perf
32     rm  proc_2_.out  proc_2_.perf
```

Figure 3.18: Simulation Makefile for `test1`

a dependency order that causes `proc_1_.perf` to be generated first, by executing `proc_1_` to run the simulation for `proc[1]`.

The second `.perf` file in the dependency order is `fpga.perf`. For each of the application edges leading into `fpga`, `xmodel` is run to simulate communication over these edges and generate the required `avl.ts` files. The data files are simply moved over to make them ready for the ModelSim run. After the input files are ready, the ModelSim command line call `vsim` is used to simulate the FPGA.

Finally, the last simulation for `proc[2]` is run after the input files have been readied. The command `make clean` removes the output and `.perf` files from the simulations.

If an application has two resources that can be run in parallel, then the Makefile reflects this parallelism by only expressing the required dependency order and nothing more. For example, if there was a mapping where data was produced in `proc[1]` and then data went out to two output blocks, one in `proc[2]` and one in `proc[3]`, then it is possible to run the two processor simulations in parallel. The simulation for `proc[1]` must still be run first. However, after that the simulations for `proc[2]` and `proc[3]` can be run in parallel because they only depend on the file `proc_1_.perf` created by the simulation for `proc[1]`. Running simulations in parallel and other techniques for simulation speedup will be examined in Chapter 5.

3.3 Simulation Trace Files

In this section, we will describe the timing and data trace files generated in an X-Sim run. There are four types of timing trace files: `avl.ts`, `in.ts`, `out.ts`, and `tpt.ts`. Timestamps inside the first three types of trace files directly correspond to data recorded in data trace files. `avl.ts` timestamps correspond to the times that data became *available* at the input port of a block. `in.ts` timestamps correspond to when data was *consumed* at an input port. `out.ts` timestamps, as one might expect, correspond to when data was output from an output port. In contrast to these types of timing trace files, `tpt.ts` timestamps are not directly associated with a data port, nor do they

correspond directly to any ports. Instead, `tpt.ts` timing trace files store timestamps at points specified by the user *inside* the implementation using macros as was explained in the previous section. The naming convention for timing trace files follows the template:

`<block>_<port|testpoint>_<type>.ts`

For example, the timing trace file `top_gen1_y0_out.ts` has `out.ts` traces for the `y0` port of block `top.gen1`. The file `top_gen1_src_tpt.ts`, on the other hand, has `tpt.ts` traces for the `src` testpoint of block `top.gen1`.

The template `<block>` refers to the full name of a block instance (e.g., `top_gen1`). The `<type>` refers to the type of the port: `avl`, `in`, `out`, or `tpt`. `<port|testpoint>` refers to the port (e.g., `GENERATE`'s output port `y0`) or, in the case of `tpt.ts` timestamps, the name of the testpoint.

Data files have the following naming convention:

`<block>_<port>_<type>.dat`

The templates `<block>` and `<port>` are the same as for timestamp files. Note that a data file can never have a testpoint instead of a port in the name because data traces are not recorded for testpoints inside a function. The `<type>` refers to whether data was recorded coming into or going out of a port. An output port has a `out.ts` and a `out.dat` file associated with it. An input port has a `avl.ts`, `in.ts` and a `in.dat` file associated with it. A testpoint has a `tpt.ts` file associated with it.

During the X-Sim simulation of our example mapping, `proc[1]` is the first resource simulated. The simulation of `proc[1]` produces the trace files shown in Figure 3.19. `GENERATE` blocks were only set to run three times for brevity. The time traces shown here use made up numbers for illustration purposes only, and were not recorded from actual simulations. Actual timing trace results from X-Sim simulations are presented in later chapters.

The `top_gen1_src_tpt.ts` file stores timestamps gathered from the `TIMESTAMP()` call in `gen1`'s `go` function. The `top_gen1_y0_out.ts` file stores timestamps for when the data in the

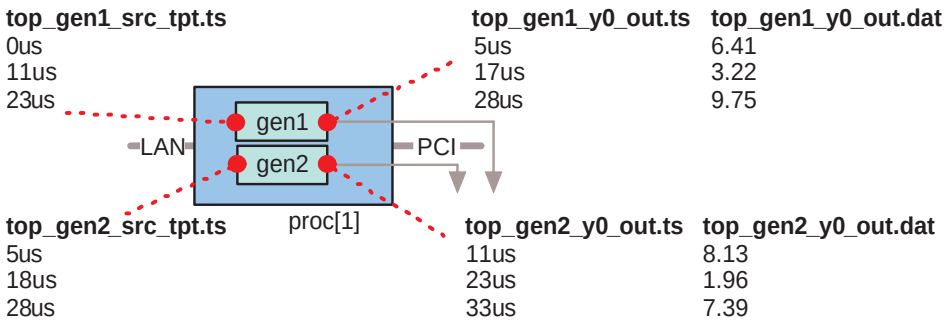


Figure 3.19: Simulation traces for `proc[1]`

`top_gen1_y0_out.dat` file was output from `gen1`'s port `y0`. The other three files parallel the same functions as described, this time for block `gen2`.

3.3.1 Trace File Formats

We will now describe the formats in which data and timestamps are stored in trace files. Data trace files have a straightforward format, storing data directly in binary (rather than ASCII) form. The `top_gen1_y0_out.dat` file, for example, contains `FLOAT32` data values, each of which is stored as a 32-bit binary value.

Timing trace files have a slightly more complex format. Timestamps are stored as 64-bit unsigned binary values. In processor simulations, the system clock for the *simulation target* is directly recorded as timestamp values. Recall that the simulation target is the native processor that is available for running the simulation on. It may be the same as or different from the final *physical target* processor. When the system clock timestamps are combined with knowledge of the *simulation target* processor's clock frequency and the initial system clock time, the simulation times can be accurately calculated. The first 512 bytes of timing trace files are reserved for a header stored in human readable ASCII format. The header for file `top_gen1_y0_out.ts` is shown in Figure 3.20. The first line identifies this files as an X timestamp trace file. The second line stores the simulation processor's clock frequency, which as an example is shown to be 3.2GHz. The third line stores the compile time of the binary being executed in the simulation, purely for additional information for a user. The fourth line stores the offset in clock ticks. The offset is the system clock recorded at the

start of simulation. Note that once again these numbers are synthetic, and that the offset is set as a nice round number to make calculations easier to follow.

```
\#XTSFile
freq=3200000000
\#compile_time=10:36:14
offset=1000000000000000
end
```

Figure 3.20: Header for timing trace file `top_gen1_y0_out.ts`

As an example, if the first timestamp in the trace file was 1000000016000000, then the simulation time would be calculated as given below:

(timestamp - offset) / frequency
= (1000000016000000-1000000000000000) / 3.2GHz = 5us

This calculation tells us that 5μs after the simulation started running, the `gen1` block output the first random number on its `y0` port. In the case of ModelSim simulations, the default frequency stored in timing trace files is 1GHz, because ModelSim simulations are run using ns as the base time resolution. Since timing trace headers are in ASCII, they can be viewed directly, using commands such as `head`.

To view the binary timestamps in the body of a file, a binary viewing tool such as `hexdump` (hexadecimal dump for UNIX) or `od` (octal dump for cygwin) must be used. For example, the command given below will print out all the binary timestamps in a file.

```
od -j512 -t d8 -w8 top_gen1_y0_out.ts
```

The `-j512` option skips the first 512 bytes of the file (the header). The `-t d8` tells `od` that the timestamps are 8 bytes long numbers that should be printed out in decimal format. The `-w8` option causes `od` to print out 8 bytes (one timestamp) per line. The output from the command is given in Figure 3.21. The left column values are offset values inside the timestamp file in octal format (512, 520, 528, 536). The right column values are the recorded system clock values. Simulation times can be calculated by subtracting the offset clock value (1e15) and then dividing these values by the

clock frequency (3.2GHz). These calculations give the values 5μs, 17μs, and 28μs, matching the values that were shown in Figure 3.19.

```
0001000 1000000000016000
0001010 1000000000054400
0001020 1000000000089600
0001030
```

Figure 3.21: Timestamps for timing trace file `top_gen1_y0_out.ts`

3.3.2 Reconstructing a Simulation Run from Traces

This section shows how a simulation run can be reconstructed using the traces generated by a sample X-Sim simulation run, starting with the `proc[1]` simulation traces given previously in Figure 3.19

At simulation time 0μs, the binary for `proc[1]` started executing by calling `go` on block `gen1`. The `go` function then generated the float data value 6.41 at time 5μs. In the binary's round robin schedule, `gen2`'s `go` function was called next. That function generated 8.13 at 11μs. The round robin scheduler then went back and called `gen1`'s `go` function again, and so on.

After `proc[1]`, the communication of data from `proc[1]` to `fpga` over `PCI` is simulated. Assuming that X-Model uses a simple constant 5μs delay to model the communication delay, the traces recorded are given in Figure 3.22. X-Model simply adds 5μs to `out.ts` times to get the corresponding `avl.ts` times. `out.dat` files are copied over to create `avl.dat` files.

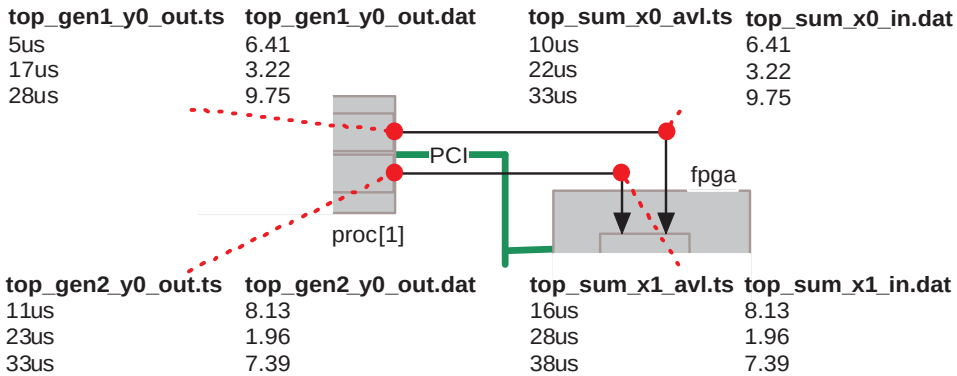


Figure 3.22: Simulation traces for communication from `proc[1]` to `fpga`

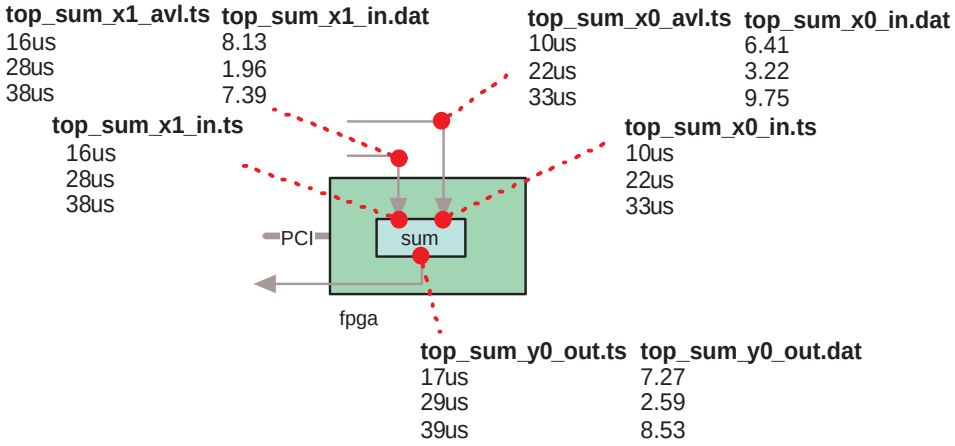


Figure 3.23: Simulation traces for `fpga`

The next simulation is of `fpga`. The trace files for this simulation are given in Figure 3.23. The `avl.ts` and `in.dat` files were set up by the previous simulation. The first piece of data to become available was the float 6.41 on port `sum.x0` at time 10μs. Since the ModelSim simulation was not busy doing any previous processing, it was able to input the data at that time, making 10μs the first `in.ts` time for `sum.x0`. Then the simulation had to wait till time 16μs for the data on port `sum.x1` to become available, at which time the float 8.13 was input into the module. At time 17μs, the simulation finished calculating the sum and output the value 7.27 on port `sum.y`. The simulation then waited till time 22μs so it could input data on port `sum.x0`. It further waited till time 28μs to input data on port `sum.x0`. Calculation of the second sum was complete at time 29μs. Calculation of the third sum was done in a similar manner to the first two, and was completed at time 39μs. The simulation of data transfer from `fpga` over `PCI` to `proc[2]` is again modeled by a constant 5μs delay, and is shown in Figure 3.24.

Finally, the simulation for `proc[2]` is run, generating the traces shown in Figure 3.25. Note that no data is gathered by X-Sim for the testpoint at the end of the `store` block. The `store` block, or any *sink* block, for an application presumably always either records its output data in an output file, prints it out to screen. The user can examine the output file generated by the application to check for correctness. This is independent of the trace recording done by X-Sim. Future work may include the ability to record additional data traces at testpoints, which would allow data to be recorded at the end of sink blocks, as well as any other arbitrary points.

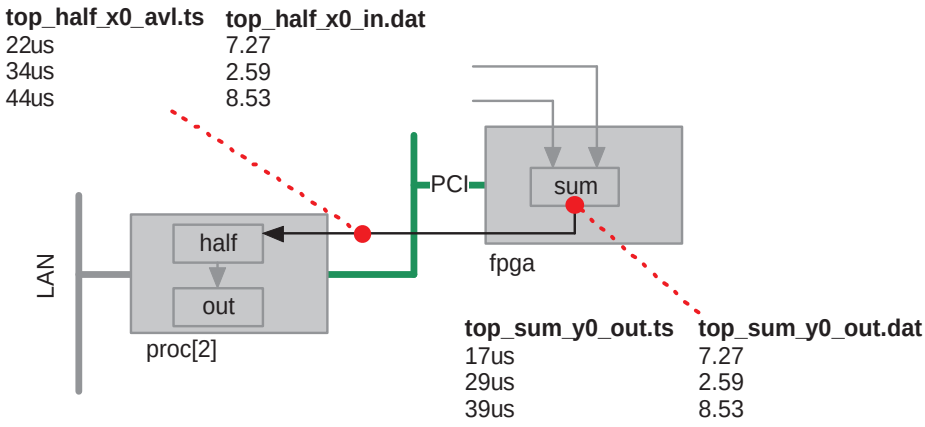


Figure 3.24: Simulation traces for communication from `fpga` to `proc[2]`

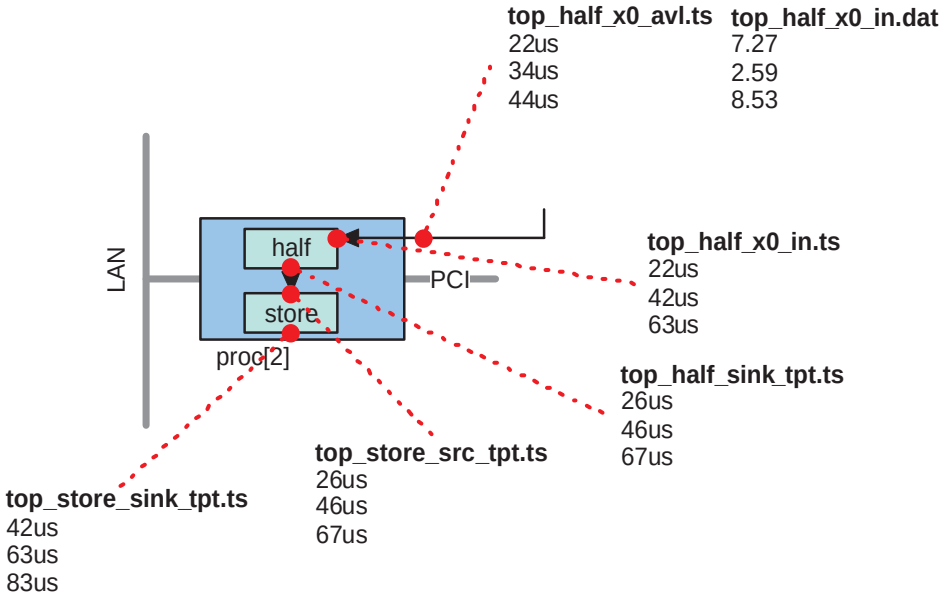


Figure 3.25: Simulation traces for `proc[2]`

The simulation of `proc[2]` is interesting because it shows blocking occurring. The first piece of data is input as soon as it appears, at simulation time 22μs. Processing of this data then took 20μs, resulting in the creation of a `tpt.ts` timestamp at 42μs. Although data was available at `half`'s input port at time 34μs, it is only input into the CR at time 42μs because the CR was busy processing an earlier piece of data. Similarly, the third piece of data on the input port is queued from time 44μs till 63μs before it could be processed. Simulation of the `proc[2]` CR, and of the entire X-Sim simulation, ends at simulation time 83μs.

Resource	Block/Edge		$data_1$	$data_2$	$data_3$	Mean
`proc[1]`	`gen1`	wait time	0μs	0μs	0μs	0μs
		exec. time	5μs	6μs	5μs	5.33μs
	`gen2`	wait time	0μs	0μs	0μs	0μs
		exec. time	6μs	5μs	5μs	5.33μs
`PCI`	`gen1.y0` -> `sum.x0`	wait time	0μs	0μs	0μs	0μs
		exec. time	5μs	5μs	5μs	5μs
	`gen2.y0` -> `sum.x1`	wait time	0μs	0μs	0μs	0μs
		exec. time	5μs	5μs	5μs	5μs
`fpga`	`sum`	wait time	0μs	0μs	0μs	0μs
		exec. time	1μs	1μs	1μs	1μs
`PCI`	`sum.y0` -> `half.x0`	wait time	0μs	0μs	0μs	0μs
		exec. time	5μs	5μs	5μs	5μs
`proc[2]`	`half`	wait time	0μs	8μs	19μs	9μs
		exec. time	4μs	4μs	4μs	4μs
	`store`	wait time	0μs	0μs	0μs	0μs
		exec. time	16μs	17μs	16μs	16.33μs

Table 3.1: Performance results for `test1`

Table 3.1 shows a summary of all the timing results from traces collected for our `test1` example. These results can be gathered by using X-Eval, as will be shown in the next chapter.

3.4 Limitations of X-Sim

This section talks about limitations on application mappings that can be simulated using X-Sim. As explained before, X-Sim is a federated simulation system where individual simulations for each CR

are run in the correct dependency order. For example, for the mapping that we have been looking at for our `test1` example (e.g. Figure 3.1), the simulations are run in the following order:

- `proc[1]`
- communication of data from `proc[1]` to `fpga`
- `fpga`
- communication of data from `fpga` to `proc[2]`
- `proc[2]`

The native execution simulation for `proc[1]` is run to completion, generating complete output data and timing traces for ports `gen1.y0` and `gen2.y0`. None of the blocks mapped to `proc[1]` need data from blocks mapped to other CRs, so the simulation for `proc[1]` can be the first to run. The `sum` block mapped to `fpga` is dependent on data produced by `proc[1]` and communicated over `PCI`. Thus, both the simulations for `proc[1]` and for the communication of data from `proc[1]` to `fpga` must be done, before the simulation for `fpga` can be run.

Let us now consider the mapping shown in Figure 3.26. In this mapping, `gen1` and `gen2` are mapped to `proc[1]`, and `sum` is mapped to `fpga` as before. However, the `half` and `store` blocks are now mapped to `proc[1]` instead of to `proc[2]`. In this mapping, the simulation for `proc[1]` cannot be run to completion because the `half` block needs data from the `sum` block. Output data and timing traces for the `sum` block are only created once the simulation for `fpga` has been run. Thus, the simulation for `proc[1]` is dependent on the simulation for `fpga`. However, the `sum` block (which is mapped to `fpga`) needs data from the `gen1` and `gen2` blocks (which are mapped to `proc[1]`), so the simulation for `fpga` is dependent on the simulation for `proc[1]`. There are thus cyclical dependencies between the simulations of CRs `proc[1]` and `fpga`. Although the given mapping is possible to implement using the X Language, its simulation is not supported by X-Sim.

If a mapping can result in cyclical dependencies, then the current X-Sim design is not able to simulate it. The simulation for a CR needs to have a complete record of all the traces it is dependent

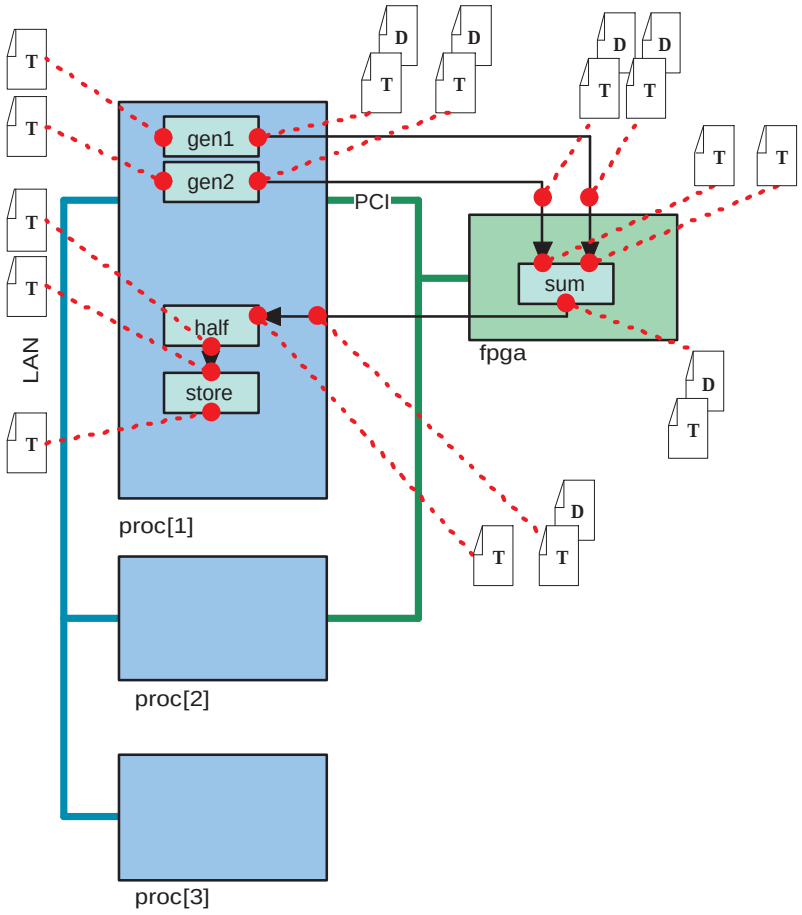


Figure 3.26: Cyclical mapping for example `test1`

on before it can start running. A cyclical dependency between two CRs results in a situation where both CRs are waiting for the other to be run so that they have all the required traces.

A possible future way to overcome this limitation is to run multiple simulations in parallel, feeding data to each other via pipes. For example, the simulations `proc[1]` and `fpga` can be run in parallel, with `gen1` and `gen2` producing data on `proc[1]`, `sum` operating on that data on `fpga`, and `half` using the summed data on `proc[1]`, all simultaneously. With parallel simulations that feed data to each other simultaneously, a simulation can pause execution when it is waiting for data from other simulations. A mapping that would, in real life, not cause any deadlocks would not cause any deadlocks for the simulation either. The ability to run cyclically dependent simulations will be added in a future version of X-Sim.

A related limitation of X-Sim is the simulation of IRs does not accurately model communication delays if cyclical dependencies occur. Cyclical dependencies in IRs are often caused by multiple edges sharing a common IR. For example, data entering the `fpga` CR has to first be simulated being transferred over the `PCI` IR. The `fpga` CR simulation is thus dependent on the `PCI` IR simulation. However, data exiting the `fpga` CR also has to be simulated being transferred over the `PCI` IR. The `PCI` IR simulation is thus dependent on the `fpga` CR simulation. Once again, we see a cyclical dependency, this time between the simulations for `PCI` and `fpga`. Cyclical dependencies between IR simulations and other simulations are particularly common in mappings, because often the same IR links multiple CRs together.

To solve this problem, X-Sim simulates each edge in isolation. For example, the two input edges into `fpga` and the output edge out of `fpga` are all independently simulated. The two input edge simulations are done before the `fpga` simulation, while the output edge simulations is done after the `fpga` simulation. The downside to this independent edge simulation approach is that it does not take into account the effect of IR sharing between multiple edges. For a single edge simulation, the communication simulator is smart enough to queue data transfers while applying latency delays, and make sure that multiple data elements cannot be transferred over the same medium at the same time. However, multiple edges are simulated independently, so the communication simulator cannot account for data being transferred over the same medium simultaneously over *different* edges. Because of this problem, communication latency over an IR is modeled well, but bandwidth sharing between multiple edges is not.

This may be a serious issue in applications where multiple edges mapped to the same IR and one or more edges take up a significant portion of the IR's bandwidth. In such cases, the traffic on a high traffic edge can significantly affect the performance of traffic on other edges. The long-term solution to this problem is, again, to implement the ability to run simulations in parallel. All the edges mapped to a single IR are mapped by a single communication simulator. This communication simulator is run in parallel to the simulations for all CRs, so that cyclical dependencies can be taken into account. The ability to model the performance of multiple edges mapped to a single IR is already available, in the form of the `xmodel2` tool. Combining this multiple-edge simulator with the ability (in a future X-Sim version) to run multiple simulators in parallel will allow the

performance of IRs to be more accurately modeled. The `xmodel2` tool will be explained in more detail in Chapter 5.

In this chapter, we described how X-Sim can be used to create a comprehensive trace record of the operation of an algorithm mapped to a hybrid target architecture. In the next chapter, we will describe how X-Eval can use the generated trace files to analyze the algorithm and mapping, and to provide insight to the user to help optimize the application.

Chapter 4

Analysis using X-Eval

After a simulation run is done, an extensive set of data and timestamp traces are available that capture a history of the federated simulation execution. Traces are recorded at all algorithm edges that are mapped to interconnect resources (IRs), as well as at any explicitly marked trace testpoints. The mapping of example `test1` reproduced in Figure 4.1 will again be used as an illustrative example.

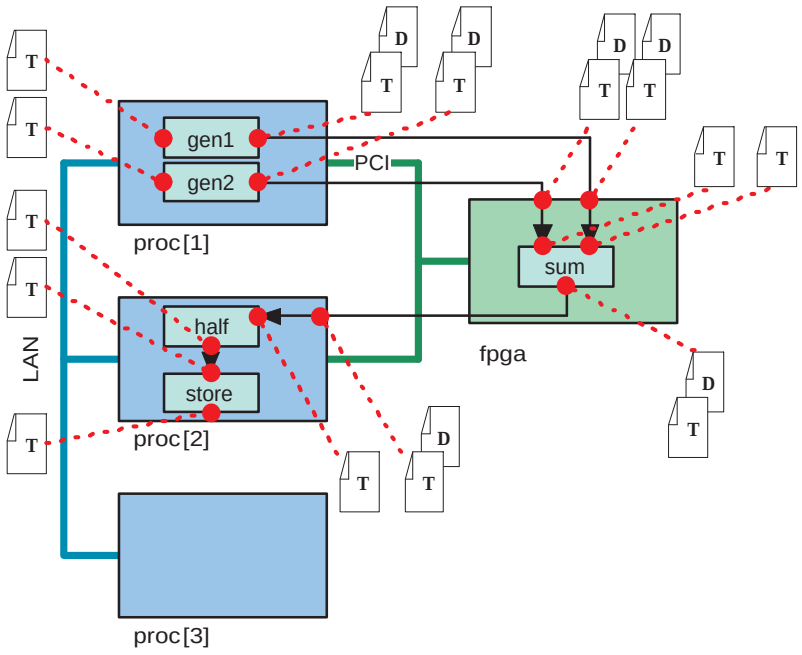


Figure 4.1: Traces recorded for example `test1` (Same as Figure 2.23)

This chapter tackles the problem of how to do performance analysis on simulation traces. The first section in this chapter, Trace Visualization, shows how X-Eval can generate a graphical event timeline that shows a visual representation of all the events recorded in X-Sim trace files. The next section shows X-Eval being run to analyze the `test1` application. The third section, Production Rules in X-Eval, describes how *production rules* can be used to characterize the operation of blocks and allow performance metrics for blocks to be calculated. Finally, the last section gives a tutorial on how a user can run X-Eval on their application to generate performance results.

4.1 Trace Visualization

Traces are stored in binary in a number of different trace files, with one trace file recorded for each port. While binary dump tools provide an instant mechanism to view the raw data in trace files, they do not provide a convenient method to understand system wide simulation trace results. As part of the analysis tool-set, X-Eval can process all the trace files recorded by X-Sim and combine these traces into a single global event timeline. Initial prototypes of the timeline graph were developed in cooperation with Greg Galloway, a student working with the SBS group.

An example of a timeline graph that is automatically generated by X-Eval is shown in Figure 4.2. This timeline shows the traces generated for a simulation run of `test1`. The same timeline zoomed in on the first 100μs of the simulation run is shown in Figure 4.3. Note that for these timeline graphs, the `test1` application simulation was configured to generated 100 values in each of the two `GENERATE` blocks, rather than 3 values as before. This X-Eval generated graph shows all the events that happened during the simulation on one combined timeline graph. Each line on this timeline corresponds to the traces recorded in a different timestamp trace file. For example, the first line displays all the traces that were recorded for the testpoint for the beginning of processing for the `gen1` block. The second line shows the traces recorded at the output of the same block. Lines 3 and 4 show the corresponding events for block `gen2`. The next two lines show the `avl.ts` and `in.ts` traces recorded at port `x0` of block `sum`. The rest of the lines display the remaining traces gathered for the simulation run.

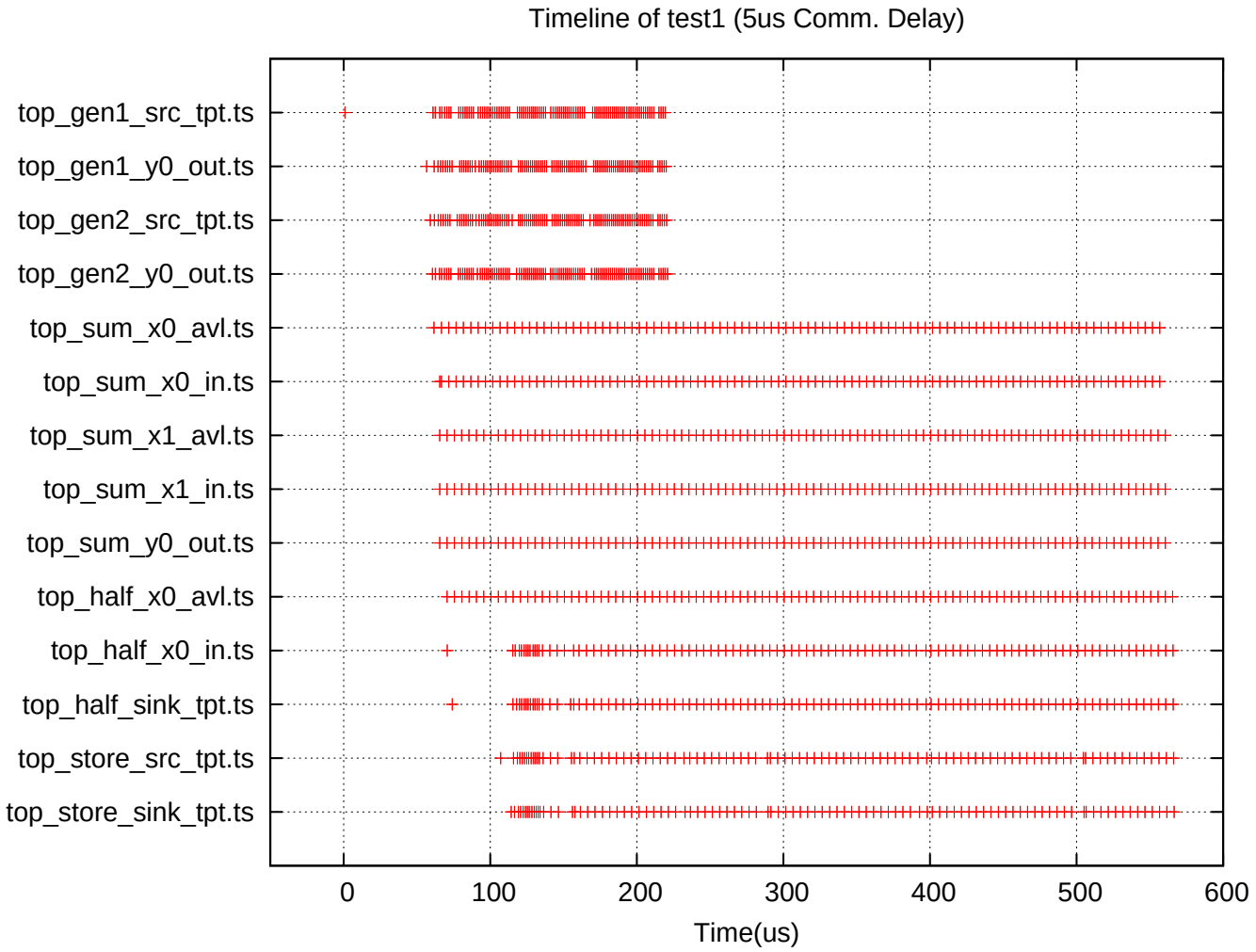


Figure 4.2: Timeline generated for `test1`

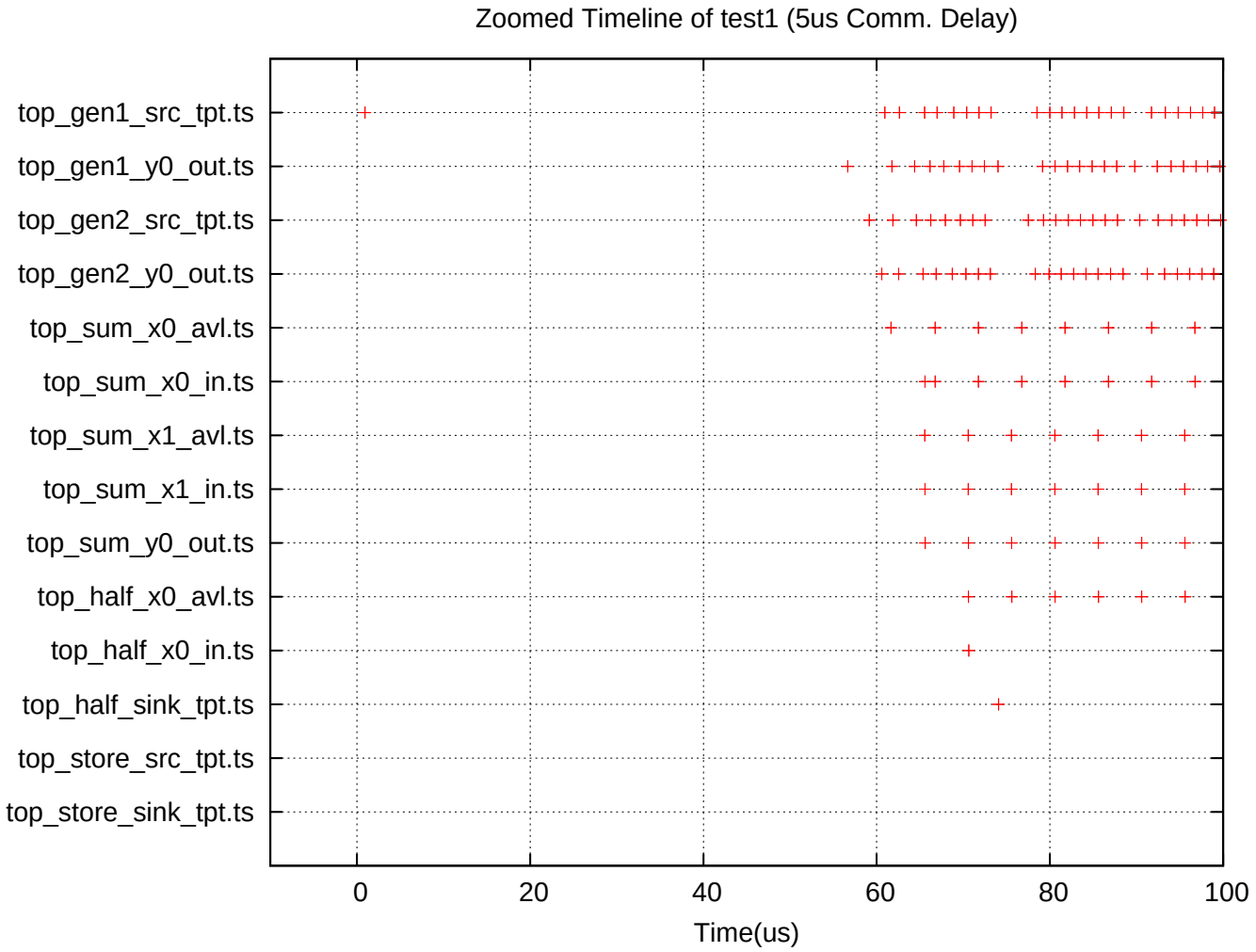


Figure 4.3: Timeline generated for `test1` (zoomed in version)

Let us now consider a sequential reading of the timeline, starting from the left and going across to the right. The start of the simulation corresponds to the first `gen1.src` timestamp. This represents the time that the CR `proc[1]` started running. After running for approximately 50μs, `proc[1]` outputs a data value on `gen1`'s output port. The timeline shows that the next timestamp is recorded at `gen2`'s `src` testpoint, indicating that processing on `proc[1]` switches to the `gen2` block. Generation of the first number from block `gen2` happens very soon afterword. Processing switches between `gen1` and `gen2`, as expected from the round robin scheduling implemented in the X Language. The shown timeline is for a simulation run where each `GENERATE` block is configured to output 100 elements before terminating. From the timeline, we can see that generation of the last 199 numbers from `proc[1]` takes about 150μs, while the generation of just the first number alone took about 50μs. This can be attributed to caching effects, because the first time number generation function is called on `proc[1]` it must be loaded into cache from main memory. For all but the first number, the instructions for number generation are present in cache and thus runs much faster.

After data is produced on the output ports on `proc[1]`, it must travel over the `PCI` IR. In our simulation run, the communication delay model for the `PCI` IR is a 5μs delay per data element. The first data on `gen1.out` is produced around the 50μs mark. The first data becomes available on the other side of the interconnect resource 5μs later, as shown on the timeline marked `top_sum_x0_avl.ts`. Thereafter, data is produced on `gen1`'s output port in a steady stream. The communication delay simulator X-Model has to wait the modeled 5μs to transfer each data element before it can move to the next element. This constant delay turns out to be a bottleneck, as data that was produced within a period of 150μs is spread out over a period of 500μs. A simple calculation shows that 100 data elements transferred over a communication link with a fixed delay model of 5μs must accrue a delay of at least 500μs, so the timeline shows that the simulation acted in accordance with theoretical expectations, and that the communication link is a bottleneck for this application.

Data on the output of the `fpga` CR is produced almost as soon as data is available at the CR's inputs. Data elements at the input of the `proc[2]` CR's `half` block are available every 5μs, still reflecting the bottleneck effect of delays caused by the `PCI` IR. Once again, we can see a caching effect in the way that the first processing of the `half` block takes a lot longer than subsequent executions. The

last time for the entire simulation timeline can be found on the line marked `print.out`, at about simulation time 560μs.

A preliminary visual inspection and analysis of the generated timeline thus shows that the communication link `PCI` is the bottleneck in the simulation. To test this theory, another simulation was run with the same mapping and application, where the communication delay for the `PCI` IR is modeled as a constant 0μs delay (Figure 4.4).

Total processing time for the simulation run has approximately been cut in half, from 560μs to 280μs, confirming our observation that communication delay over `PCI` was a critical bottleneck in the system. This simple re-run of the simulation for `test1` shows how the X-Sim infrastructure, along with simple trace visualization techniques, help users to understand application performance, identify bottlenecks, and test out hypotheses.

In addition to visualization of X-Sim traces, X-Eval is also capable of more in-depth performance analysis of applications developed under Auto-Pipe. The next section takes a look at how this is done for the `test1` example application.

4.2 An Example X-Eval Performance Analysis

In this section, we will look at how X-Eval does performance analysis on each block of the `test1` example application. To keep things simple, we will use the same traces as were shown in the simulation run described in Figures 3.19, 3.22, 3.23, 3.24, 3.25. Recall that for that simulation run, the `GENERATE` blocks were configured to generate 3 values each.

Consider the first block `gen1`. The `top_gen1_src_tpt.ts` timestamp file, generated by the user-defined testpoint `src`, captures each time the `gen1` started generating a number. The `top_gen1_y0_out.ts` timestamp file, on the other hand, captures each time the `gen1` finished generating a number and output it on its output port `y0`. The time each number generation took for

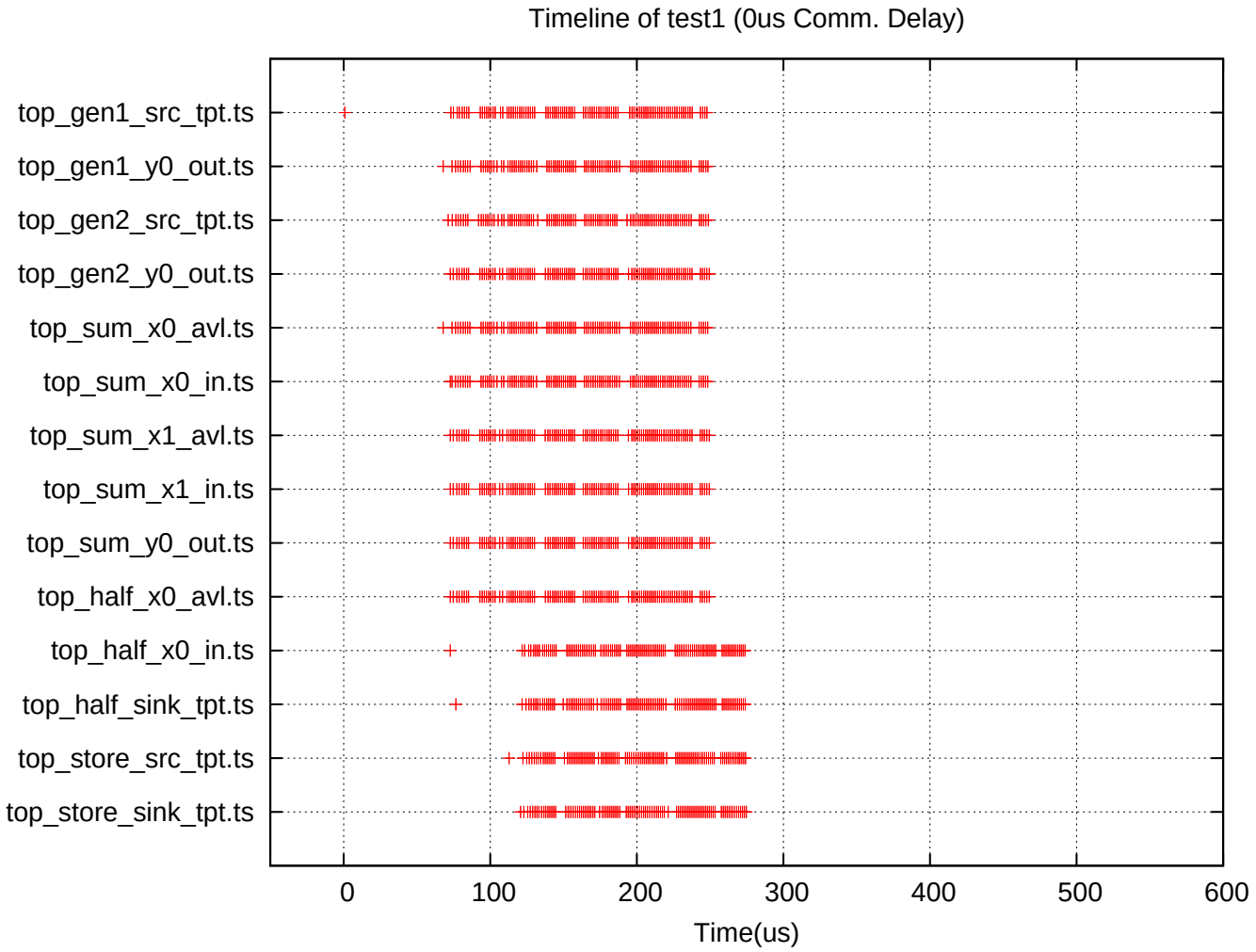


Figure 4.4: Timeline for `test1` with 0μs `PCI` delay

block `gen1` can thus be calculated by subtracting each output `top_gen1_y0_out.ts` timestamp from the corresponding testpoint `top_gen1_src_tpt.ts` timestamp. This relationship of timestamps for block `gen1` can be represented by the following *production rule*:

`gen1.src` $\rightarrow$ `gen1.y0`

X-Eval views blocks as entities that run production rules. A production rule consists of two parts, an input condition (on the left side of the $\rightarrow$), and an output action (on the right side of the $\rightarrow$). The input condition specifies which timestamps cause the rule to be invoked, or *triggered*. For block `gen1`, a timestamp on the `gen1.src` testpoint triggers `gen1`'s production rule. On being triggered, a production rule starts running. In `gen1`'s case, the rule always starts running right after being triggered, meaning that the waiting time (start time - trigger time) is always zero for this rule. After starting, a production rule runs for a certain amount of time (the execution time), and then produces timestamps specified by the output action. For block `gen1`, the production rule produces a timestamp on `gen1.y0`. The execution time for a block's production rule is calculated by subtracting rule start from rule finish times. For the first time `gen1`'s production rule ran, we find that the execution time was (5μs-0μs=) 5μs.

Figure 4.5 presents a `gen1` timeline which shows both X-Sim timing traces as well as X-Eval block production rule timings. The top two timelines show the X-Sim timing traces from trace files `top_gen1_src_tpt.ts` and `top_gen1_y0_out.ts`. These correspond to the times that were shown in Figure 3.19. The bottom three timelines represent the X-Eval production rule timings for the `gen1` block. The first timeline represents the time the production rule was triggered. For the `gen1` block, the trigger time corresponds directly to the time recorded by the `gen1.src` testpoint. The second timeline represents the time the production rule start running. For the `gen1` block, this also corresponds directly to a `gen1.src` timestamp. The third timeline represents the time the production rule finished running. A time on this timeline corresponds directly to a timestamp in the `top_gen1_y0_out.ts`

The three times the production rule ran are shown with different symbols in the figure. The first run is represented by a + symbol, the second run by a $\times$ symbol, and the third run by an $*$ symbol. Subtracting the production rule start time from the production rule trigger time gives production

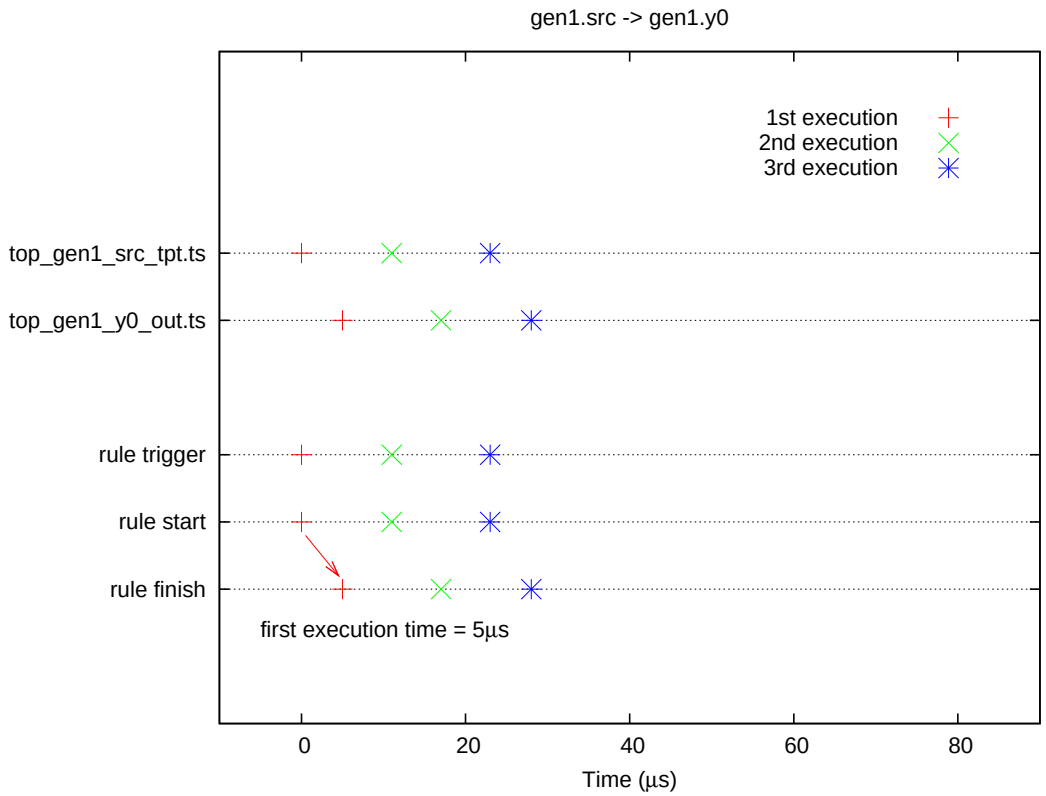


Figure 4.5: Timeline for block `gen1`

rule waiting times that are zero for each run. Subtracting the production rule finish time from the production rule start time gives the following production rule execution times for the `gen1` block: 5μs, 6μs, and 5μs. The average execution time is 5.3μs. The execution times for block `gen1` correspond directly to the times calculated by subtracting `gen1_src_tpt.ts` timestamps from `gen1_y0_out.ts` timestamps.

Block `gen2`, which operates similarly to block `gen1`, has the following production rule:

`gen2.src` $\rightarrow$ `gen2.y0`

In this case, a timestamp on testpoint `gen2.src` triggers block `gen2` to produce a timestamp on output port `gen2.y0`. The timeline for block `gen2` is given in Figure 4.6. The execution times for each production rule run for this block are all 5μs, so the average execution time is also 5μs. The production rule waiting times are once again zero for each run.

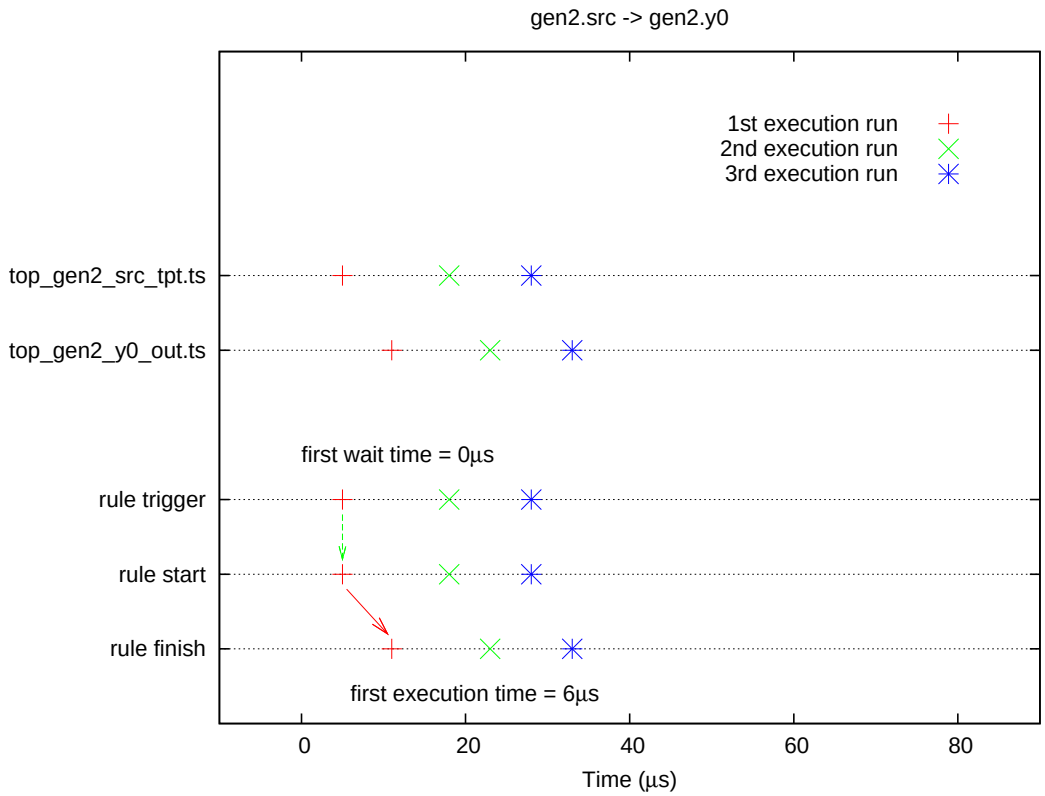


Figure 4.6: Timeline for block `gen2`

For both blocks `gen1` and `gen2`, the production rule is a simple one-to-one rule. One timestamp in a testpoint file triggers the production rule to produce one timestamp in an output port timestamp file. Block `sum` provides a more interesting case. This block takes in a value on each of its two input ports `sum.x0` and `sum.x1` , adds them, and outputs the result on its output port `sum.y0`. The production rule for the `sum` block is given by:

`sum.x0 and sum.x1` → `sum.y0`

The input condition for this rule is the `and`'ed combination, or *conjunction*, of the two input ports `sum.x0` and `sum.x1`. This means that the production rule will be triggered by the combination of one timestamp on `sum.x0` *and* one timestamp on `sum.x1`. When both of these input values become available (i.e. at the latter of the two `avl.ts` timestamps) the production rule is triggered. For example, the first value on port `sum.x0` became available at simulation time 10μs, and the first value on port `sum.x1` became available at 16μs. Corresponding to the latter of these two times, the

production rule was triggered for the first time at 16μs. The time the production rule starts running corresponds to the time when all triggering values have been input by the block (i.e. the latter of the two `in.ts` timestamps). The latest triggering value (in this instance, the value on port `sum.x1`) was input at time 16μs, so this is the start time for the first time the production rule ran.

The output action for the rule is simply the output port `sum.y0`. Thus, corresponding to the first timestamp in the output timestamp file for port `sum.y0`, the first execution run for the production rule finished at 17μs. The difference of the rule start and finish times, 1μs, gives the execution time for the first production rule run. All three execution runs for block `sum` actually took 1μs, so the average execution time for the block is also 1μs. These and subsequent timestamps and production rule runs are shown in Figure 4.7.

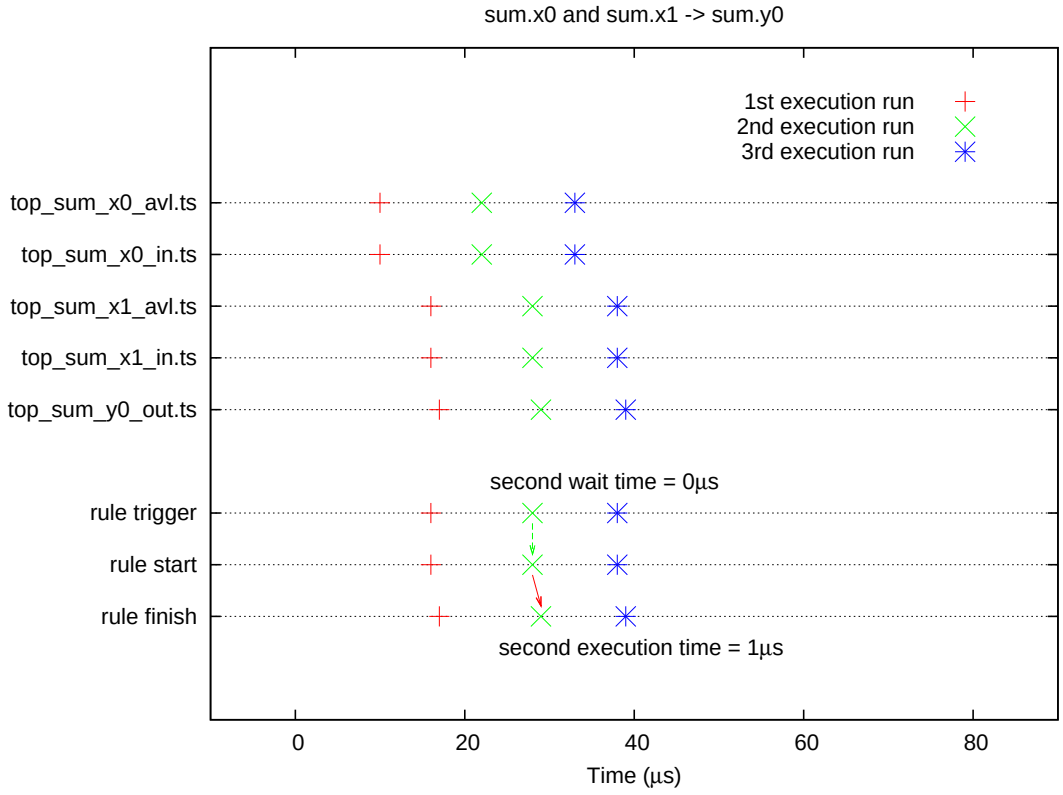


Figure 4.7: Timeline for block `sum`

The `half` block takes in a value on its input port, halves it, and outputs it on its output port. The production rule for this block is:

`half.x0` → `half.y0`

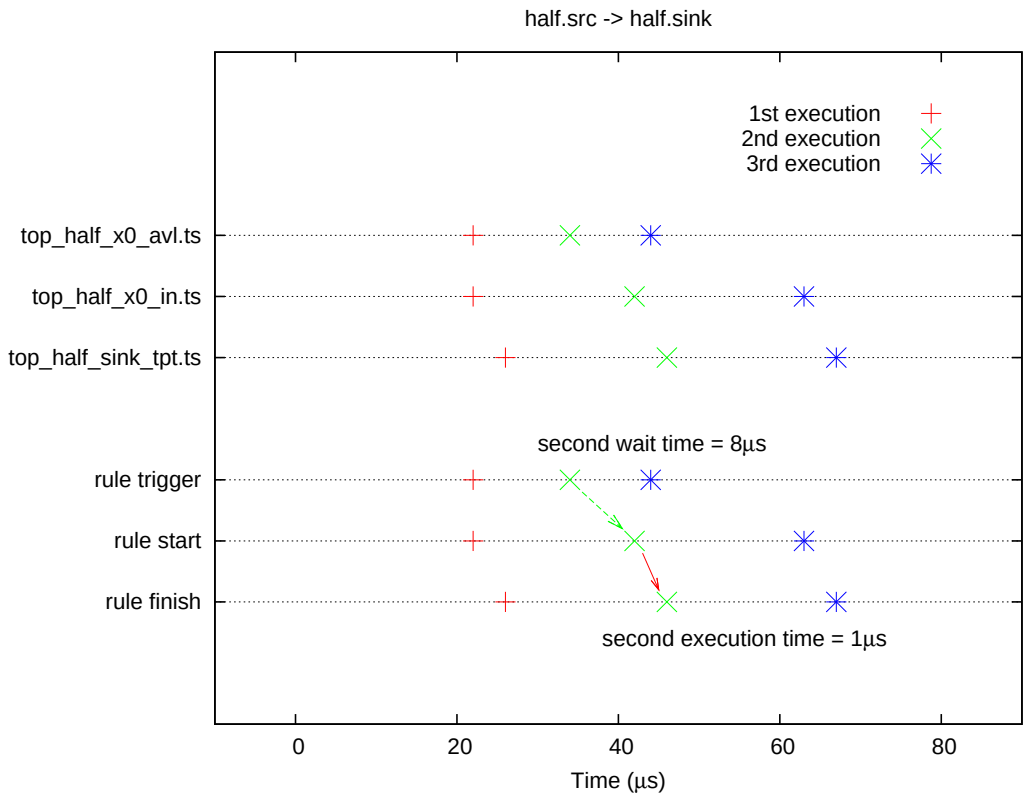


Figure 4.8: Timeline for block `half`

The timestamp and production rule timeline for block `half` is given in Figure 4.8. The first value arrived at port `half.x0` at time 22μs. The `half` block started processing that value right away, and finished processing it at time 26μs, passing it on to the `store` block. For the `half` block, the next value arrived at its input port at time 34μs, but processing of this value could only start at time 42μs, indicating that this value had a waiting time of (42-34=) 8μs. The reason that the `half` could not start processing the second value as soon as it was available was that the `store` block, which is mapped to the same CR, was busy processing the first value. The waiting times for the three values for block `half` are 0μs, 8μs, and 19μs, while the three execution times are all 4μs.

For the `store` block, the production rule for block `store` is given by:

`store.src` → `store.sink`

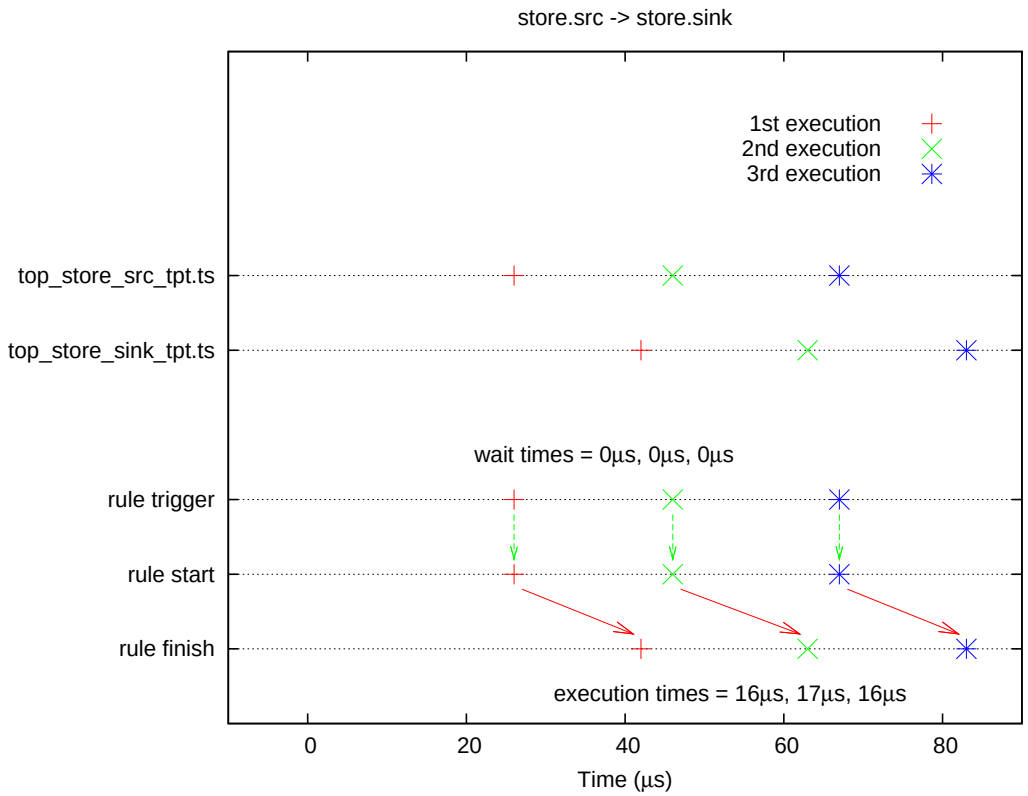


Figure 4.9: Timeline for block `store`

The timeline for this block is given in Figure 4.9. The waiting times for this block are all zero, indicating that the `store` block was able to process data as soon as it was provided by the `half` block. The reason this happens is that when data is pushed onto port `half.y0`, it results in the `store` block's push function being called. Refer to [20] for more details of how binaries generated from X Language applications run. The execution times for the `store` block are 16μs, 17μs, and 16μs.

X-Eval Block	Mean Wait Time	Mean Exec. Time
`gen1`	0μs	5.33μs
`gen2`	0μs	5.33μs
`sum`	0μs	1μs
`half`	9μs	4μs
`store`	0μs	16.33μs

Table 4.1: Summary of performance results for `test1`

Running X-Eval on `test1` with the given production rules generates all the individual waiting and execution times that were presented in this section, and also presented in a table format in the previous chapter in Table 3.1. X-Eval also gives a short summary of the analysis results, presenting the figures given in Table 4.1. The fact that data entering `proc[2]` experiences waiting time before being processed indicates that the performance of `proc[2]` is a limiting factor in `test1`'s overall performance. A possible optimization would be to pipeline the `half` and `store` blocks over separate processors. This re-mapping will be explored in a later chapter. The next section, meanwhile, provides an explanation of production rules that is not specific to the `test1` example.

4.3 Production Rules in X-Eval

4.3.1 An Informal Approach

Production rules can be used in X-Eval to capture the relationship between X-Sim traces for an otherwise black-box X Language block. Consider the simple case where a single block is mapped to a CR. This is shown in Figure 4.10. At the end of an X-Sim simulation, trace files are available by default for each of the two input ports as well as for the output port. X-Eval needs to analyze the traces to generate a history of execution and waiting times for the block.

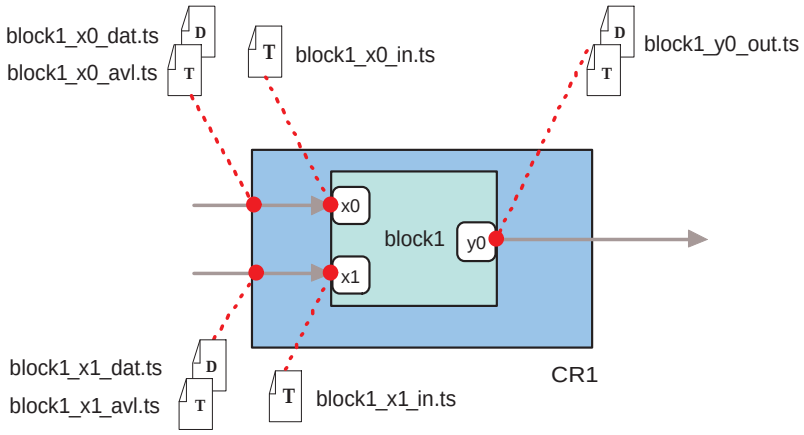


Figure 4.10: Simple X Language block mapped to a CR

Using production rules, a user can specify how X-Eval should analyze the traces from the input and output ports of a block. For instance, the above block could be a ‘sum’ block that consumes two inputs and produces one output. It could also just as easily be a ‘pass through’ block that consumes an input on either input and produces an output.

Production rules can be used to distinguish between these two relations of input to output traces. If `block1` is a ‘sum’ block, its production rule can be represented by:

```
block1.x0 and block1.x1 → block1.y0
```

According to this specification, `block1` consumes one piece of data on both of its input ports `x0` and `x1` to produce a single piece of data on its output port `y0`. If `block1` was a ‘pass through’ block instead of a ‘sum’ block, its production rule would instead be:

```
block1.x0 or block1.x1 → block1.y0
```

This specifies that `block1` consumes a piece of data on either one of its input ports to produce a piece of data on its output port.

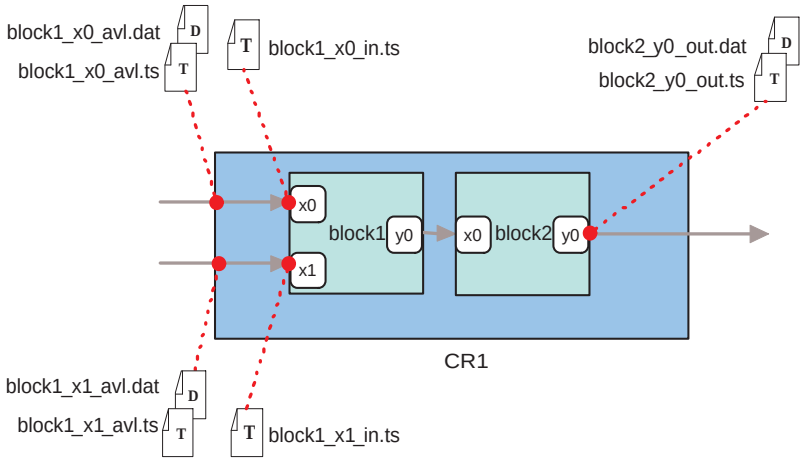


Figure 4.11: Two simple X Language blocks mapped to a CR

Consider the block mapping shown in Figure 4.11, where two X Language blocks are mapped to `CR1`, and traces are only collected at the default points indicated in the figure. In this case, it is

possible to make X-Eval treat the combination of the two X Language blocks as a single X-Eval block. If `block1` is a 'sum' block while `block2` is a simple 'pass through' block, the production rule for the X-Eval block formed by the combination of the two X Language blocks is given by:

`block1.x0 and block1.x1 → block2.y0`

If `block1` is a 'pass through' block instead, then the combined production rule is:

`block1.x0 or block1.x1 → block2.y0`

The flexibility in defining an X-Eval block allows even multiple CRs to be grouped together in a single X-Eval block. For example, consider again example `test1` that was given in Figure 4.1. Say that we want to find latency metrics for when a pair of numbers was first generated to when the average of the two numbers was finally calculated. The time that `proc[1]` first started generating the two numbers is give by the timestamp recorded by `gen1.src`, and the time that `proc[2]` finally finished storing the result is given by the timestamp recorded by `store.sink`. An end-to-end latency analysis of the `test1` application can thus be done by the production rule:

`gen1.src → store.sink`

The entire `test1` can be considered to be a single X-Eval block, with the above production rule defined for it. The execution time for a run of this X-Eval block will be calculated by subtracting a `store.sink` timestamp from the corresponding `gen1.src` timestamp. Figure 4.15 in the next section will provide additional details on treating the entire `test1` application as a single X-Eval block with a single production rule.

Users are also given the flexibility to define multiple production rules for a single X-Eval block. For example, the user can choose to treat the `proc[1]` CR from the `test1` example as a single combined X-Eval block with two production rules, one each for `gen1` and `gen2`:

```
gen1.src → gen1.y0
gen2.src → gen2.y0
```

Another equally valid way for the user to specify `proc[1]`'s operation is to combine the two production rules into one:

```
gen1.src or gen2.src → gen1.y0 or gen2.y0
```

A lot of flexibility is provided to users so that they can tailor X-Eval to analyze their application in the way that makes most sense to them. They can choose to do analysis per X Language block, per CR, or for a combination of block groupings. The production rules simply provide a method to express the semantic operation of otherwise black box X Language blocks. It is up to the user to describe production rules that give meaningful and useful waiting time and execution time statistics. The next section gives a formal description of X-Eval production rules, as well describing how production rules are used internally by X-Eval to analyze an application. The final section, Section 4.4, gives a tutorial with a concrete example of production rules and X-Eval being used in analyzing the `test1` application.

4.3.2 A Formal Approach

Consider Figure 4.12, where a single X Language block `blk` is mapped to a CR. This is the *X-Eval block*, *b*, that we will use as an illustrative example in this section.

An X-Eval block can have multiple input and multiple output *ports*. An X-Eval block port corresponds either to an X Language port, or to an X Language testpoint. Let p represent an arbitrary X-Eval block input or output port. Then $e_{p,i}$ represents the ith *event* on port p (e.g., $e_{blk.x2,2}$ is the second event on input port `blk.x2`). Events are either *input events* or *output events*. An input event corresponds to a piece of data arriving at and entering an input X-Eval block port, and has associated arrival and input times. An output event corresponds to a piece of data exiting an output

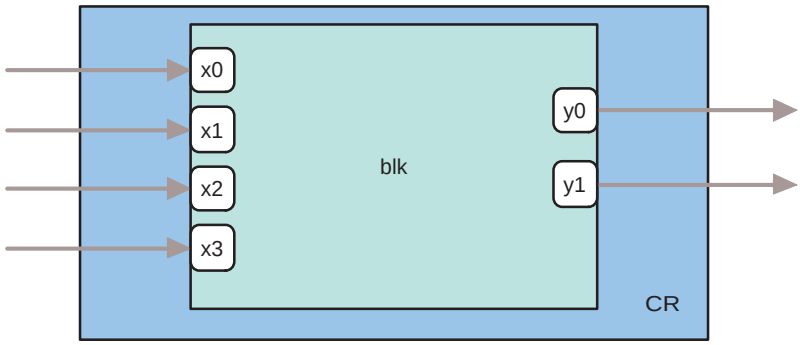


Figure 4.12: X-Eval block *b*

X-Eval block port, and has an associated output time. If *p* is an input X-Eval block port, then $e_{p,i}$ is an input event. Similarly, if *p* is an output X-Eval block port instead, then $e_{p,i}$ is an output event.

The set of all input ports for an X-Eval block *b* is represented by $P_{b,in}$, and the set of all output ports is represented by $P_{b,out}$. For X-Eval block *b*,

$P_{b,in}$ = { `blk.x0, blk.x1, blk.x2, blk.x3` }

$P_{b,out}$ = { `blk.y0, blk.y1` }

For brevity in this section, whenever a port term such as `x0` is used without an associated block name, assume that it refers to block `blk`. Port `x0` thus implicitly refers to port `blk.x0`.

A sequence of terms will now be introduced that will allow us to give a formal definition of a production rule. A *port multiple* np represents the pairing of the port *p* and non-zero natural number *n*. For example, `2x0` represents the pairing of the number 2 and the port `x0`. The term *p* by itself can be used implicitly to mean the port multiple $1p$. A port multiple np is said to be *satisfied* by a set of events when that set of events consists of *n* events on the port *p*.

One or more port multiples combined using `and`s forms a *conjunction*, such as:

```
2x0 and x2
```

which consists of the two port multiples `2x0` and `x2` `and`ed together. A conjunction is said to be *satisfied* by a set of events when that set of events satisfies **all** the port multiples in that conjunction. For example, the above conjunction is satisfied by 2 events on port `x0` and 1 event on port `x2`. Note that a port multiple by itself can be considered a conjunction with only a single port multiple in it (e.g. `x1` can be considered a conjunction).

A set of conjunctions combined using `or`s forms a *disjunction*, such as:

```
(2x0 and x2) or x1
```

The two conjunctions in the above disjunction are 1) `2x0 and x2`, and 2) `x1`. A disjunction is said to be *satisfied* by a set of events when that set of events satisfies **any** of the conjunctions in that disjunction. For example, the above disjunction is satisfied by either 1) 2 events on `x0` and 1 event on `x2`, or 2) 1 event on x1. Note that here too, a conjunction by itself can be considered a disjunction with a single conjunction in it. Thus, `x1` can be considered to be a port, a port multiple (`1x1`), a conjunction (consisting of a single port multiple), and a disjunction (consisting of a single conjunction).

A restriction in defining conjunctions is that the ports must be either **all** input or **all** output ports. For example, the following conjunction is **illegal** because `x0` is an input port while `y0` is an output port:

```
2x0 and y0
```

A conjunction can either be an input conjunction with all input ports in it, or an output conjunction with all output ports in it. Similarly, a disjunction is also either an input disjunction with all input ports in it, or an output disjunction with all output ports. A further restriction on disjunctions is that one port can only appear in a single conjunction for a single disjunction. This restriction simplifies the task of matching X-Sim traces to the correct X-Eval production rule, as will be discussed in Section 4.3.3.

A *production rule* q consists of an input disjunction $d_{q,in}$ and an output disjunction $d_{q,out}$. The production rule is represented by the expression:

$$d_{q,in} \rightarrow d_{q,out},$$

An example of a production rule is:

production rule $q_{b,1}$: `( 2x0 and x2 ) or 2x1` $\rightarrow$ `2y0`

A production rule $q_{b,j}$ is fully represented by its X-Eval block b and its index j. For example, $q_{b,j}$ corresponds to the jth production rule of block b. Production rule $q_{b,j}$ is *triggered* when a set of events happens at X-Eval block b's input ports that *satisfies* the disjunction $d_{q_{b,j},in}$. A disjunction is satisfied by a set of events when *any* of its conjunctions is satisfied by the event set. A conjunction, in turn, is satisfied when *all* of its port multiples are satisfied, and a port multiple np is satisfied when the set of events includes exactly n input events on port p. When the first matching set of input events is found for a production rule q, they combine to form the first *input record*, $r_{q_{b,1},in,1}$. The lth set of events that triggered q combine to form the lth *input record*, $r_{q_{b,1},in,l}$. For example, consider production rule $q_{b,1}$ given previously for the X-Eval block shown in Figure 4.12, along with the following timeline of sequential input events.

input events : $e_{x0,1}$, $e_{x2,1}$, $e_{x1,1}$, $e_{x0,2}$, $e_{x1,2}$

Note that the sequential order of input events is the same as the sequential order of the corresponding `in.ts` timestamps. Event $e_{x0,1}$ happened before event $e_{x2,1}$, event $e_{x2,1}$ happened before event $e_{x1,1}$, and so on. An X-Eval block evaluates each input event in sequence until it has encountered a combination of events that triggers one of its production rules. The combination of input events makes up an input record. On triggering a production rule, the events in the input record are *consumed*, and cannot trigger that rule again. For the given sample sequence of input events, the first three events cannot trigger any production rule because they do not satisfy the input disjunction for any rule. When the fourth event $e_{x0,2}$ is considered in addition to the first three, a combination can be formed that satisfies the input disjunction for production rule $q_{b,1}$. The input record formed is:

$r_{q_{b,1},in,1} = (e_{x0,1}, e_{x2,1}, e_{x0,2})$

This combination of events satisfies the first conjunction in $q_{b,1}$'s input disjunction:

```
2x0 and x2
```

With the triggering of the production rule, these events are consumed and cannot be used to trigger any rule again. However, the event $e_{x1,1}$ has not been consumed and is still available to form a rule triggering combination. When the fifth input event $e_{x1,2}$ is considered next by the block b, it forms a valid combination along with $e_{x1,1}$ to trigger production rule $q_{b,1}$. The input record formed in this case is:

$r_{q_{b,1},in,2} = (e_{x1,1}, e_{x1,2})$

This combination satisfies the second conjunction in $q_{b,1}$'s input disjunction, `2x1`.

When triggered, a production rule q produces a set of events satisfying $d_{q,out}$, by satisfying *any* of the conjunctions in $d_{q,out}$. This set of events forms an *output record*, $r_{q,out,l}$. The first triggering of the production rule $q_{b,1}$ from our example sequence of input events produces the following output record:

$r_{q_{b,1},output,1} = (e_{y0,1}, e_{y0,2})$

The second triggering produces the following output record:

$r_{q_{b,1},output,2} = (e_{y0,3}, e_{y0,4})$

These output records show up on the output ports of the X-Eval block as the following sequence of output events:

output events : $e_{y0,1}$, $e_{y0,2}$, $e_{y0,3}$, $e_{y0,4}$

The lth triggering of a rule q corresponds to one input record $r_{q,input,l}$ and one output record $r_{q,output,l}$. The execution time $t_{q,l}$ for the production rule run can be calculated from:

$$t_{q,l} = t_{q,output,l} - t_{q,input,l}$$

$t_{q,input,l}$, the lth record input timestamp for rule q, is the *latest* timestamp from all the events in $r_{q,input,l}$. Similarly, the lth record output timestamp $t_{q,output,l}$ is the *latest* timestamp from all the events in $r_{q,output,l}$. Once all the input and output records for a production rule are determined, the distribution of its execution times can thus easily be calculated. The crucial problem of how to determine which events make up each input and output record for each production rule is tackled in the next section.

4.3.3 Determining Production Rule Records from X-Sim Traces

An X-Eval block b can have multiple production rules, with the set of production rules represented by Q_b. The set of input and output ports are represented by $P_{b,in}$ and $P_{b,out}$ respectively. The sequentially ordered set of all events for port p is represented by E_p. The ordered set of all events for *all* input ports for block b is represented by $E_{b,in}$. $E_{b,out}$ represents the corresponding set for b's output ports. The ordered sets of input and output records for production rule q are represented by $R_{q,in}$ and $R_{q,out}$ respectively. The ordered sets combining input and output records for *all* production rule for block b are $R_{b,in}$ and $R_{b,out}$. Production rules have the property that they *preserve order*. This means that the output events for the lth triggering of a rule q cannot happen sequentially *after* any output events for q's $(l+1)$th triggering. Production rules run in the order they were triggered in, on a FIFO (First In First Out) basis.

Given Q_b, $P_{b,in}$, $P_{b,out}$, $E_{b,in}$, $E_{b,out}$, and the property of order preservation, we would like to be able to deterministically find $R_{b,in}$ and $R_{b,out}$.

Recall that in the previous section, we specified that two disjunctions can never have a port (input or output) in common. This means that any two rules on a block always cover **exclusive** sets of ports. A rule q can only consume events on ports covered by its input disjunction, and only produce events on ports covered by its output disjunction. No other rules can consume events on rule q's input ports, nor produce events on q's output ports. This leads to the important conclusion that when formulating input and output records for a block b, **we can consider production rule q and its covered input and output ports *in isolation* without considering the operation of other rules on the block**.

Now, also consider the property that each conjunction within any disjunction also has an exclusive set of ports (i.e., two conjunctions within a disjunction cannot share a port). This means that we can consider each conjunction of a rule's disjunction in isolation as well. For example, consider the rule $q_{b,1}$ presented before and reproduced below.

production rule $q_{b,1}$: `( 2x0 and x2 ) or 2x1` $\rightarrow$ `2y0`

When assigning input records for rule $q_{b,1}$, we need to look at two conjunctions. The first conjunction covers ports `x0` and x2, while the second conjunction covers just the port `x1`. Conforming to the port exclusivity principle, these conjunctions do not have any ports in common. When creating input records from input events for rule $q_{b,1}$, we can thus look at the first conjunction first and create all the records corresponding to it by just looking at the events on ports `x0` and `x1`. After all the input records for the first conjunction have been created, we can then consider the second conjunction and create all the input records for it, only looking at events on port `x1`. Finally, all output records for the single conjunction for rule $q_{b,1}$ can be created by looking at the events on port `y0`. **When creating records, a single conjunction is considered at a time.**

A record always consists of the *first* input events that correspond to a particular conjunction. For an input conjunction, the first input events to satisfy it form an input record and trigger a production rule. For an output conjunction, all the output events from a rule being triggered are guaranteed to be produced before any output events produced by a later triggering because of the principle of order preservation.

```
foreach (rule q in block b){
    foreach (disjunction d in q){
        foreach (conjunction c in d){

            while (events left on c's covered ports){
                create empty record r
                foreach(portmultiple np in c){
                    add n events on p to r
                }
                store r
            }

        }
    }
}
```

Figure 4.13: Algorithm for X-Eval Record Generation Code

Given our discussion of how input and output records can be generated by considering production rule conjunctions in seclusion, the pseudo-code given in Figure 4.13 shows the algorithm implemented inside X-Eval to create records for an X-Eval block b. By describing blocks using production rules, X-Eval can generate distributions of waiting and execution times for each "run" of the block. This allows users to get an idea of performance of individual blocks inside the application. Although most types of application blocks can be modeled and analyzed using production rules, there are some types of blocks that cannot directly be modeled by X-Eval. They are described in the next section.

4.3.4 Restrictions on X-Eval Modeling

Block and production rule definitions given by the user tell X-Eval how to group together X-Sim traces to form input and output records, and from that generate waiting and execution times. In this section, we will look at what the main requirements and restrictions are in defining production rules, and what types of blocks *cannot* directly be modeled by X-Eval. A list of the restrictions when specifying production rules is given below:

- disjunctions and conjunctions must be non-null

- blocks must preserve execution order of rules
- a port can only appear once inside a rule disjunction
- a port cannot be used in more than one rule specification

The first requirement is that disjunctions and conjunctions must be non-null. This means that rules cannot be triggered by a null set of input events, nor can they produce a null set of output events when triggered. **This prevents, for example, filter blocks from being modeled directly (without added testpoints) as an X-Eval block.** Say a block `blk1` (no corresponding figure) takes in an input value on its input port `x0`, and then depending on the value either passes on the value onto its output port `y0` or does not produce any data on its output port. One might want to characterize this production rule as:

```
blk1.x0 → blk1.y0 or ( )
```

This is not a valid specification because the second output conjunction is null. The problem is that X-Eval can figure out when the rule triggers, but has no X-Sim trace that can let it realize when the rule executed but did not produce any data on the `y0` output port. To be able to model this block as X-Eval block with production rules, it is necessary for the block to produce a trace when it decides even to *not* produce data on its output port. This can be achieved by adding a testpoint that is timestamped whenever `blk1` filters out the `x0` value. If the testpoint is named `filtered`, then the correct production rule definition would be:

```
blk1.x0 → blk1.y0 or blk.filtered
```

Similarly if the block `blk1` were a data generation block that only had a single port, the output port `y0`, then one may want to characterize its production rule as:

```
( ) → blk1.y0
```

This definition is not valid because of the null input disjunction, and can be rectified by adding a testpoint (e.g., `src`, which is timestamped whenever the block starts generating a number). The corrected production rule definition would then be:

```
blk1.src → blk1.y0
```

This method was seen for the `gen1` and `gen2` blocks in our `test1` example. A similar solution exists for data sink blocks, like the `store` block in the `test1` example.

The second restriction for production rules is that production rules preserve execution order. Suppose a rule has two input records, r_1 and r_2, and two output records r_3 and r_4. The corresponding times are t_1, t_2, t_3, and t_4, where $t_1<t_2$ and $t_3<t_4$. The order preservation principle guarantees that r_1 triggered the rule to produce r_3, and r_2 triggered the rule to produce r_4. Essentially it allows sequentially corresponding input and output records to be paired together to calculate execution times.

This restriction can be a problem when multiple X Language blocks are combined together to form an X-Eval block. For example, consider the block `blk2` with one input port `x0` and two output ports `y0` and `y1`, and the internal structure shown by Figure 4.14.

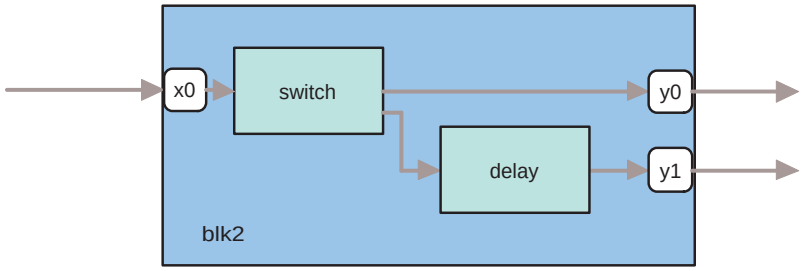


Figure 4.14: An X-Eval block with a long and short production rule execution path

The production rule for `blk2` is given by:

```
blk2.x0 → blk2.y0 or blk2.y1
```

Data entering on port `blk2.x0` is switched to either the first or second output ports of the internal switch block. Data that travels through the second path must go through a second internal block before being output on port `blk2.y1`, and will usually take longer to be output than data that travels through the first path and is output on port `blk2.y0`.

Consider the following scenario. Data (corresponding to record r_1) enters `blk2.x0`, and is switched along the second (bottom) path. As the first piece of data is being processed by the delay block, another piece of data (record r_2) arrives at the switch block. The internal switch block and delay block can run in parallel, for example if they are mapped to separate CRs, or if they are mapped to a CR that can run blocks in parallel (e.g., an FPGA). As the first piece of data is being processed by the delay block, the second piece of data is switched along the first (top) path. It exits `blk2.y0` (forming r_3), while the first piece of data is *still* being processed by the delay block. Finally, the first piece of data finishes being processed by the delay block, and exits `blk2` from port `y1` (forming r_4).

When X-Eval reads the X-Sim traces for this block (ports `x0`, `y0` and `y1`), it will associate r_3 with r_1, and r_4 with r_2. This is the expected association according to the order preservation principle. However, this is incorrect given the internal structure of block, and our given case scenario.

A solution to this problem is for the user to split `bl2k` into two X-Eval blocks, with a testpoint (e.g. a testpoint called `internal`) at the edge between the switch and delay) components. The production rules for the two different X-Eval blocks are then given by:

```
blk2.x0 → blk2.y0 or blk2.internal
blk2.internal → blk2.y1
```

An alternative solution is to just be aware of this problem, but accept that output records may get associated with the wrong input records. The number of input and output records are still equal so the right number of production rule runs will still be found. The individual execution times calculated will be off, but the *average* (and total) execution time will stay the same because:

$(t_3\text{-}t_1) + (t_4\text{-}t_2) = (t_3\text{-}t_2) + (t_4\text{-}t_1)$

The third restriction for a production rule is that a port can only appear once inside a disjunction. Consider the case if block `blk3` has a single input port `x0` and a single output port `y0` (no corresponding figure shown). The function the block provides is to input a piece of data on port `x0`, and then produce either one or two pieces of data on port `y0`. One may want to specify the production rule for this block as:

```
blk3.x0 → blk3.y0 or 2blk3.y0
```

However, this violates the third restriction, because port `y0` appears twice in the output disjunction. Consider the case where two input pieces of data appear on port `x0`, and `blk` produces three input pieces of data on port `y0`. There is no way for X-Eval to be able to figure out whether the second output event was produced by the first rule execution or the second rule execution. This problem, too, can be solved by adding a testpoint `done` that is timestamped by the block whenever it is done executing, and produces one or two outputs on `y0`. The new production rule definition would then simply be:

```
blk3.x0 → blk3.done
```

The final restriction on production rules is that a port cannot be used in more than one rule specification. For the last block we considered, with either one or two outputs being produced for every input, another way the user may have chosen to describe the operation of the block might have been by specifying two rules:

```
blk3.x0 → blk3.y0
blk3.x0 → 2blk3.y0
```

This presents the same problem as before, and can also be solved the same way as before by adding the testpoint `blk3.done` and defining a single production rule for the block.

In conclusion, if production rules are specified and perform according to the definitions given in the previous section and restrictions given in this one, then input and output record lists can be created for each rule. Using these record lists, performance numbers can be calculated for each X-Eval block.

4.3.5 Generating Performance Metrics and Models

Once X-Eval blocks and production rules have been defined, performance metrics can be calculated and analytic performance models can be created. First, the record generation algorithm presented earlier is used to translate event lists into record lists for each production rule. The model for X-Eval blocks is that one block consists of a set of input and output ports and a set of production rules. When a combination of input events occurs that triggers a production rule, those input events form an input record for that rule. On being triggered, a rule produces output events that satisfy its output disjunction. These events form an output record. After a complete simulation run and X-Eval analysis, a production rule has the same number of input and output records.

The difference between the time an output record is produced by a rule and the time that the rule started running is defined as that rule's execution time. The lth execution time for rule q, $t_{q,l}$, is thus given by the expression:

$$t_{q,l} = t_{q,l,out} - t_{q,l,in}$$

where $t_{q,l,out}$ is the time the last event of output record $r_{q,l,out}$ was produced, and $t_{q,l,in}$ is the time the last event of input record $r_{q,l,in}$ was consumed. A complete history of the times it took to execute a rule q can thus easily be calculated once a complete input and output record trace is generated for the production rule.

This trace of execution times can be used as a substitute for the actual simulation of a CR, when a re-simulation of an entire application is done. Similarly, any variety of statistical results can be calculated (e.g. mean, standard deviation, etc.) to parametrize any desired statistical model. The use of traces and models a simulation speedup technique is described in more detail in the next

chapter. Before moving on to the next chapter, however, we provide a tutorial on how to use X-Eval to analyze an application.

4.4 Tutorial for Analyzing an Application using X-Eval

Suppose the user would like to get end-to-end latency times for the example `test1`. Figure 4.15 shows a minimal *semantics* file (.smx file) that causes X-Eval to calculate the required values.

```
1 block top
2     port gen1.src in_port
3         event app_gen1_src_tpt.ts avl_event
4         event app_gen1_src_tpt.ts in_event
5     port store.sink out_port
6         event app_store_sink_tpt.ts out_event
7     rule gen1.src −> store.sink
```

Figure 4.15: Basic semantics file specification for example `test1`

A semantics file declares all X-Eval blocks, all the ports and rules for these blocks, and a list of X-Sim trace files where timestamps can be found for events that happened on the given ports. For example, in the example semantics file, line 1 declares an X-Eval block called `top`. The next line declares an input port on this block called `gen1.src`. This corresponds to the testpoint `src` on the `gen1` X Language block. X-Sim testpoints can be declared as X-Eval block ports. However, X-Eval block ports can be either input port or output ports, so testpoints must always be declared as one type or the other when used as ports. Note that here, the testpoint `gen1.src` is used as an input port. In the semantics file, all the X-Sim trace files associated with the X-Eval block port are listed, along with what type of events the timestamps should be treated as. In the case of a testpoint being treated as an input port, the timestamps collected are treated as both availability and input times. Line 5 in the semantics file shows another testpoint, `store.out`, being used as an output X-Eval block port. The X-Sim timestamp file for that testpoint is interpreted to provided output times for events on that output port, as specified on line 6. Finally, line 7 defines the production rule for the single X-Eval block as a simple single-input single-output trace relation.

X-Eval can be invoked by following command line call:

```
xeval -x test1.smx
```

A summary of results is printed to the output, and can be piped into a results file if desired. This summary of results gives the average execution and waiting times for each production rule. For our simple semantics file, the output summary given by running X-Eval is shown in Figure 4.16.

Block	**Rule**	**Mean Wait**	**Mean Exec**
top	rule1	0μs	51.33μs

Figure 4.16: Results summary for end-to-end analysis of `test1`

In addition to the results summary, more detailed results files are created for each production rule that record details from individual runs of the rule. The three files created for the production rule given in our simple .smx file are `top_rule1_index.ts`, `top_rule1_exec.ts`, `top_rule1_wait.ts`. The first file, `top_rule1_index.ts`, records which conjunction in the output action a production rule actually output to for each execution run. This value is used in trace-driven simulation substitution, and its use will be explained in the next chapter. The second file, `top_rule1_exec.ts`, records the execution time for each time the production rule ran. Finally, the `top_rule1_wait.ts` file records the waiting time for each time the rule ran. The index file records values in 8-bit unsigneds, while the other two files record values in 64-bit unsigneds. Execution and waiting times are recorded in nanoseconds for high precision.

For our example, the three values in the index file would all be 1 because there is only one output conjunction that produces output according to the production rule. The three execution times, representing the end-to-end application latency, would be 42μs, 52μs, and 60μs. The three waiting times, which are not too useful for an end-to-end application analysis, are all zero.

The average latency to generate one output for the entire application is shown by X-Eval to be 51.33μs. As we saw in the last chapter, the total time to run the application and generate three outputs was 83μs. If we had zero parallelism, the time to generate three outputs would have been 154μs. Parallelism in the application implementation has allowed the process to be sped up from taking 154μs to only 83μs, a speed up of roughly twice when generating three outputs.

```
1  block gen1
2      port gen1.src in_port
3          event top_gen1_src_tpt.ts avl_event
4          event top_gen1_src_tpt.ts in_event *
5      port gen1.y0 out_port
6          event top_gen1_y0_out.ts out_event *
7      rule gen1.src -> gen1.y0
8  block gen2
9      port gen2.src in_port
10         event top_gen2_src_tpt.ts avl_event
11         event top_gen2_src_tpt.ts in_event *
12     port gen2.y0 out_port
13         event top_gen2_y0_out.ts out_event *
14     rule gen2.src -> gen2.y0
15 block sum
16     port sum.x0 in_port
17         event top_sum_x0_avl.ts avl_event *
18         event top_sum_x0_in.ts in_event *
19     port sum.x1 in_port
20         event top_sum_x1_avl.ts avl_event *
21         event top_sum_x1_avl.ts in_event *
22     port sum.y0 out_port
23         event top_sum_y0_out.ts out_event *
24     rule sum.x0 and sum.x1 -> sum.y0
25 block proc2
26     port half.x0 in_port
27         event top_half_x0_avl.ts avl_event *
28         event top_half_x0_in.ts in_event *
29     port half.sink out_port
30         event top_half_sink_tpt.ts out_event *
31     port store.src in_port
32         event top_store_src_tpt.ts avl_event
33         event top_store_src_tpt.ts in_event *
34     port store.sink out_port
35         event top_store_sink_tpt.ts out_event *
36     rule half.x0 -> store.sink
```

Figure 4.17: Detailed semantics file for example `test1`

Block	Rule	Mean Wait	Mean Exec
gen1	rule1	0μs	5.33μs
gen2	rule1	0μs	5.33μs
sum	rule1	0μs	1μs
half	rule1	9μs	4μs
store	rule1	0μs	16.33μs

Figure 4.18: Results summary for `test1` analysis

A more informative analysis of application performance can be done by breaking up the application into more X-Eval blocks, as shown in Figure 4.17. In this semantics file, each X Language block corresponds to an X-Eval block. An asterisk at the end of a line that lists an X-Sim trace file tells X-Eval that events from that file should be added to the auto-generated timeline plot.

Running X-Eval on this more complete semantics file gives the results shown in Figure 4.18. We can see from these results that data has to wait an average of 9μs before it can be processed by the `half` block. Also, the `store` block is exhibiting an average execution time of 16.33μs which is relatively high compared to other blocks' execution times. Given that the `half` and `store` block share the `proc[2]` CR, a possible next step would be to map the `store` block to the unused `proc[3]` CR, and re-simulate and re-analyze the application. The next chapter, Simulation Speedup, shows techniques for speeding up re-simulations of an X Language application.

Chapter 5

Simulation Speedup Techniques

After an application has been written and debugged using X-Com and X-Sim, the next step is to optimize its performance. Optimizing the performance of the application consists of a cycle of re-mapping the algorithm, re-simulating the new mapping, and re-analyzing the simulation traces, as was shown in Figure 1.5 in Chapter 1.

By speeding up the re-simulations of applications, new mappings can be tried out faster, shortening the development and performance optimization cycle. This advantage will become even more important when an automated X-Opt tool is developed, and the performance optimization cycle is able to run multiple iterations in a short amount of time.

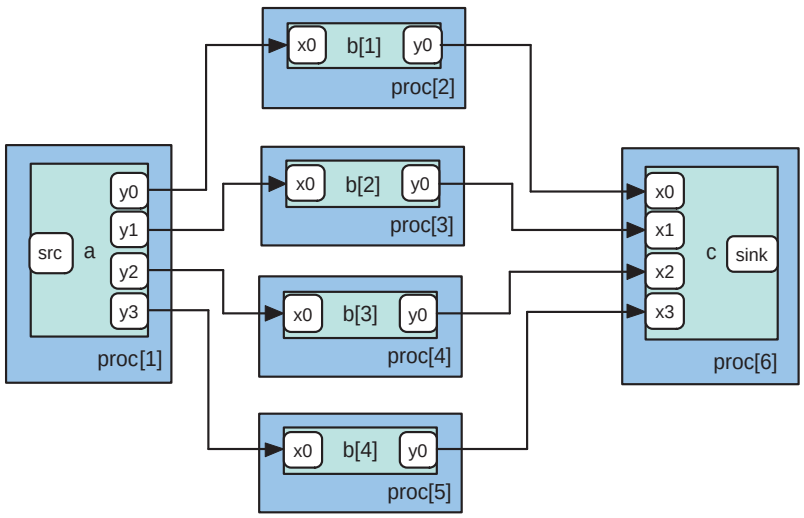


Figure 5.1: First mapping of example application `test2`

Consider the example application `test2` given in Figure 5.1. Each block in this application has a built-in delay of 1s per execution. Block `a` waits for 1s and then generates four numbers, one on each of its four outputs. It runs ten times, making the total number of values generated 40. Blocks `b[1]` through `b[4]` each wait 1s before passing through a data value from their input ports to their output ports. Block `c` waits 1s before inputting four data values, one from each of its input ports. Each block is mapped to its own processor CR in this mapping.

For our analysis and discussion, we can consider the communication delay to be negligible. This is because communication delay is on the order of microseconds, while processing time is on the order of seconds (i.e. a million times longer).

We will use the `test2` application to illustrate how simulation time can be reduced. There are two primary methods of speeding up X-Sim application simulations. The first technique involves running multiple federate simulations in parallel. The second technique involves substituting federate CR simulations with either trace-based models or with analytic distribution-based models. These techniques are described in detail in the following sections.

5.1 Simulating CRs in Parallel

As we saw in Chapter 3, X-Dep can be used to create an X-Sim Makefile that coordinates the running of different federate simulators. A federate simulator can be either the simulation of a CR (e.g., a processor or FPGA simulation), or the simulation of an edge mapped to an IR (using `xmodel`). The Makefile generated by running X-Dep on the `test2` application is shown in Figure 5.2.

Let us consider what happens when the simulation described by this Makefile is run on a single-processor core machine. The first simulation to be run for the given mapping is the CR simulation of `proc[1]`. Nothing else can run before this simulation, because everything else in the application mapping is downstream from `proc[1]`. Only block `a` is mapped to `proc[1]`. This block runs 10 times, with each run taking 1s. Thus, a core must run for 10s to simulate the complete operation of `proc[1]`.

```
1  simulate: proc_[1-6].perf
2      echo Simulation done.
3
4  proc_1_.perf:
5      proc_1_ >& proc_1_.out
6      echo Done > proc_1_.perf
7
8  proc_2_.perf: proc_1_.perf
9      xmodel -i top_a_y0_out.ts -1 freq=3.4e9 \
10            -o top_b_1__x0_avl.ts -2 freq=3.4e9
11     mv top_a_y0_out.ts top_b_1__x0_avl.dat
12     proc_2_ >& proc_2_.out
13     echo Done > proc_2_.perf
14
15 proc_3_.perf: proc_1_.perf
16     xmodel -i top_a_y1_out.ts -1 freq=3.4e9 \
17           -o top_b_2__x0_avl.ts -2 freq=3.4e9
18     mv top_a_y1_out.ts top_b_2__x0_avl.dat
19     proc_3_ >& proc_3_.out
20     echo Done > proc_3_.perf
21
22 proc_4_.perf: proc_1_.perf
23     #### ... similar to proc[2] and proc[3] ###
24
25 proc_5_.perf: proc_1_.perf
26     ### ... similar to proc[2] and proc[3] ###
27
28 proc_6_.perf: proc_[2-5]_.perf
29     xmodel -i top_b_1__y0_out.ts -1 freq=3.4e9 \
30           -o top_c_x0_avl.ts -2 freq=3.4e9
31     mv top_b_1__y0_out.ts top_c_x0_avl.dat
32     xmodel -i top_b_2__y0_out.ts -1 freq=3.4e9 \
33           -o top_c_x1_avl.ts -2 freq=3.4e9
34     mv top_b_2__y0_out.dat top_c_x2__avl.dat
35     xmodel -i top_b_3__y0_out.ts -1 freq=3.4e9 \
36           -o top_c_x2_avl.ts -2 freq=3.4e9
37     mv top_b_3__y0_out.dat top_c_x2_avl.dat
38     xmodel -i top_b_4__y0_out.ts -1 freq=3.4e9 \
39           -o top_c_x3_avl.ts -2 freq=3.4e9
40     mv top_b_4__y0_out.dat top_c_x3_avl.dat
41     proc_6_ >& proc_6_.out
42     echo Done > proc_6_.perf
43
44 clean:
45     rm proc_[1-6]_.out proc_[1-6]_.perf
```

Figure 5.2: Simulation Makefile for first mapping of `test2`

Each of the four processors, `proc[2]`, `proc[3]`, `proc[4]`, and `proc[5]`, take 10s to run to completion. To simulate these four processors, the single-processor core machine must run a total of 40s. Note that the communication modeling time is being ignored because it is negligible in comparison to the CR simulation time. Finally, the `proc[6]` CR takes approximately 10s to run and simulate. The total time required to simulate the `test2` application on a single-core machine is thus approximately 60s.

Consider now the scenario where the same X-Sim simulation is run on a four-core chip multiprocessor (CMP) machine. As before, the simulation for `proc[1]` is run before anything else in the application. This simulation takes 10s, same as before. The simulations for `proc[2]`-`proc[5]` are dependent on the `proc[1]` simulation, but are not dependent on each other. Thus, all the simulations for these four CRs can be run in parallel on the four processors of the CMP. Each of the four simulations take 10s to run. When run in sequence, the four simulations took 40s to run. In contrast, running the four simulations in parallel on the CMP only takes 10s total.

Finally, `proc[6]`, which is dependent on the simulations for `proc[2]` - `proc[5]`, is run on a single core. This last simulation takes 10s. The total time for this CMP parallel simulation of `test2` is thus 30s rather than the 60s for the single-core sequential simulation. This represents a speedup of $2\times$ to run the X-Sim simulation. The speedup possible through the use of a CMP simulating parallel CRs is dependent on the parallelism present in the application mapping and on the number of processors available on the CMP the simulation is being run on. Future work may include the ability to distribute simulation work over a cluster of computers, allowing even more speedup in the simulation of application mappings with large parallelism.

5.2 Substituting CR Simulations

Another technique for speeding X-Sim simulations is to substitute CR simulations with either trace-based models or with analytic distribution models. It is important to note here that simulation speedup using CR substitution is only possible when the CR has been defined as an X-Eval block

in a `.smx` file. Figure 5.3 shows a semantics file where the user has defined complete production rules for each block.

```
1  block a
2      /* ... */
3      rule a.src -> a.y0 and a.y1 and a.y2 and a.y3
4  block b1
5      /* ... */
6      rule b1.x0 -> b1.y0
7  block b2
8      /* ... */
9      rule b2.x0 -> b2.y0
10 block b3
11     /* ... */
12     rule b3.x0 -> b3.y0
13 block b4
14     /* ... */
15     rule b4.x0 -> b4.y0
16 block c
17     /* ... */
18     rule c.x0 and c.x1 and c.x2 and c.x3 -> c.sink
19 block top
20     /* ... */
21     rule a.src ->c.sink
```

Figure 5.3: Semantics file for example `test2`

5.2.1 Using Trace-Based Models

If the user runs the simulation for `test2` and analyzes the results, they will find that the total resultant application running time for the application is 30s. All the CR simulations are native processor executions, so the simulation running time is also 30s. Note that the application running time refers to the predicted running time of the deployed application, while the simulation running time refers to the time it took to simulate the application to get the predicted time. For the initial mapping of `test2`, thus, both the application running time and the simulation running time are 60s.

If the first three (from a total of ten) executions for block `c` were plotted, the timeline shown in Figure 5.4 would be generated. This timeline has been specially modified to show each execution of

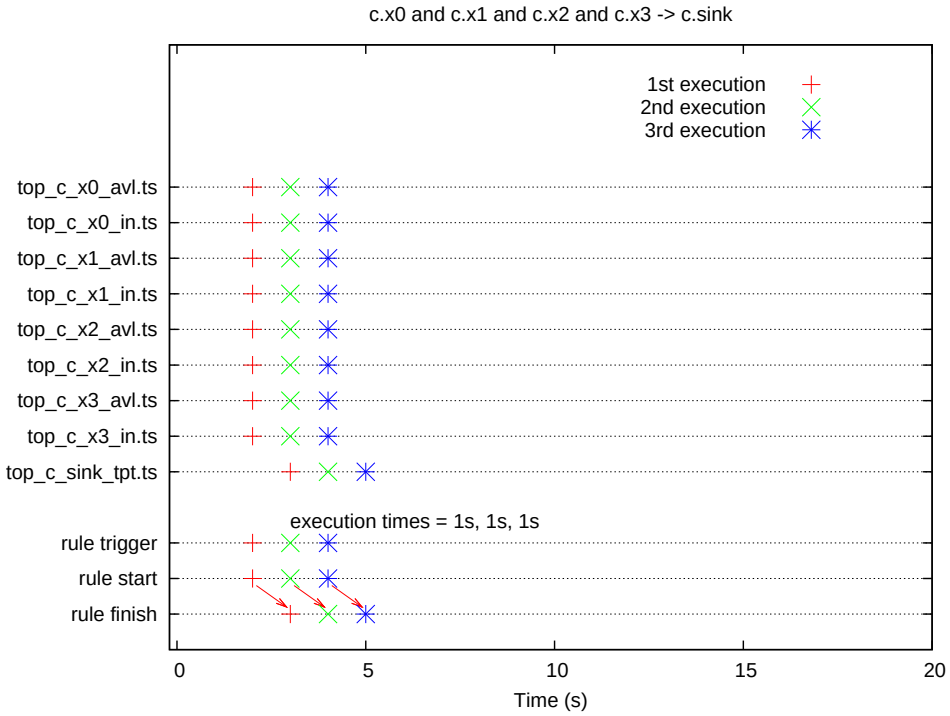


Figure 5.4: Timeline for block `c` (First mapping of `test2`)

the production rule using a different symbol. Additionally, production rule trigger, start, and finish times have also been added to show how execution times were calculated. Since we are looking at theoretical numbers here, we find that the timeline gives us exactly 1s as the execution times for each time block `c` ran.

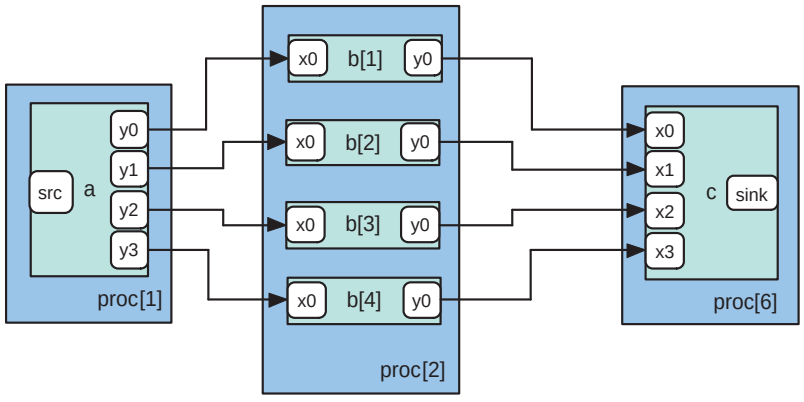


Figure 5.5: Second mapping for application `test2`

```
1  simulate: proc_[126].perf
2      echo Simulation done.
3
4  proc_1_.perf:
5      proc_1_ >& proc_1_.out
6      echo Done > proc_1_.perf
7
8  proc_2_.perf: proc_1_.perf
9      xmodel -i top_a_y0_out.ts -1 freq=3.4e9 \
10            -o top_b_1__x0_avl.ts -2 freq=3.4e9
11     mv top_a_y0_out.ts top_b_1__x0_avl.dat
12     xmodel -i top_a_y1_out.ts -1 freq=3.4e9 \
13            -o top_b_2__x0_avl.ts -2 freq=3.4e9
14     mv top_a_y1_out.ts top_b_2__x0_avl.dat
15     xmodel -i top_a_y2_out.ts -1 freq=3.4e9 \
16            -o top_b_3__x0_avl.ts -2 freq=3.4e9
17     mv top_a_y2_out.ts top_b_2__x0_avl.dat
18     xmodel -i top_a_y3_out.ts -1 freq=3.4e9 \
19            -o top_b_4__x0_avl.ts -2 freq=3.4e9
20     mv top_a_y3_out.ts top_b_4__x0_avl.dat
21     proc_2_ >& proc_2_.out
22     echo Done > proc_2_.perf
23
24 proc_6_.perf: proc_1_.perf proc_2_.perf
25     xmodel -i top_b_1__y0_out.ts -1 freq=3.4e9 \
26            -o top_c_x0_avl.ts -2 freq=3.4e9
27     mv top_b_1__y0_out.ts top_c_x0_avl.dat
28     xmodel -i top_b_2__y0_out.ts -1 freq=3.4e9 \
29            -o top_c_x1_avl.ts -2 freq=3.4e9
30     mv top_b_2__y0_out.dat top_c_x2__avl.dat
31     xmodel -i top_b_3__y0_out.ts -1 freq=3.4e9 \
32            -o top_c_x2_avl.ts -2 freq=3.4e9
33     mv top_b_3__y0_out.dat top_c_x2_avl.dat
34     xmodel -i top_b_4__y0_out.ts -1 freq=3.4e9 \
35            -o top_c_x3_avl.ts -2 freq=3.4e9
36     mv top_b_4__y0_out.dat top_c_x3_avl.dat
37     proc_6_ >& proc_6_.out
38     echo Done > proc_6_.perf
39
40 clean:
41     rm proc_[126]_.out proc_[126]_.perf
```

Figure 5.6: Simulation Makefile for second mapping of `test2`

Now suppose that the user wants to try out a slightly different mapping for comparison, where blocks `b[1]`-`b[4]` share the same processor `proc[2]`. This new mapping is shown in Figure 5.5. The Makefile generated for this new mapping is shown in Figure 5.6. Re-running this simulation will take 10s for `proc[1]`, 40s for `proc[2]`, and 10s for `proc[6]`, giving a total simulation run time of 60s. Blocks `b[1]` - `b[4]` are now combined into one single-threaded binary. Thus, in a native execution simulation, only a single processor can be used to simulate `proc[2]`, resulting in a 40s simulation running time for `proc[2]`.

Say that the user is confident that X-Eval block `c` (corresponding to X Language block `c` and CR `proc[6]`) will perform about the same as in the previous simulation run. Recall that running X-Eval on the simulation results for an application generates `exec.ts` execution time trace files for each X-Eval block. The execution time traces in these files can be applied to the new `avl.ts` timing traces (from the re-simulation) to generate `in.ts` and `out.ts` timing traces.

Figure 5.7 shows the `avl.ts` that are generated for the input ports of X-Eval block `c` (i.e., for CR `proc[6]`). These `avl.ts` timestamps reflect the times data became available to the `proc[6]` under the new mapping. Note the difference in `avl.ts` timestamps in this simulation and the `avl.ts` timestamps in the previous simulation. The reason data became available at later times in the simulation of the new mapping is that blocks `b[1]` - `b[4]` have to share processing time on a single processor, and thus take longer to produce data.

Once new `avl.ts` times for block `c`'s input ports have been generated in the re-simulation, the execution times calculated from the previous simulation of block `c` (1s, 1s, 1s, ...) can be applied to the new `avl.ts` times to generate new `in.ts` and `out.ts` times. Figure 5.8 shows the new `avl.ts` generated by the re-simulation of the re-mapping, as well as the new `in.ts` and `out.ts` times generated by applying the execution times from before to the new `avl.ts` times.

First the rule trigger times are found by looking at the `avl.ts` times. For example, the first rule trigger time is calculated to be 5s, because that is the that latest `avl.ts` time for the first input record. Note that an input record for block `c` consists of one event from all four input ports. At 5s, there was data available at every input port, and so the rule was triggered for the first time. Since the CR is not busy executing a previous triggering (this is the first triggering), it can start executing

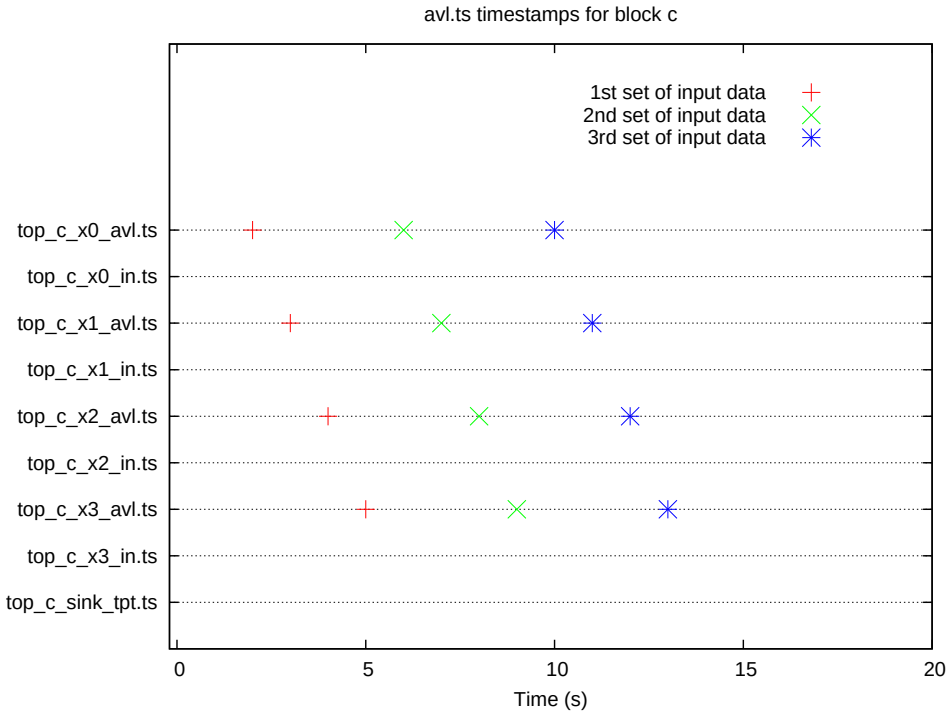


Figure 5.7: `avl.ts` times for block `c` (Second mapping of `test2`)

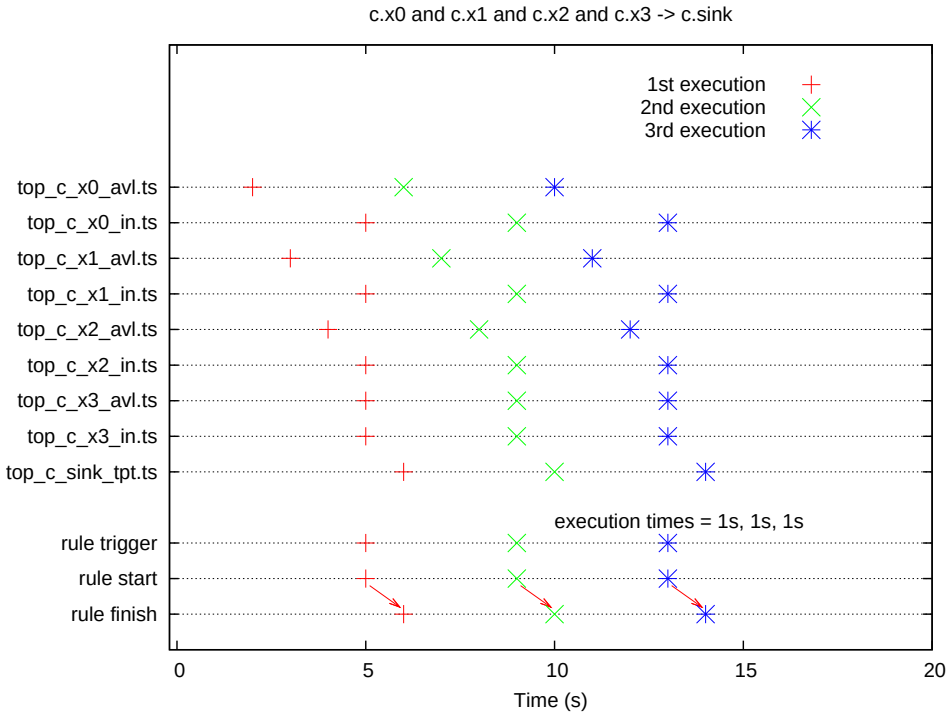


Figure 5.8: Using execution time traces for block `c` (Second mapping of `test2`)

the production rule right away. The first rule start time is thus also 5s. All the `in.ts` times for all input data is set to this time. A delay of 1s is applied to the rule start time to generate the rule finish time to get 6s. A delay of 1s is used because that is the execution time recorded from the previous X-Sim run of block `c`. All the `out.ts` times are set to the rule finish time, 6s. All the execution times recorded for block `c` from the previous simulation run can be applied to the new `avl.ts` times to get new timing traces for block `c` without having to re-run the CR simulation for `proc[6]`. This same technique can also be applied to block `a`. Block `a` is a source block, so there are no data availability constraints on its operation. Execution times are applied one after the other in a sequence to generate `top_a_src_tpt.ts` and `top_a_y[0-3]_out.ts` timing traces for each execution of block `a`.

The tool that uses execution traces as a substitute for re-simulating a CR is called `xmodel2`. Using `xmodel2` as a substitute for a CR simulation can greatly reduce the time required to simulate that CR. For example in the re-simulation of `test2`, the re-simulation times for `proc[1]` and `proc[6]` can be reduced to almost zero, making the total X-Sim simulation time 40s instead of 60s. This corresponds to a theoretical simulation speedup of 1.5×.

To make the simulation substitutions, lines 31, 34, 37, and 40 are deleted, because the CR simulation substitution does not makes use of any data files. Additionally, line 5 from Figure 5.2 is replaced by:

```
5 xmodel2 -x test2.smx -b a
```

and line 41 is replaced by:

```
37 xmodel2 -x test2.smx -b c
```

The template for invoking `xmodel2` is:

```
xmodel2 -x <smxfile> -b <block>
```

The `-x <smxfile>` option supplies `xmodel2` with the `.smx` file which lists production rules and timestamp files for each block, while the `-b <block>` option specifies which block to simulate. Each production rule for the specified block is run, with the corresponding `exec.ts` file used to get execution times from the previous simulation run for each production rule. The corresponding `index.ts` file is used by `xmodel2` to figure out which ports a particular production rule run output data to. The `index.ts` file stores the index of the output conjunction that each run of a production rule output to.

The `xmodel2` tool is essentially a more complicated version of the `xmodel` edge communication simulator discussed earlier in Chapter 3. Recall that `xmodel` is used to simulate communication over an edge by inputting a single `out.ts` timing trace file, applying a constant delay (e.g. 5μs), and outputting a single `avl.ts` timing trace file. The `xmodel2` tool is more complicated, and can be used as a substitute for CR simulations. The basic operation of the `xmodel2` is that it reads `avl.ts` timing trace files for input ports on a CR, applies an execution time delay, and writes `in.ts` and `out.ts` timing trace files. If a testpoint is used as an input port for an X-Eval block, there are no `avl.ts` times that need to be considered by `xmodel2`. It triggers and starts running production rules whenever it is done executing the previous production rule run.

5.2.2 Using Analytic Distribution Models

The `xmodel2` tool can also use analytic distributions to model a CR's timing performance. When reading an `exec.ts` file, `xmodel2` reads the trace file header to check if the user wants to use an analytic distribution to model the execution time. If an analytic distribution is supplied, `xmodel2` generates a random value from the distribution for each production rule run and uses that as the execution time.

Currently, only a normal distribution is supported for the execution time analytic modeling. Two header options, `mean` and `stddev`, can be set to values by the user to cause `xmodel2` to use a normal distribution to model execution times. For example, to model block `c`'s execution time with a normal distribution with a mean of 1s and standard deviation of 0.001s, the header of the `top_c_rule1_exec.ts` can be modified to the following:

```
\#XTSFile
mean=1
stdev=0.001
end
```

The header of a `.ts` file should always be 512 bytes long. The easiest way for users to modify a `.ts` header is to edit the file in insert mode, taking care to make sure the header file has the keyword `end` as the last word.

Chapter 6

Benchmarks

This chapter presents results gathered from running X-Sim and X-Eval on sample X-Applications. The first section presents results gathered from simulating and analyzing a version of the `test1` application that has been used as an illustrative example throughout this thesis. The second section presents the VERITAS application, a real world scientific application that has been developed using the Auto-Pipe toolset.

6.1 The `test1` Example Application

In the `test1` application, each `GENERATE` block generates a single 32-bit unsigned number. These individual numbers are passed through various blocks to calculate the average. To reduce the inter-block data movement overhead associated with X Language generated code, arrays of 32-bit numbers are generated and processed rather than individual numbers. For the results presented in this section, arrays of size 1000 were used. Each generate block is run one million times, resulting in a total of one million *arrays* of averages (or equivalently one billion *individual* averages) being calculated. The `test1` application blocks are the same as before, with `gen1` and `gen2` feeding into `sum`, `sum` feeding into `half`, and finally `half` feeding into `store`. The only difference now is that arrays of numbers are passed around rather than individual numbers.

We will consider the problem of mapping this array version of the `test1` application to a four-processor SMP (Symmetric Multi-Processing) system. The actual physical system used to test out

deployments was an AMD Athlon 64 X2 4400+ system with four cores, 1MB L2 cache and 8GB of system memory. A simple graphical representation of the architecture is shown in Figure 6.1, where four processors are connected to each other by common shared memory. The figure also shows the first mapping we will consider, where the entire application is mapped to a single processor, `proc[1]`.

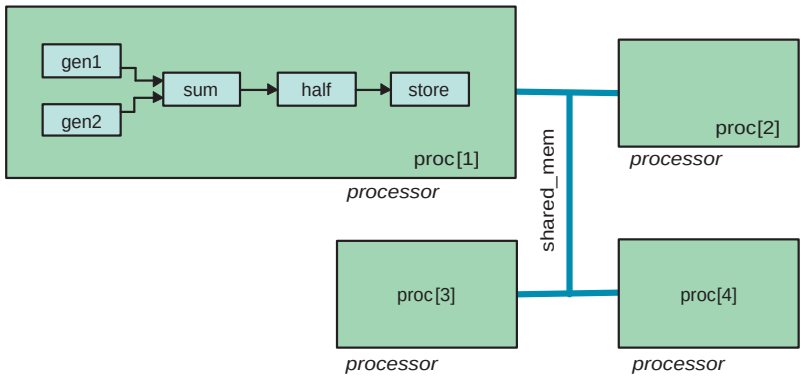


Figure 6.1: A single-processor mapping of `test1`

A simulation was initially run with no testpoints added, but with cumulative statistics collection active. Note that the X-Sim simulation for this mapping is simply running the compiled binary natively on the target machine, and thus corresponds directly to running the actual physical deployment. Times for both the complete simulation run, as well as for the complete physical deployment run, are thus the same value. The total measured time to run the entire application to completion in this one-processor mapping was 267.5 seconds. Since the native machine 1-processor simulation corresponds directly to the deployment binary, simulation validation (i.e., whether simulation run times reflect actual deployment run times) is not required.

Figure 6.2 shows the cumulative execution time for each block, gathered from the built-in statistics collection, over the entire run of the `test1` application. These values represent the total amount of time spent in each block to process all million arrays (of a thousand numbers each) over the entire application run.

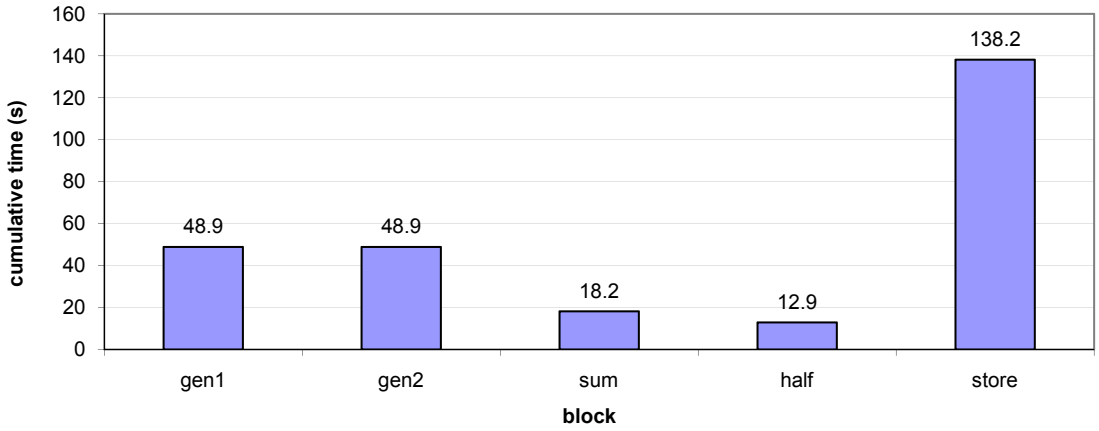


Figure 6.2: Total processing time per block in `test1`

As a next experiment, testpoints were inserted at the start and end of the execution of the `store` block. These testpoints provide very detailed data about when the block started and ended processing each array of numbers. For example, the first 6 execution time traces (i.e., processing for six successive arrays) for the `store` block are shown below:

171.9μs, 132.5μs, 135.1μs, 132.3μs, 149.8μs, 133.5μs

An inspection of all the execution times shows that the first execution took the longest, 171.9μs. The rest of the execution times are in the range 132μs-150μs. The mean of all the execution time traces is 135μs. It is possible that the reason that the first execution time is higher than all subsequent times is that `store` block incurs a small penalty associated with accessing a file for the first time.

The distribution of execution times that was shown in Figure 6.2 shows that the `store` block takes roughly half of the total application run time (i.e., 138.2s / 267.5s). To improve performance, the `test1` application was re-mapped to two processors with only the `store` block mapped to the second processor, as shown in Figure 6.3.

X-Sim was used to simulate this mapping, with the shared memory link modeled with a zero delay communication link. This simulation gave a complete application running time of 138.3s. Note that in this case, a single processor was used to first simulate execution on `proc[1]`, then to simulate the communication (zero delay) between `proc[1]` and `proc[2]`, and finally to simulate the

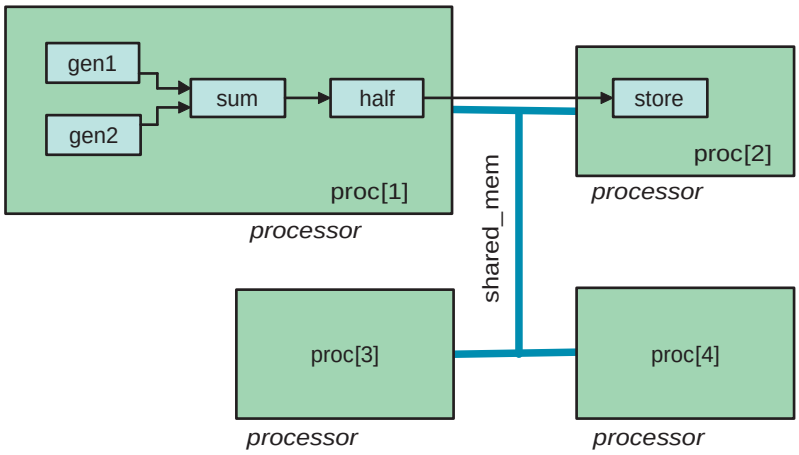


Figure 6.3: A two-processor mapping of `test1`

execution on `proc[2]`. The X-Sim simulation thus predicts that the new dual-processor mapping will result in a speedup of 1.93× (i.e., 267.5s/138.3s) over the single-processor mapping.

To validate these simulation results, the new mapping of the `test1` application was deployed on the previously described AMD SMP. Two processors were utilized in this 2-processor mapping, with actual shared memory the communication mechanism rather than the simplified zero-delay communication model system that was employed by X-Sim. Running this application took 143.3s with both processors running in parallel, showing a speedup of 1.87× (267.5s / 143.3s). Thus, the predicted time given by X-Sim (138.6s) was 3.5% off from the actual deployed two-processor mapping (143.3s). Application running time results for the simulation and deployed runs for both mappings is shown in Figure 6.4.

The 2-processor mapping, with `store` on one processor and everything else on another processor, thus shows an almost 2× speedup over the 1-processor mapping. Without splitting up the `store` block, it is not possible to get higher speedups using 3 or 4-processor mappings. This is because the `store` block takes up almost half the processing, and in any distributed processing it will stay the bottleneck, thus limiting the speedup to about 2×. More distributed mappings and varied architectures will be explored in the next section, where a scientific application is the target for performance optimization.

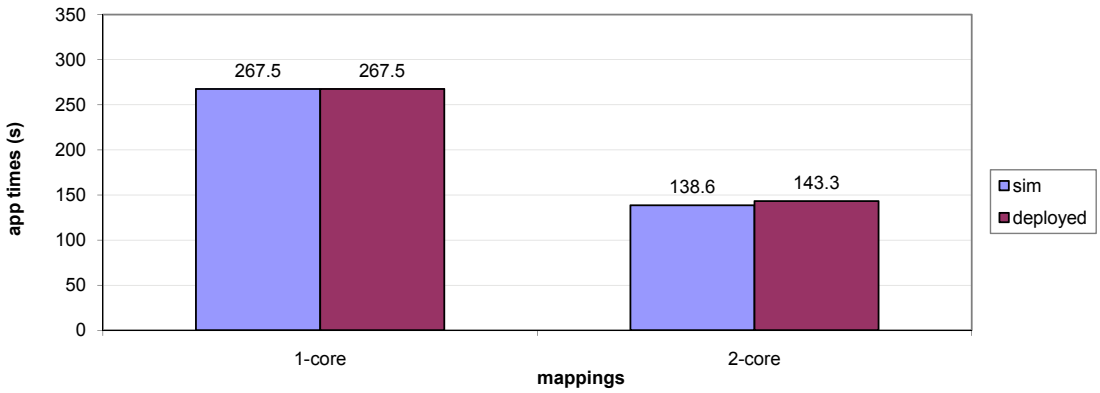


Figure 6.4: Run times for 1-processor and 2-processor `test1` mappings

6.2 The VERITAS Application

We now move on to simulating and analyzing the VERITAS [22] application, a gamma ray event parametrization application from the field of astrophysics. The acronym VERITAS stands for Very Energetic Radiation Imaging Telescope Array System. High energy gamma rays are emitted from a variety of extraterrestrial sources, including pulsars, supernova explosions, and supermassive black holes. The gamma rays strike the Earth's atmosphere and produce Cherenkov radiation, electromagnetic shockwaves in the blue through ultraviolet portion of the spectrum. This radiation is recorded in the form of arrays of pixels by ground-based telescopes. A significant amount of processing must be done on the raw signals to produce meaningful data that physicists can use. In a real-time deployed system, data from the telescopes is streamed through and processed on the fly. In an off-line version of the system, data is read from files and processed. This is the version of the VERITAS application that is considered in this section. A simple representation of the major blocks in the VERITAS application is shown in Figure 6.5.

In this simplified representation of the VERITAS application, the first block, the `FileRead` block, reads telescope measurements from files on disk. Data is organized into `Events`, where each `Event` consists of 499 parallel `Pixels`, where each `Pixel` is the signal output from a single photon detector in a telescope detector array. Timing measurements presented in this section are for application runs done with 5000 `Events`. The next block after the `FileRead` block is the `Splitter` block, which splits each `Event` up into its constituent `Pixels`. The `Splitter`

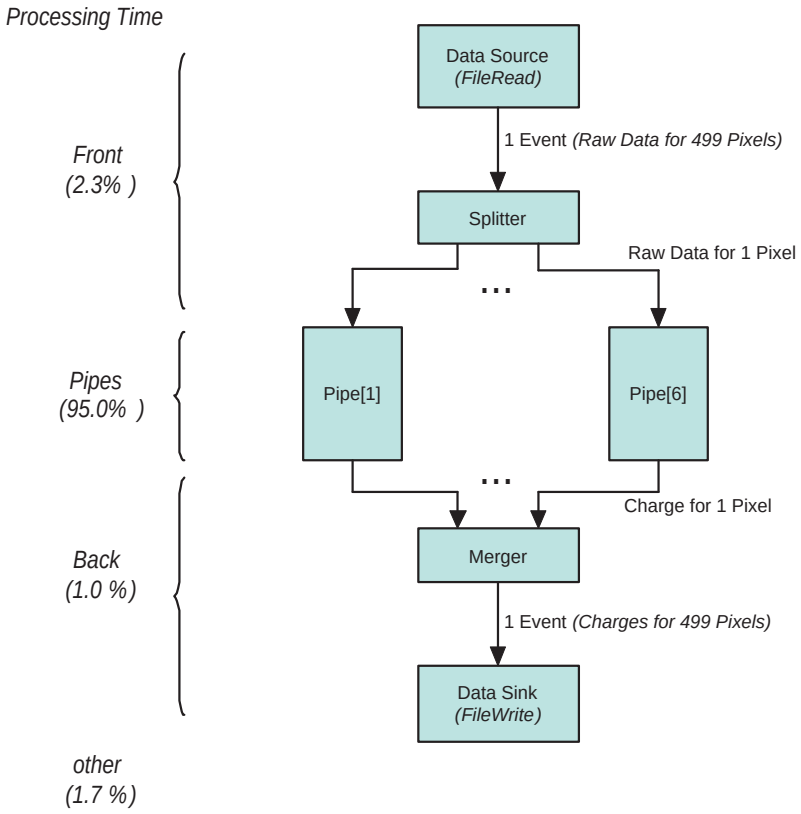


Figure 6.5: The VERITAS application

block outputs a batch of 6 `Pixels` at a time, sending one `Pixel` to the each of parallel downstream `Pipe` blocks. With 6 `Pipes`, the `Splitter` block thus produces 84 separate sequential batches (499/6=83.2) of 6 `Pixels` each. The `Splitter` block sends null `Pixels` on the last batch to even out the number of `Pixels` in it to 6. The number of `Pipes` is a customizable parameter, but for now we will consider it to be fixed at 6.

After the `Pipe` blocks have processed the `Pixel` values into `Charge` values, the data values are streamed into a `Merger` block. This block merges together each set of 499 charges, performs additional computations on the aggregate `Charge` values and creates a processed `Event`. Finally, the `Output` block records the results into a file on disk. The `Merger` block, like the `Splitter` block, is smart enough to handle situations where the number of `Pixels` (i.e., 499) does not divide evenly into the number of `Pipes` (e.g., 6). Note that this block diagram is a highly simplified representation of the actual VERITAS application, and [20] should be referred to for a more detailed

description of the application. On the left of the diagram, a three stage-division of the VERITAS application has been shown, along with percentages for the portion of processing time spent in each stage. These numbers are gathered from running the application on a single processor, and serve to illustrate the processing needs of different parts of the application.

On a single processor of the AMD Athlon system, the entire VERITAS application run took 127.2s. The `FileRead` and `Splitter` blocks are grouped under the `Front` section, and accounted for 2.3% of the total running time. The middle `Pipes` accounted for the bulk of the processing time, taking up 95.0%. The `Back` section took 1.0% of the application running time, while the remaining 1.7% of the running time was unaccounted for, taken up by processing outside the X-Language blocks.

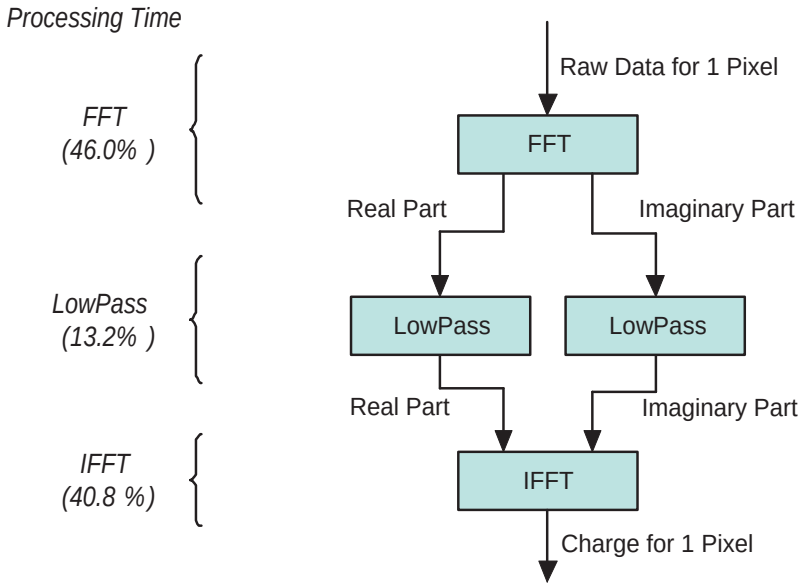


Figure 6.6: A `Pipe` block from the VERITAS application

From these percentages, we can see that the bulk of the processing is done inside the `Pipes` section. A more detailed view of a `Pipe` is shown in Figure 6.6. The first block in the pipe is the `FFT` (Fast Fourier Transform) block, taking up 46.0% of the total time spent in a `Pipe`. The two parallel `LowPass` filter blocks take up 13.2% of the `Pipe's` processing time, while the final `IFFT` (Inverse FFT) block takes 40.8%. Note that once again, this is a simplified view of the actual algorithm. The `FFT` and `IFFT` blocks shown in this diagram actually include some other blocks that

have been lumped along with the FFT and IFFT. These other blocks will be described later in this section.

6.2.1 Partitioning the VERITAS Application

We now look at the problem of mapping the VERITAS application to a two processor system, and consider both simulation and deployed results. From the distribution of processing times, we know that the bulk of processing time is taken up in the `Pipe` blocks. We will look at two approaches to partitioning the VERITAS application to two processors, one *vertical* and one *horizontal* (Figure 6.7). In the vertical mapping, the `Front` section and the three `Pipe` blocks on the left are mapped to one processor, while the `Back` section and the three `Pipe` blocks on the right are mapped to the other processor. In the horizontal mapping, everything feeding the `IFFT` blocks is mapped to one processor (i.e., everything up to and including the `LowPass` blocks), while the `IFFT` and downstream blocks are mapped to the other processor. Note that the horizontal mapping could be made more balanced by moving the `LowPass` blocks from `proc[2]` to `proc[1]`. However, these blocks have been mapped to `proc[2]` for demonstrative purposes, making the horizontal partitioning significantly more imbalanced than the vertical partitioning.

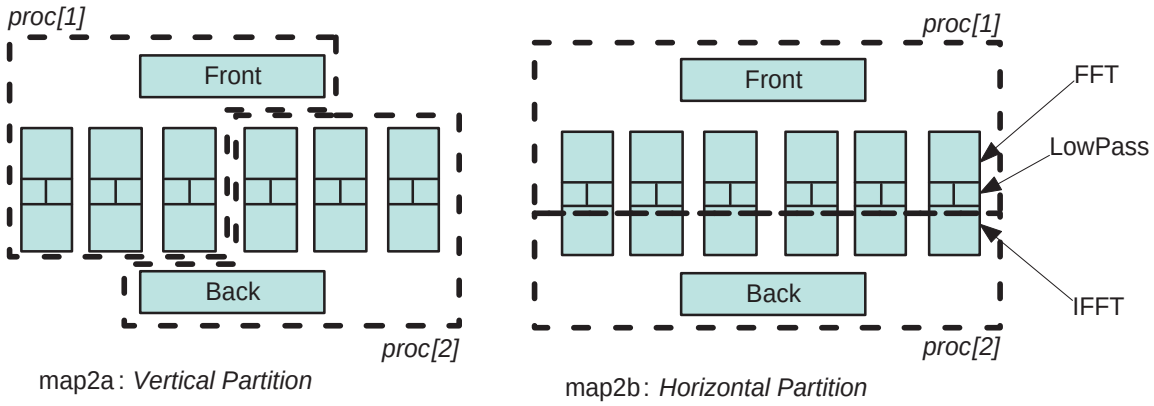


Figure 6.7: Vertical and Horizontal 2-Core Mappings

Figure 6.8 shows the results. The bars labeled `map2a` and `map2b` show the total application run time measured from X-Sim simulations as well as deployments of the two mappings. Also shown in this figure is the running time for running the 1-processor mapping, as well as 3-processor mappings

that are described later. There are two main things to note here. The first is that the simulation times are within 5% of the corresponding times measured on the multi-processor deployed application. Also of note is that the vertical and horizontal mappings are relatively close to each other, with times of 70.2s and 75.4s. The vertical 2-processor mapping is a bit faster than the horizontal 2-processor mapping, due to the fact that it is a more balanced partitioning of the processing costs. Note that in the figure, speedups (1.81× for `map2a` and 1.69× for `map2b`) over the 1-processor mapping are shown on the y-axis rather than absolute times.

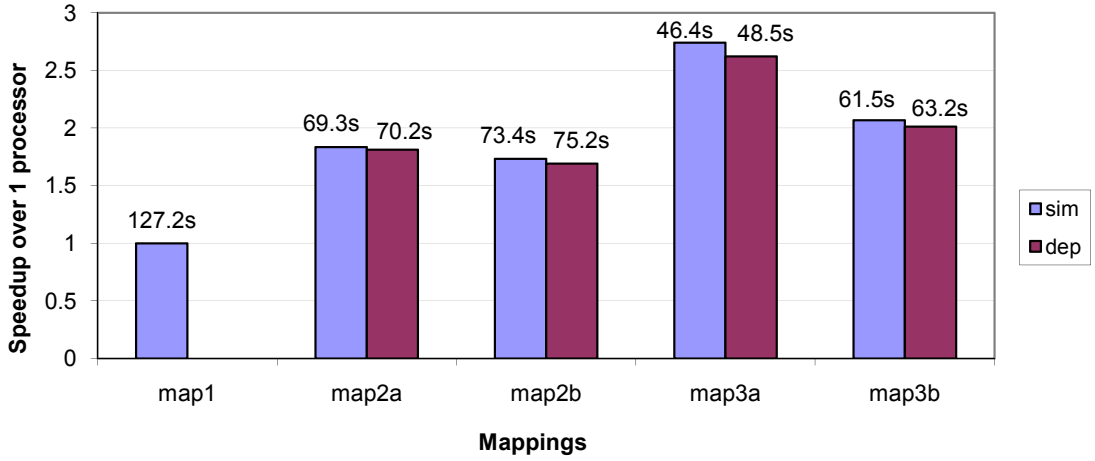


Figure 6.8: VERITAS speedups on 2 and 3 processors

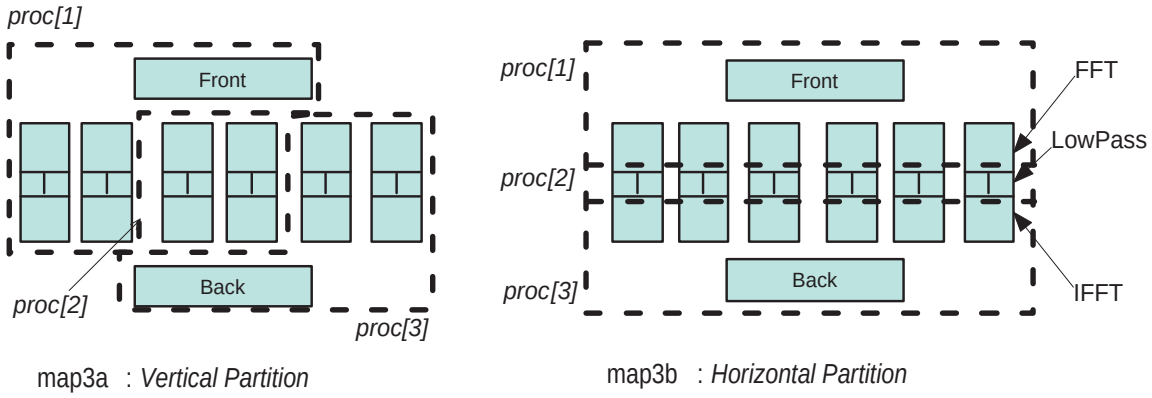


Figure 6.9: Vertical and Horizontal 3-Core Mappings

The next mapping problem was to run the VERITAS application on three processors. Once again, two mappings were evaluated, one vertical and one horizontal. These mappings are shown in Figure 6.9. In the vertical mapping, two `Pipe` blocks are mapped to each processor, while the

`Front` block is additionally mapped to `proc[1]` and the `Back` block is additionally mapped to `proc[3]`. In the horizontal 3-processor mapping, the `LowPass` blocks are the only things mapped to `proc[2]`. Everything upstream of the `LowPass` blocks is mapped to `proc[1]`, while everything downstream is mapped to `proc[3]`.

The vertical mapping is a relatively balanced partitioning. The horizontal mapping, mapping however, is not very balanced. This is because the part (`LowPass`) of the `Pipe` blocks mapped to the second processor is not as significant a portion of total processing as the `FFT` and `IFFT` blocks. Results from the 3-processor mappings, in terms of speedups over the 1-processor mapping, are shown on the bars labeled `map3a` and `map3b` in Figure 6.8. Once again the times from running the X-Sim simulations of the two mappings are within 5% of the times gathered from running the deployed mappings.

The vertical three-processor mapping shows a 2.6× speedup over the single-processor mapping, while the horizontal three-processor mapping only shows a 2.0× speedup. This difference can be attributed to the difference in balancing the processing load between the two mappings.

So far we have been running simulations where the communication bandwidth is effectively infinite, with zero communication delay on edges. We will now consider cases where the communication bandwidth is limited, and the effect this has on different mappings.

6.2.2 Communication Bandwidth Modeling

Recall that communication modeling in X-Sim is done on a *per edge* basis, with a fixed communication delay per transfer on that edge. To simulate a IR (Interconnect Resource) bandwidth, the total bandwidth on that IR must be split up among the different edges that are mapped to that IR. Each edge is allocated a fixed bandwidth proportional to the amount of data transferred over that edge, compared to the total amount of data transferred over all edges mapped to that IR. For example, if an IR (Interconnect Resource) has a 1Gbps fixed bandwidth capacity, and two edges carrying equal amounts of data are mapped to this IR, then each edge is allocated 500Mbps bandwidth. If one edge carries four times as much data as the other, then the bandwidth allocated to the first edge would be

800Mbps, while the bandwidth allocated to the other edge would be 200Mbps. The total amount of data transferred over each edge can be determined either by an analytic calculation, or by examining the size of the data trace files for each edge after running an X-Sim simulation.

Once a bandwidth has been allocated to each edge mapped to an IR, the delay per data transfer on each edge is calculated by dividing the amount of data transferred on a push by the edge's bandwidth. For instance, if a push on a 200Mbps bandwidth edge consists of 100 bytes, then the communication delay on that edge is (100B / 200Mbps =) 4μs.

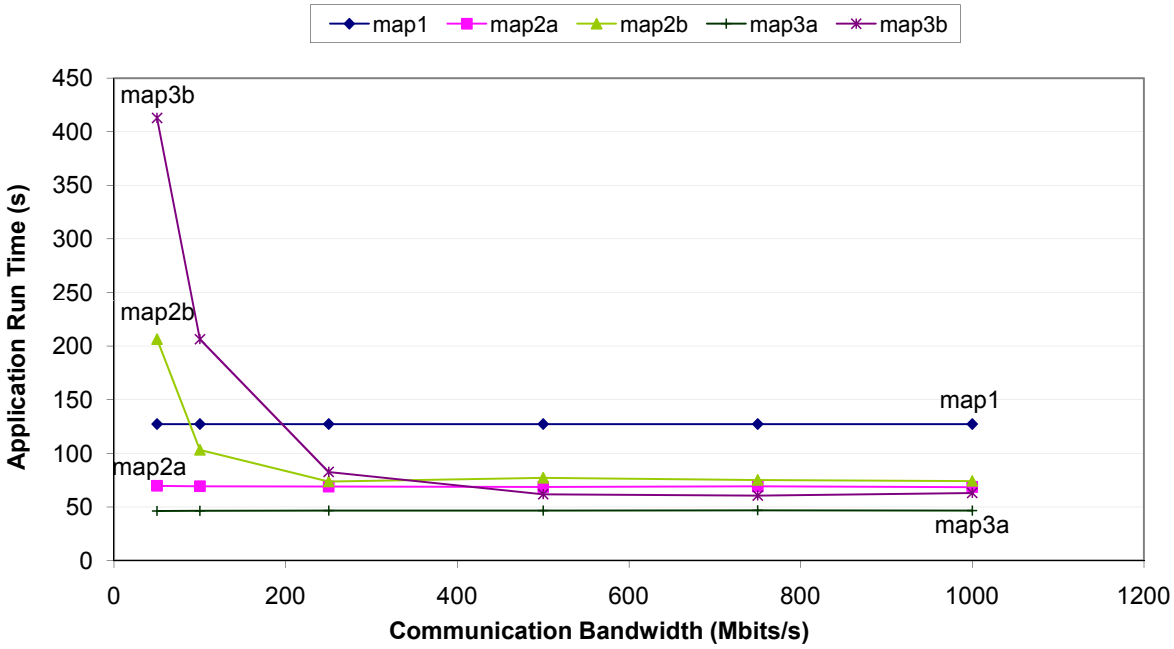


Figure 6.10: Effect of bandwidth on VERITAS running times

Figure 6.10 shows the running times gathered from simulations of the different VERITAS mappings considered so far. For each mapping, the application running time is mapped against the total communication bandwidth on a shared IR. Shared memory was effectively an infinite-bandwidth shared IR for VERITAS. This plot shows the effect of different simulated shared IRs with different communication bandwidths. Consider the data plots on the very right side of the plot. These represent application running times for each mapping when the total communication bandwidth is 1Gbps (i.e., equal to the theoretical bandwidth on Gigabit Ethernet).

In this scenario, the communication bandwidth is high enough that it is not a bottleneck. As would be expected, the one-processor mapping, `map1`, takes the longest time, 127.2s. Next slowest is the horizontal two-processor mapping, `map2b`, which takes 74.2s. The vertical two-processor mapping `map2a`, with a more balanced partitioning of processing, takes a little less time, 68.4s. Among the three-processor mappings, the horizontal mapping `map3b` at 61.5s is only a little faster than `map2a`. The vertical three-processor mapping `map3a` is the fastest of all the mappings, with an application running time of 46.4s.

It is worth noting at this point that processing by the the `FFT` block results in a very large expansion of data. The `LowPass` blocks operate on this expanded data, before the `IFFT` blocks perform a data reduction. Thus, the edges to and from the `LowPass` blocks are very high traffic compared to any other edges. The horizontal two-processor mapping, `map2b`, forces data going to the `LowPass` blocks to travel over the communication mechanism, using up limited bandwidth. The horizontal three-processor mapping, `map3b`, maps both the edges going to and coming from the `LowPass` blocks to the communication resource, and thus forces twice as much data over the same limited bandwidth.

As the communication bandwidth is decreased to 750Mbps, and then to 500Mbps, we can see that the running times for all of the mappings stay relatively constant. This is because at these levels of communication bandwidth, the communication mechanism is not yet a bottleneck. As the bandwidth is reduced to 250Mbps, the running time for `map3b` increases, while the running times for all the other mappings stays constant. This is because a communication bandwidth of 250Mbps is a bottleneck for `map3b`, but not for any of the other mappings.

When the bandwidth is reduced to 100Mbps, it becomes a bottleneck for both `map2b` as well as `map3b`. With twice as much traffic over the bottleneck communication link, `map3b` has a running time that is roughly twice the running time for `map2b`. The bandwidth is not a bottleneck for any of the other mappings, and so their running times are still the same as before.

Even when the bandwidth is halved to 50Mbps, `map1`, `map2a`, and `map3a` have the same running time as they did before, indicating that the communication link is still not a bottleneck for these

mappings. Meanwhile, `map2b` and `map3b` take twice as long to run the VERITAS application on a bandwidth of 50Mbps as they did to run on a bandwidth on 100Mbps.

6.2.3 Performance Scaling with Multiple Processors

In the next experiment, we will see the X-Sim simulation results from mapping the VERITAS application to 1, 2, 3, 4, 8, and 16-processor systems. In the previous section, we saw that a vertical partitioning of the VERITAS blocks results in relatively balanced distribution of the processing load. We will take a 16-`Pipe` version of VERITAS and map the `Pipe` blocks evenly among the processor resources. For example, the 4-processor mapping has four `Pipe` blocks mapped to each processor. For the 3-processor mapping, a 15-`Pipe` version of the application was used with five `Pipe` blocks mapped to each processor.

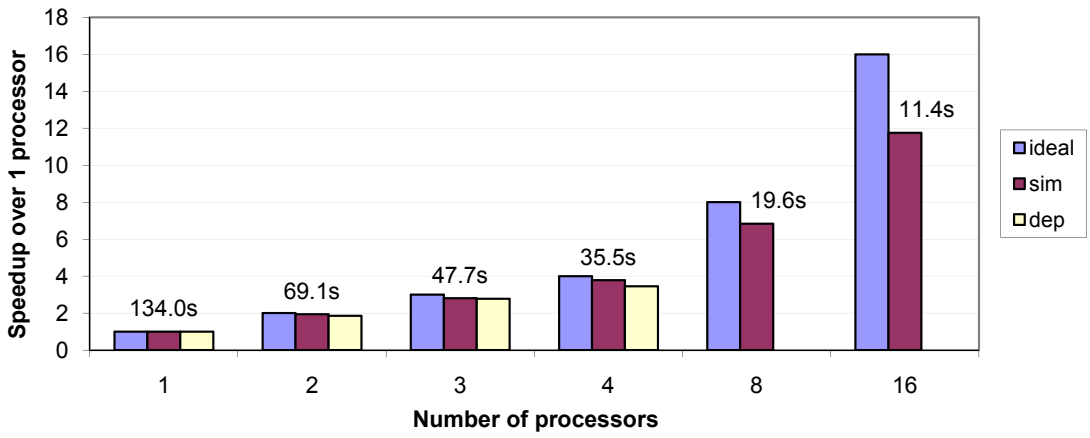


Figure 6.11: VERITAS performance scaling with number of processors

The results for simulation runs (in terms of speedups over the 1-processor mapping) for each of the multi-processor mappings are shown in Figure 6.11. Also shown in this figure are the deployed application run times for the 1 through 4-processor mappings, and the ideal speedups for each mapping. Results in the figure are shown as speedups over the 1-processor mapping.

The ideal speedup for each mapping is equal directly to the number of processors. For example, the ideal speedup for a 4-processor system is 4. The graph shows that X-Sim predicts times that

roughly follow the ideal speedups, with the difference from the ideal increasing with the number of processors. The likely cause for this is that the `Front` and `Back` sections are always mapped to the first and last processors, and only the `Pipe` section processing is evenly distributed among the different processors. Say the processing on `Front` takes f seconds, while the processing on a `Pipe` takes p seconds. For the 8-processor mapping, `proc[1]` has the `Front` section and 2 `Pipes` mapped to it (and takes $f + 2p$ seconds). For the 16-processor mapping, `proc[1]` has the `Front` section and 1 `Pipe` mapped to it (and takes $f + p$ seconds). The speedup (for `proc[1]`) from 8-processor to 16-processor would thus be $(f + 2p)/(f + p)$. Going a step back, the speedup from 4-processor to 8-processor can similarly be calculated to be $(f + 4p)/(f + 2p)$. Thus, the speedup from 8-processor to 16-processor is less than the speedup from 4-processor to 8-processor.

A four-processor system was used to test out physical deployments of the VERITAS application using shared memory as the communication mechanism. As shown by the graph, the 1 and 2-processor deployments match the simulation predictions fairly closely. The deployed four-processor application ran approximately 6% slower than the predicted X-Sim simulation. One possible reason for this discrepancy is that in utilizing all 4 processors available on the deployment system, the application is more likely to be interrupted by OS processes which need to run simultaneously. Another reason for discrepancies is that shared memory is able to hide its latency by allowing computation to occur in parallel with memory accesses. In mappings where a processor must both read from and write to memory, it is harder for shared memory to hide its delay, and this adds to processing time.

The comparison of simulated and deployed times helps validate X-Sim simulations of zero-delay communications. The simulation times for the 8-processor and 16-processor systems, on the other hand, show how X-Sim can be used to predict times for application runs on systems that are not available currently.

6.2.4 Simulation of A Heterogeneous System

So far we have considered various multi-processor mappings for VERITAS. In this section, we will look at a mapping of VERITAS that is targeted to a heterogeneous architecture consisting of

processor and FPGA resources. In this section, we will use X-Sim to analyze the effect of three different factors on application performance, and how they interact with each other. The first factor is communication bandwidth. We already saw in the previous section how, after a certain critical level, communication bandwidth can strongly affect the performance of an application. The second factor we will consider is parallelization. We will see how the performance of VERITAS can change when it is configured with 1, 2 or 3 `Pipes`. The third and final factor is *mapping*. Mapping is one aspect the user (and X-Opt once it has been developed) can easily change, and, as we shall see, potentially dramatically improve the performance of an application.

The heterogeneous architecture used as a target in this section is shown in Figure 6.12. In this mapping, two processors and an FPGA are connected to each other on a PCI-X bus. In the deployment system, the two processors are again AMD Athlon 64 X2 4400+ cores. The FPGA is a Virtex II 6000 running at 100MHz. The PCI-X bus is running at 100MHz with width 64bits, giving a theoretical maximum bandwidth of 800MBps.

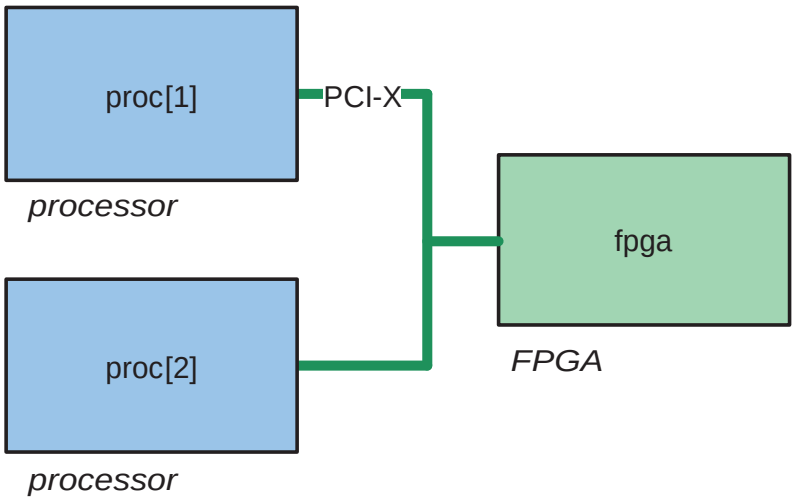


Figure 6.12: Target heterogeneous architecture for VERITAS

The driver used to communicate between the processors and the FPGA over the PCI-X are described in [4]. In this paper, the authors report sustained throughputs of 815MBps over a 133MHz PCI-X bus while transferring a steady stream of data. The scaled value for the 100MHz PCI-X bus used in the current setup is a throughput of 613MBps (815MBps $\times$ 100MHz/133MHz).

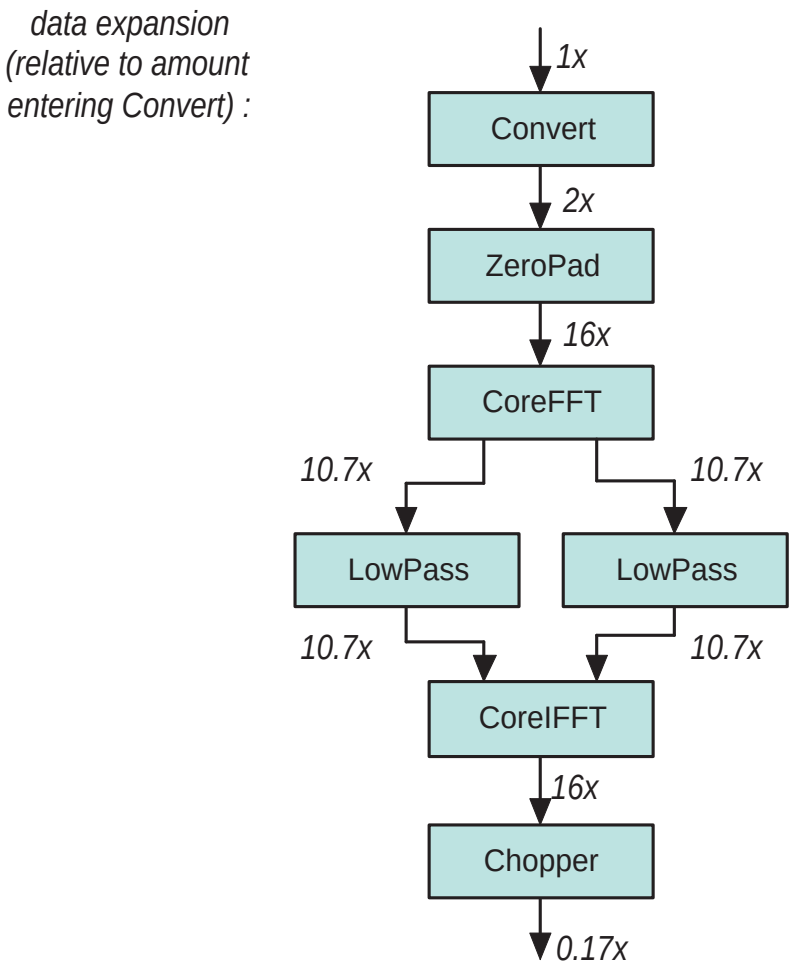


Figure 6.13: Detailed view of a VERITAS `Pipe`

Before going into the details of a mapping done to this heterogeneous architecture, it is necessary to examine a VERITAS `Pipe` in more detail. Figure 6.13 shows a `Pipe` with the `FFT` and `IFFT` blocks expanded out to show more detail. The `FFT` block has been replaced by a `Convert`, `ZeroPad`, and `CoreFFT` block. Note that the `CoreFFT` does the bulk of the computation that was done in `FFT` block. The `Convert` and `ZeroPad` blocks do not spend much time processing, but they do result in data expansions. The data output from the `ZeroPad` block is a 16× expanded version of the input into the `Convert` block.

Similarly, the `IFFT` block from before has been replaced by `CoreIFFT` and `Chopper` blocks. In this case, the `Chopper` block does not spend a large amount of time processing compared to the

CoreIFFT block, but it does do a very significant data reduction. The amount of data going out of the Chopper block is 96 times *less* than the amount of data entering it.

At the time of writing this thesis, hardware implementations (i.e., VHDL or Verilog descriptions) of the Convert and Chopper blocks were still in the process of development. As a result, a constraint was that these two blocks had to be mapped to processors. The rest of the blocks in VERITAS Pipes however had hardware (specifically VHDL) implementations available, so it was possible to map them to an FPGA. The first mapping, mapA1, is shown in Figure 6.14. VERITAS has been configured to have one Pipe. In this mapping, blocks ZeroPad through CoreIFFT were mapped to the fpga CR (Computational Resource). Two more mappings, mapA2 and mapA3 were also developed. These are simply 2-Pipe and 3-Pipe versions of mapA1. Figure 6.15 shows mapA2.

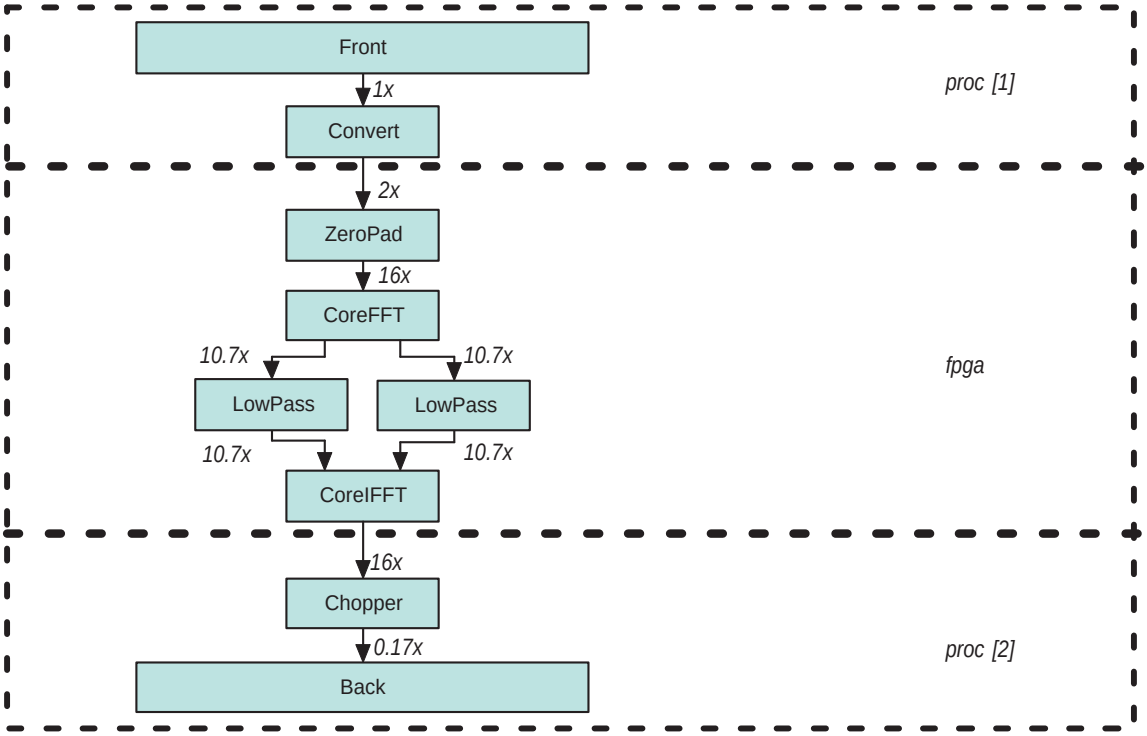


Figure 6.14: mapA1: 1-Pipe VERITAS mapped to a heterogeneous architecture

First, a heterogeneous simulation of mapA1 was run, with the PCI-X IR bandwidth set to 640MBps (rounded from the maximum of 615MBps). X-Sim used native execution simulations for proc[1] and proc[2], and a ModelSim simulation for the FPGA. The first simulation was run with only

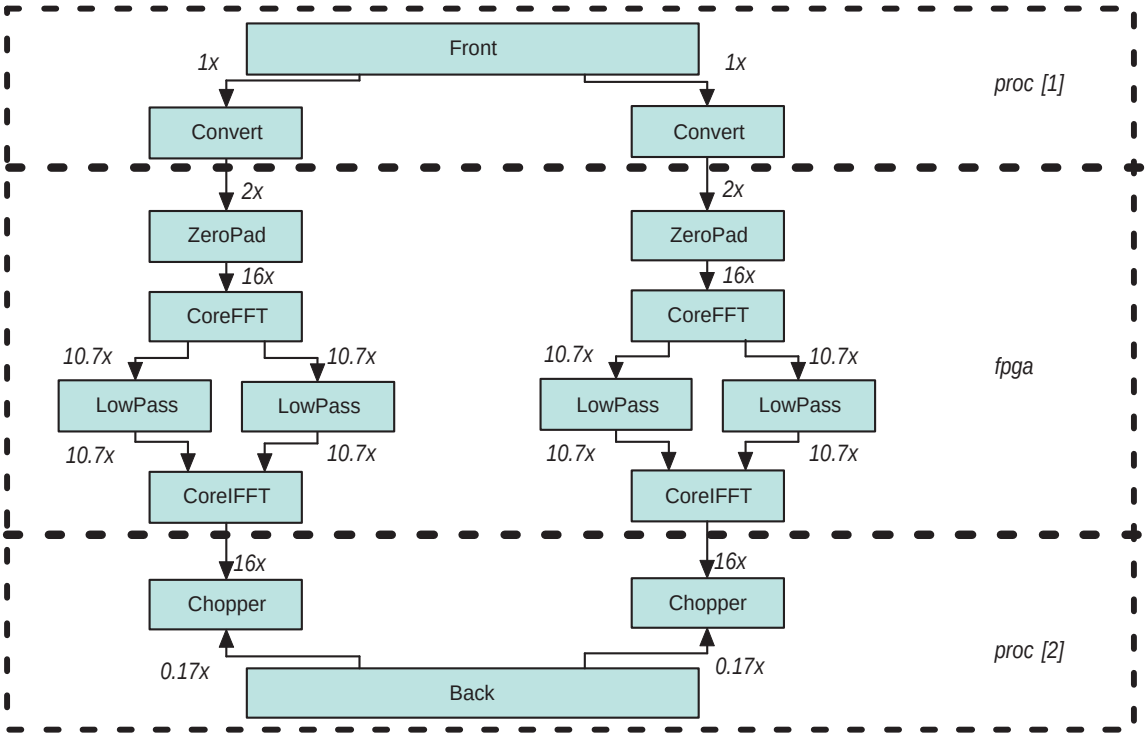


Figure 6.15: `mapA2`: 2-`Pipe` VERITAS mapped to a heterogeneous architecture

a single event for the entire simulation. This was done because the ModelSim simulation for the FPGA processing a single event took about 10 minutes. We would like to do multiple X-Sim simulations of thousands of events each, but the time required to run hardware simulations is prohibitive for such large experiments. Analytic model substitution was used to speed up the simulation. Examining the output timestamps from the ModelSim simulation showed that data was produced every 6.6μs. This corresponded roughly to the fixed clock delay associated with the `coreFFT` block, the longest delay block inside the pipeline mapped to the FPGA.

A simulation of `mapA1` was run with the `ZeroPad` through `CoreIFFT` blocks mapped to a *processor* instead of an FPGA, with the VERITAS application run on 5000 events. This generated all the trace data files that were used in subsequent simulations of the heterogeneous system. A heterogeneous simulation was then run with the FPGA analytically modeled with a fixed execution time of 6.6μs. The X-Sim simulation yielded a time of 18.3s for the entire VERITAS application run of 5000 events for the heterogeneous system.

As a comparison, the deployed system was run on the two processors and a physical FPGA. This deployed run gave a complete application run time of 96.3s, much higher than given by the X-Sim simulation (18.3s). (Note: The actual deployed application experiment was for 15000 events, and yielded a time of 289s. This time has been scaled back for 5000 events.)

Recall, however, that the X-Sim simulation was run with a communication bandwidth of 640MBps. The actual driver used for communication over PCI-X are heavily optimized for a steady stream of large chunks of data. The X-Com generated code for the deployed system, however, attempts to send smaller chunks of data, along with a high overhead of *command* or *header* data. This results in a lot less sustained throughput. In fact, the sustained throughput for `mapA1` can be calculated by dividing the total amount of data transferred over PCI-X (1028MB) by the total application run time (96.3s). The result of the above calculation gives an effective bandwidth of 10.67MBps, compared to the maximum possible streaming bandwidth of 613MBps cited earlier.

The effective bandwidth is thus much less when using the driver with small chunks of data with lots of overhead data. It might be possible in the future to improve the effective communication bandwidth, for example, by improving how the generated X Language code uses the driver, or by sending larger chunks of data at a time. A series of X-Sim simulations were run to see the effect this would have on the performance of the `mapA1` heterogeneous mapping of VERITAS. Simulations were also run to determine the performance of the 2-`Pipe` mapping `mapA2` and 3-`Pipe` mapping `mapA3` under different communication bandwidth restrictions. Figure 6.16 shows the results from these experiments.

For the `mapA1` mapping, we can see linear improvements in effective bandwidth result in linear improvements in application performance. Once the effective bandwidth is increased to about 80MBps, the interconnect ceases to be a bottleneck, and the running time settles at around 18.3s. Analytically, total processing time spent on the FPGA throughout the whole application run is given by:

execution time $\times$ number of executions

= 6.6μs $\times$ (499$\times$5000) = 16.5s

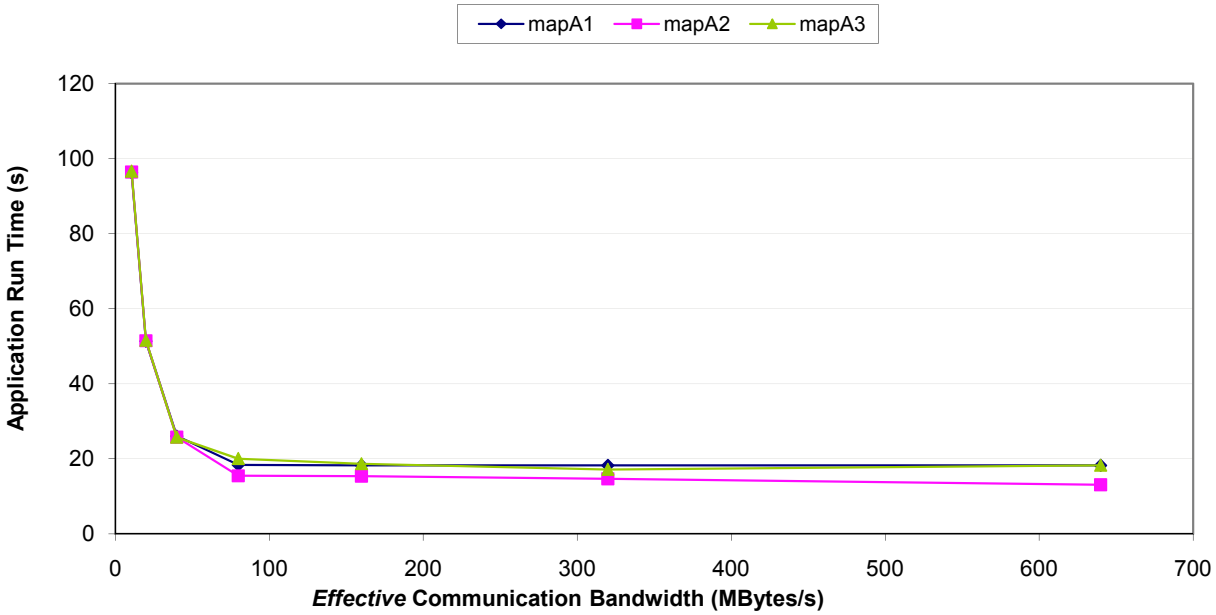


Figure 6.16: Performance of `mapA1,A2,A3` heterogeneous mappings

The running time given by the simulation (18.3s) is thus close to the analytically derived minimum application run time imposed by the FPGA in `mapA1` (16.5s). The Virtex II 6000 has the resources to fit up to 3 `Pipes`. Since the `Pipes` are the bulk of processing in the VERITAS application, one might want to speed up processing by running more `Pipes` in parallel on the FPGA. To reflect this, `mapA2` and `mapA3` were developed.

As can be seen on the left side of the graph, the same communication bottleneck that applied to `mapA1` still applies to `mapA2` and `mapA3`, and these mappings get the same performance as `mapA1`. In simulations of `mapA2` and `mapA3` where the effective communication bandwidth limitations were eased, communication ceased to be the bottleneck. Recall that the new bottleneck for `mapA1` had been the FPGA. For `mapA2`, with twice as much processing power on the FPGA, the new bottleneck is `proc[2]` instead. The total processing time for `proc[2]` was found to be 10.5s from the X-Sim simulation, and this forms a lower bound for how much time the whole application run takes. The total application run time for `mapA2` was found to be 13.1s, not much higher than the total time spent just on `proc[2]`. In a bandwidth rich scenario, `mapA2` (13.1s) took less time than `mapA1` (18.3s).

The time taken by `proc[2]` in `mapA3` (15.4) was found by the simulation to actually be higher than in `mapA2` (10.5s), possibly due to the effect of more pipes on the processing times of the `Chopper` blocks. When bandwidth was not a limiting factor, `mapA3` (18.4s) thus wound up taking about as much time as `mapA1`. To summarize, the execution times and bottlenecks for the different scenarios are given in Table 6.1. Note that the resource with the highest utilization (i.e., processing time) was determined to be the bottleneck.

	Bandwidth = 640MB		Bandwidth = 10.67MB	
	App. Run Time	Bottleneck	App. Run Time	Bottleneck
`mapA1`	18.3s	`FPGA`	96.3s	`PCI-X`
`mapA2`	13.1s	`proc[2]`	96.5s	`PCI-X`
`mapA3`	18.4s	`proc[2]`	96.7s	`PCI-X`

Table 6.1: Summary of performance results for `mapA1,A2,A3`

The most important thing to note is that all three mappings suffer heavy performance penalties when the effective bandwidth is low. There are two ways to improve their performance. One is to improve the effective communication bandwidth, thus moving right along the performance curve in Figure 6.16. Another method, explored now, is to make better use of the available effective bandwidth by producing less traffic on the PCI-X bus.

Figure 6.17 shows an alternate mapping, `mapB1`, of the VERITAS application configured with a single `Pipe`. In this mapping, the `Chopper` block has been moved to the FPGA. Critically, this maps `Chopper`'s output edge rather than its input edge on PCI-X. This causes the total amount of traffic on the PCI-X IR (sum of data entering and exiting the FPGA) to be reduced by a factor of 8.3$\times$ (18$\times$/2.17$\times$). Once again, `mapB2` and `mapB3` are simply 2 and 3-`Pipe` versions of `mapB1`. The results from simulating these mappings for different communication bandwidths is shown in Figure 6.18. For comparison, the simulation results for `mapA2` are also included.

Immediately, we can see that all three new mappings have good performance even with limited effective communication bandwidth. With 8.3$\times$ less data being transferred over the PCI-X link, the `mapB` mappings are able to operate for most of the different effective communication bandwidths without the communication link becoming a bottleneck.

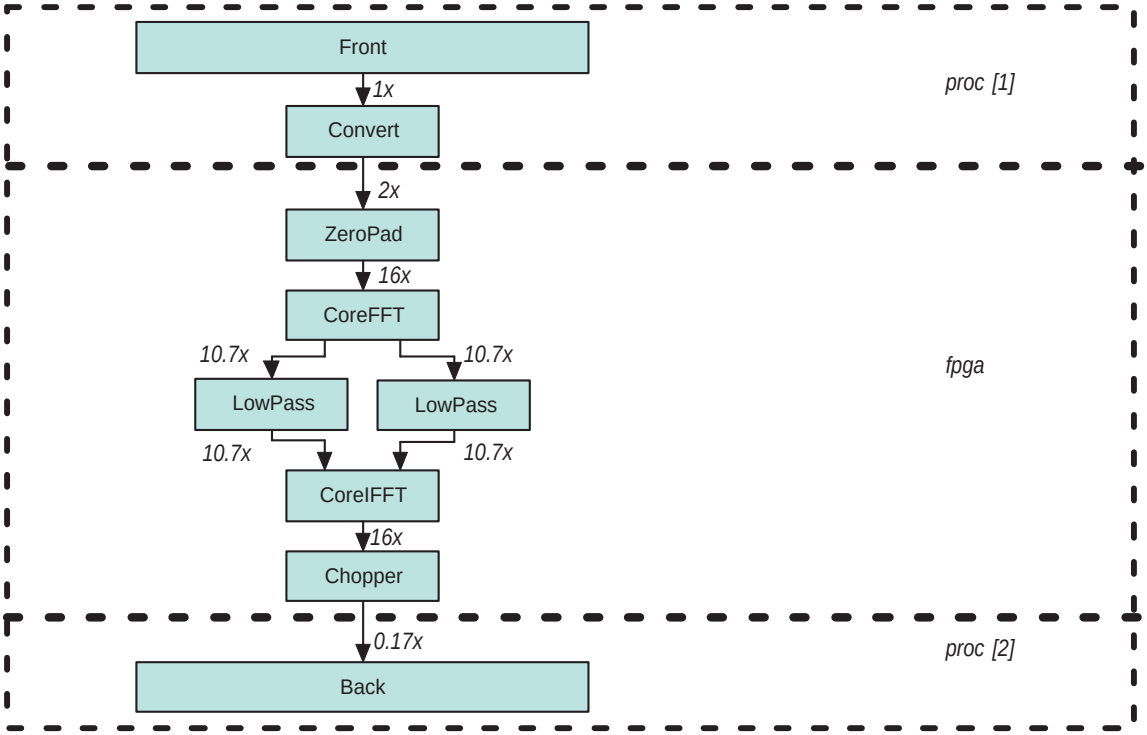


Figure 6.17: `mapB1`: An alternate 1-`Pipe` VERITAS heterogeneous mapping

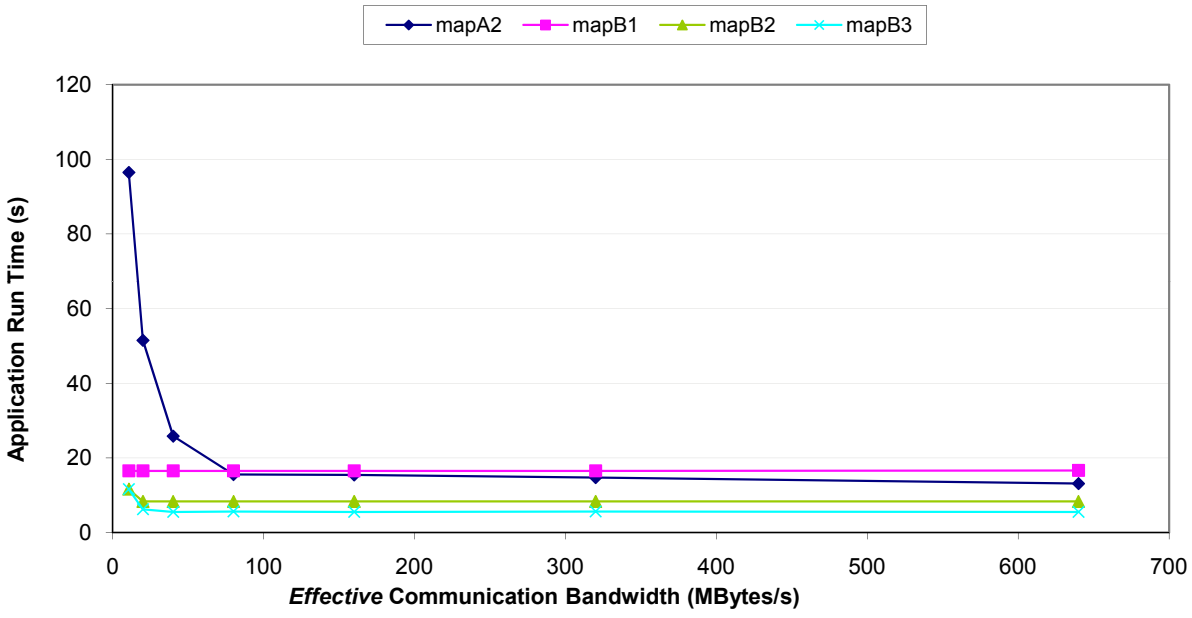


Figure 6.18: Performance of `mapA2,B1,B2,B3` heterogeneous mappings

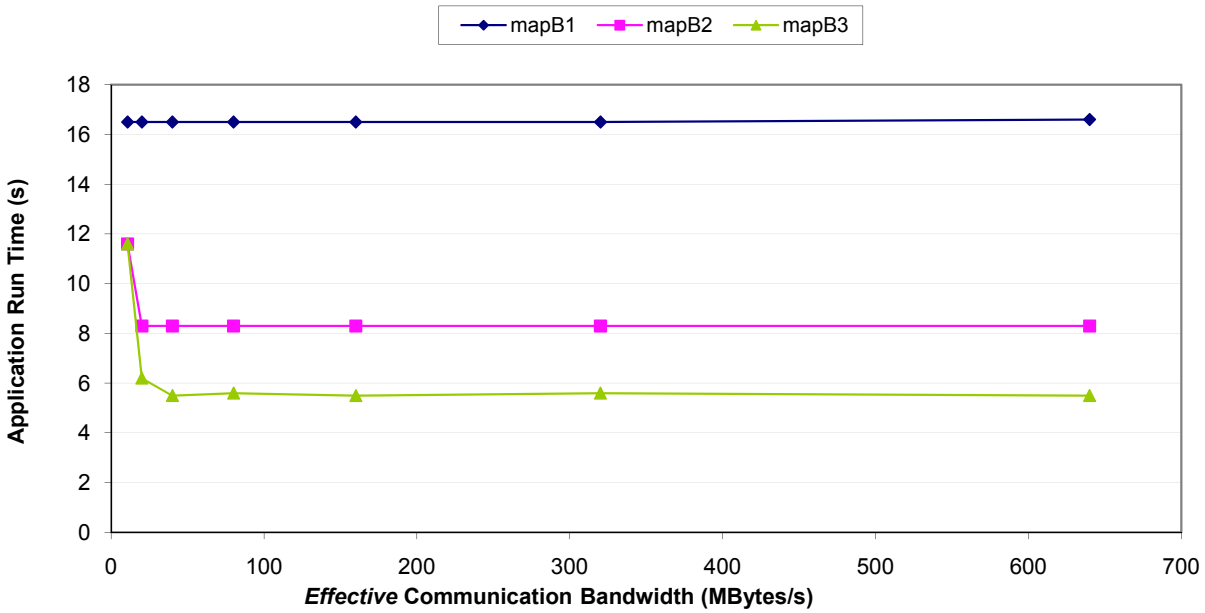


Figure 6.19: Performance of `mapB1,B2,B3` heterogeneous mappings

A close-up of the bottom part of the `mapB` performance graph is shown in Figure 6.19. The simulation times from `proc[2]` showed that `proc[2]` consistently took about 5 seconds total processing time throughout simulations of all the `mapB` mappings. The total application run time thus is lower-bound by 5 seconds. On the right side of Figure 6.19, where communication bandwidth is not a limiting factor, `mapB1`, `mapB2`, and `mapB3` have application run times of about 16.5s, 8.3s, and 5.5s respectively, relatively close to the total time spent processing on the FPGA. Processing times for the analytic model of the FPGA can be determined by the calculations below:

`mapB1` FPGA time = $6.6\mu s \times 499 \times 5000$ = 16.5s

`mapB2` FPGA time = $6.6\mu s \times (499/2) \times 5000$ = 8.2s

`mapB3` FPGA time = $6.6\mu s \times (499/3) \times 5000$ = 5.5s

For `mapB3`, all three `Pipes` can run in parallel, so we only need to analyze one of them. The execution time for a `Pipe` is still determined by the `CoreFFT`'s latency (i.e, $6.6\mu s$). Only a third

of the pixels per event (i.e., 499/3 pixels per event) are processed by this `Pipe`, and there are a total of 5000 events.

An effective communication bandwidth of 10.67MBps, the value from the physical deployment of `mapA1`, resulted in application running times of 16.5s for `mapB1` and 11.6s for `mapB2` and `mapB3`. This compares favorably with the times of about 96.5s for all the `mapA` mappings. Switching from `mapA` to `mapB` and mapping the `Chopper` block(s) to the FPGA brings substantial performance benefits, mostly because much less data is transferred over the limited effective communication bandwidth on the PCI-X IR. A summary of the results from simulations involving `mapB1`, `mapB2`, and `mapB3` are shown in Table 6.2 and can be compared to the results presented previously in Table 6.1.

	Bandwidth = 640MB		Bandwidth = 10.67MB	
	App. Run Time	Bottleneck	App. Run Time	Bottleneck
`mapB1`	16.6s	`FPGA`	16.5s	`FPGA`
`mapB2`	8.3s	`FPGA`	11.6s	`PCI-X`
`mapB3`	5.5s	`FPGA`	11.6s	`PCI-X`

Table 6.2: Summary of performance results for `mapB1,B2,B3`

Assuming the effective communication bandwidth stays the same, switching from `mapA1` to `mapA2` or `mapA3` is not likely to significantly improve performance much. Switching from `mapA1` to `mapB1` is likely to improve performance substantially, from 96.5s to 16.5s, *if the effective communication bandwidth stays the same*. It is possible that the effective bandwidth for `mapB1` might be less since it transfers smaller sized data at a time across the PCI-X bus than `mapA1`. However, even if the effective bandwidth halves, a speedup over `mapA1` can be expected.

Switching to 2 `Pipes` with `mapB2` (11.6s) from `mapB1` (16.5s) may result in a slight speedup as the bottleneck switches from the FPGA to the IR (again assuming constant effective bandwidth). However, adding a third `Pipe` is unlikely to improve performance because the bottleneck is now the IR and not the FPGA. It follows that adding even more `Pipes` in an attempt to parallelize the VERITAS application, for instance by employing a larger FPGA such as the Virtex 5 LX330, is not likely to improve performance because the FPGA is simply not the bottleneck.

So far, we have not made any mention of `proc[1]`'s processing time in any of the simulations. This is because for all the simulations, `proc[1]`, with `Front` always mapped to it, always took about 5.0s to run. Thus, it was never a bottleneck in any of the simulations run so far. It does however, represent a lower limit on how fast the mappings we have considered so far can run. It is interesting to note that `mapB3`, with 3 `Pipes`, came close (5.5s) to this when unconstrained by bandwidth limitations. The `proc[1]` run time is another limitation that argues against attempting to map more than three `Pipes` to the FPGA.

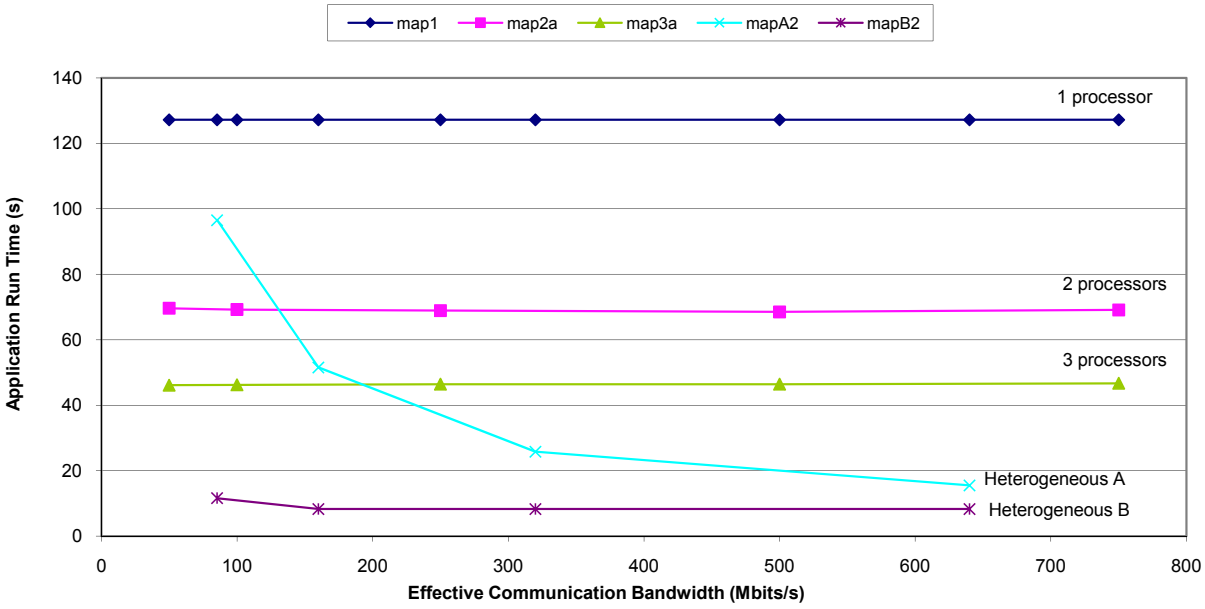


Figure 6.20: Comparison of processor-only and heterogeneous mappings

As a final comparison, Figure 6.20 shows the run times found for the 0-800Mbits/s range for mappings targeted to processor-only architectures (`map1`, `map2a`, `map3a`), as well as for mappings targeted to heterogeneous architectures using both processors and an FPGA (`mapA2`, `mapB2`). FPGAs have the advantage that computations such as FFTs can be done much faster on them than on general purpose processors. The main issue with using FPGAs is to make sure that the communication bandwidth does not become a bottleneck. This can be done by adopting smarter mappings (e.g., `mapB2` instead of `mapA2`), as well as by increasing the amount of effective communication bandwidth available.

Note that *effective* communication bandwidth has been an important concept throughout the simulation and analysis of the VERITAS application. This is a reasonable assumption when the bandwidth of the IR is the main factor affecting its performance. However, other factors can affect the communication link's performance too. For example, there might be a high per-transfer overhead associated with a particular IR. To model this, a fixed overhead delay can be added to the delay for a simulated transfer over an edge. In cases where this per-transfer overhead is high, better performance can be gained by reducing the total number of transfers done, even if the total amount of data transferred stays the same. For the VERITAS application, this could be achieved by modifying the algorithm so that multiple `Pixel` elements are grouped together before being transferred over the `PCI-X` link. Discovering performance characteristics of particular IRs requires additional experimentation with deployed applications on the physical IRs.

Throughout discussions in this section, we have seen how factors such as effective communication bandwidth, application parallelization, and mapping choices affect the performance that can be expected from different mappings of an application. Simulating different scenarios and analyzing the results can help the user, and in the future X-Opt, make intelligent decisions in optimizing high performance streaming applications such as VERITAS.

Chapter 7

Summary

7.1 Conclusion

The Auto-Pipe toolset is useful for developing streaming parallel applications that are mapped to heterogeneous architectures. Within this toolset, X-Sim provides a mechanism to run simulations and gather results. It automates the task of simulating different parts of the application in a federated manner, creating full data and timing traces of application executions where specified. Data traces can be inspected to debug the application. Timing traces can be used to analyze the performance of the application.

X-Eval provides a mechanism to analyze timing traces generated by an X-Sim simulation run. It generates a timeline visualization that allows the user to grasp the relative timing of various operations in the application. Additionally, X-Eval provides summary performance metrics like average execution time and average waiting time for blocks and computational resources.

X-Sim makes use of various techniques to speed up simulations, useful where many successive simulations must be run, as well as where individual simulation runs are lengthy. This thesis presented various test cases to demonstrate the use of X-Sim and X-Eval in simulating and analyzing streaming applications. It also presented validation of results, where possible, against physically deployed applications.

7.2 Contributions and Implementation Status

Contributions presented in this thesis include:

- X-Sim, a federated simulation tool that conveniently and efficiently simulates streaming applications mapped to heterogeneous architectures
- X-Eval, an analysis tool that can be used to analyze timestamps generated by X-Sim
- simulation speedup techniques
- two sample applications, `test1` and VERITAS, with detailed simulation, analysis, and validation

X-Sim is operational and has been extensively used by multiple users successfully. Validation of native execution simulations has been carried out for large applications (e.g. VERITAS) on Symmetric Multi Processing (SMP) systems using shared memory. Validation needs to be done for other target systems.

Testpoints are currently only supported for `C` implementations of blocks. Support for testpoints should also be added for `VHDL` implementations.

X-Eval is operational and has been used to generate timelines as well as execution time metrics for various applications. The analytic modeling of computational resources currently only works for one input edge and one output edge. The analytic modeling needs to be updated to support multiple input and output edges.

7.3 Future Work

Support for the simulation of additional resources, e.g. for Graphics Processing Units (GPUs) and Network Processors (NPs), need to be added to X-Sim as these resources are added to the Auto-Pipe framework.

Currently X-Sim runs each federate simulator in isolation, reading all the required data and timestamps from trace files at the beginning of simulation, and writing all the generated data and timestamps to trace files at the end of simulation. X-Sim keeps *all* input and output data and timestamps in memory while running the simulation, so that file access does not affect the native execution performance. This has obvious drawbacks in terms of working memory requirements. Techniques should be investigated on how to get around finite memory resources. For example, one possible technique is to 'pause' an internal simulation clock, dump out all the output data and timestamps collected so far, and then resume the simulation. That particular technique has the drawback that file-access mid-simulation can alter disk and chip cache, and thus affect the native execution performance.

X-Opt needs to be developed more concretely. This will help pin down the exact requirements in the automated optimization step, and thus guide future requirements and development directions for X-Eval.

The user must currently manually write the semantics file that is used as an input by X-Eval. A modified version of X-Dep should be developed that generates a skeleton semantics file based on the X Language description of the application.

The current communication model is a fixed delay per edge model, where the fixed delay must be assigned by the user manually. This can be streamlined by a system where the user simply assigns a fixed bandwidth to an Interconnect Resource (IR), and X-Sim automatically figures out the delay associated with each transfer on each edge on that resource. X-Sim can run an initial simulation to find out the total traffic on each edge, and allocate bandwidth proportional to traffic to each edge. The delay on an edge can then be calculated by dividing the size of a transfer by the allocated bandwidth.

A library of communication performance models must be built up with rigorous experimentation with X Language applications on different physical communication links. This will allow communication simulation in X-Sim to be done with empirical numbers that have been validated, and thus allow more accurate performance results to be generated for application mappings to a wide variety of architectures.

References

[1] Jung Ho Ahn, William J. Dally, Brucey Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the imagine stream architecture. In *Proc. 31st Annual International Symposium on Computer Architecture (ISCA'04)*, page 14. IEEE Computer Society, 2004.

[2] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20:519–526.

[3] James O. Calvin and Richard Weatherly. An introduction to the high level architecture HLA runtime infrastructure RTI. In *14th Workshop on Standards for the Interoperability of Distributed Simulations*, 1996.

[4] Roger D. Chamberlain, Stever Miller, Jason White, and Dan Gall. Highly-scalable reconfigurable computing. In *Proc. of 8th Military and Aerospace Programmable Logic Devices International Conference*, September 2005.

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

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

[7] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, et al. Ptolemy II: Heterogeneous concurrent modeling and design in java. Technical Report Memorandum UCB/ERL M99/44, University of California, Berkeley, July 1999.

[8] DMSO. Defense Modeling and Simulation Office. `www.dmso.mil`.

[9] Mark A. Franklin, Eric J. Tyson, James Buckley, Patrick Crowley, and John Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.

[10] Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, and Roger D. Chamberlain. A federated simulation environment for hybrid systems. In *Principles of Advanced and Distributed Simulation*, 2007.

[11] GNU. The GNU Compiler Collection. `http://gcc.gnu.org`.

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

[13] R. Jagannathan and A.A. Faustini. The GLU programming language. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1990.

[14] Frederick Kuhl, Richard Weatherly, and Judith S. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.

[15] Mentor Graphics Corp. ModelSim. `http://www.model.com`.

[16] Michael Taylor. btl debugging. `http://cag.csail.mit.edu/raw/memo/19/btl-debug.html`.

[17] NI. National Instruments. `http://www.ni.com/`.

[18] The Ptolemy Team. The ptolemy kernel - supporting heterogeneous design. *RASSP Digest Newsletter*, 2(1):14–17, April 1995.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: a language for streaming applications. *Proc. Inter. Conf. on Compiler Construction*, April 2002.

[20] Eric 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.

[21] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, Amarsinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(1):86–93, September 1997.

[22] T. C. Weekes, H. Badran, S. D. Biller, I. Bond, S. Bradbury, J. Buckley, D. Carter-Lewis, M. Catanese, S. Criswell, and W. Cui. VERITAS: the very energetic radiation imaging telescope array system. *Astroparticle Physics*, 17(2):221–243, May 2002.

Vita

Saurabh Gayen

Date of Birth	November 26, 1983
Place of Birth	Dhaka, Bangladesh
Degrees	B.S. Computer Science and Computer Engineering, December 2007 M.S. Computer Engineering, May 2008
Professional Societies	Institute of Electrical and Electronics Engineers (IEEE)
Publications	Roger D. Chamberlain, Eric J. Tyson, Saurabh Gayen, Mark A. Franklin, Jeremy Buhler, Patrick Crowley, James Buckley. Application Development on Hybrid Systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC07)*. November 2007. Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, Roger D. Chamberlain. A Federated Simulation Environment for Hybrid Systems. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS '07)*. May 2007. Saurabh Gayen, Eric J. Tyson, Mark A. Franklin, Roger D. Chamberlain, Patrick Crowley. X-Sim: A Federated Heterogeneous Simulation Environment. In *Proceedings of the 10th High Performance Embedded Computing (HPEC) Workshop*. September 2006. Roger Chamberlain, John Lockwood, Saurabh Gayen, Richard Hough, Phillip Jones. Use of a Soft-Core Processor in a Hardware/Software Codesign Laboratory. In *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education (MSE '05)*. June 2005.

May 2008