2006-50

# Auto-Pipe and the X Language: A Toolset and Language for the Simulation, Analysis, and Synthesis of Heterogeneous Pipelined Architectures, Master's Thesis, August 2006

Authors: Eric J. Tyson

Corresponding Author: etyson@wustl.edu

Abstract: Pipelining an algorithmis a popularmethod of increasing the performance of many computation-intensive applications. Often, one wants to form pipelines composed mostly of commonly used simple building blocks such as DSP components, simple math operations, encryption, or pattern matching stages. Additionally, one may desire to map these processing tasks to different computational resources based on their relative performance attributes (e.g., DSP operations on an FPGA).

Auto-Pipe is composed of the X Language, a flexible interface language that aids the description of complex dataflow topologies (including pipelines); X-Com, a compiler for the X Language; X-Sim, a tool for modeling pipelined architectures based on measured, simulated, or derived task and communications behavior; X-Opt, a tool to optimize X applications under various metrics; and X-Dep, a tool for the automatic deployment of X-Com- or X-Sim-generated applications to real or simulated devices.

This thesis presents an overview of the Auto-Pipe system, the design and use of the X Language, and an implementation of X-Com. Applications developed using the X Language are presented which demonstrate the

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

# AUTO-PIPE AND THE X LANGUAGE:

# A TOOLSET AND LANGUAGE FOR THE SIMULATION, ANALYSIS, AND SYNTHESIS OF HETEROGENEOUS PIPELINED ARCHITECTURES

by

Eric J. Tyson

Prepared under the direction of Professor Mark A. Franklin

---

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

ABSTRACT

---

# AUTO-PIPE AND THE X LANGUAGE: A TOOLSET AND LANGUAGE FOR THE SIMULATION, ANALYSIS, AND SYNTHESIS OF HETEROGENEOUS PIPELINED ARCHITECTURES

by

Eric J. Tyson

---

ADVISOR: Professor Mark A. Franklin

---

August 2006

Saint Louis, Missouri

---

Pipelining an algorithm is a popular method of increasing the performance of many computation-intensive applications. Often, one wants to form pipelines composed mostly of commonly used simple building blocks such as DSP components, simple math operations, encryption, or pattern matching stages. Additionally, one may desire to map these processing tasks to different computational resources based on their relative performance attributes (e.g., DSP operations on an FPGA).

Auto-Pipe is composed of the X Language, a flexible interface language that aids the description of complex dataflow topologies (including pipelines); X-Com, a compiler for the X Language; X-Sim, a tool for modeling pipelined architectures based on measured, simulated, or derived task and communications behavior; X-Opt, a tool to optimize X applications under various metrics; and X-Dep, a tool for the automatic deployment of X-Com- or X-Sim-generated applications to real or simulated devices.

This thesis presents an overview of the Auto-Pipe system, the design and use of the X Language, and an implementation of X-Com. Applications developed using the X Language are presented which demonstrate the effectiveness of describing algorithms using X, and the effectiveness of the Auto-Pipe development flow in analyzing and improving the performance of an application.

# Contents

# List of Tables

# List of Figures

# Acknowledgments

First and foremost I would like to thank my family and friends for all their encouragement during the writing of this thesis.

Thanks to my mom and dad, who have always supported the directions I've taken in life with great enthusiasm.

Thanks to my friends, who have enriched my life with all the little distractions from work that keep me sane.

Thanks to my colleagues in the Storage Based Supercomputing group, and throughout the Computer Science and Engineering and Physics departments, for discussions which have been invaluable in writing this thesis, the compiler, and the science applications. Thanks in particular to my advisor, Mark Franklin, and Roger Chamberlain, for their feedback in making this thesis both useful and usable.

Finally, I am grateful for the financial support of the National Science Foundation through grant CCF-0427794, without which this and other valuable research would not have been possible.

Eric J. Tyson

*Washington University in Saint Louis*
*August 2006*

# Chapter 1

# Introduction

This thesis introduces Auto-Pipe, a toolset and development environment that aids the creation of applications developed in a pipelined and/or parallel manner. Tools are provided to analyze performance and explore the many options in the design space. These options include, for example, the use of both real and hypothetical processing devices and connection topologies.

The X Language has been created to express the applications that use the Auto-Pipe tools. Analyses of these applications are performed with the X-Sim tool, a federated simulation environment which uses the X Language compiler. An optimization tool tentatively called X-Opt is proposed to find optimal mappings of algorithms onto complex sets of interconnected processing platforms and explore the design space. The capabilities and use of the X Language, X-Sim, and X-Opt are presented herein.

This chapter investigates the background and motivation for creating a new language and a new toolset. A set of prior work in this area is reviewed, including projects with similar techniques but dissimilar goals, some of which are complementary to the Auto-Pipe system. The design of the Auto-Pipe system and contributions thus far are discussed broadly as an introduction to the detailed discussion of the X Language and compiler found in the following chapter. An outline of the thesis is provided at the end of the chapter.

## 1.1 Background

Auto-Pipe is a development environment concentrating on high-performance applications. Auto-Pipe aids developers in creating and understanding applications distributed across multiple, potentially dissimilar computational nodes and interconnection resources.

Such applications are easily found in many areas of science and industry, including:

- Networking and communications
- Scientific computing, including real-time experimentation and offline data analysis
- Media creation and playback
- Data mining

The process of creating an efficient implementation of a given application can be broken into the following components:

1. Develop the algorithm:
   (a) Identify the tasks involved in the computation and program control.
   (b) Determine the communication structure of these tasks.
   (c) Implement (e.g., in C, C++, VHDL, etc.) the tasks as needed.
   (d) Evaluate the implementation's correctness by executing the algorithm on a simplified platform, such as a development machine.
2. Identify the *computation resources* (CRs) on which the application tasks may be deployed.
3. Identify potential communication topologies of these resources, specifically the *interconnect resources* (IRs) connecting sets of CRs.
4. Determine a *mapping* of the application tasks and communications onto the CRs and IRs. This mapping must be valid, thus it must:

- map all tasks and communications to compatible CRs and IRs in the topology,
- not exceed the resource limitations of any CR or IR, and
- map tasks in a manner that satisfies algorithm requirements for correctness.

5. Implement the mapping, creating the code to connect the tasks using the available IRs.

6. Execute the implementation and analyze the performance. Execution here may take place on real CRs and IRs, simulations of CRs and IRs, or a combination of both.

7. As needed, improve the application's performance based on the analysis of 6 by redoing the mapping in step 4, and repeating the subsequent steps.

As will be described in Section 1.3, the X Language enables expression of steps 1 through 4 above. Auto-Pipe tools then perform the remaining steps. The following discussion will examine, at a high level, the process described above.

Dataflow languages are languages that express an algorithm in terms of the flow of data between connected tasks with precise interfaces. The algorithm portion of the X Language can be considered as a dataflow language. Tasks in such a language can form a partially ordered set of data dependencies. This set is naturally visualized as a directed graph.

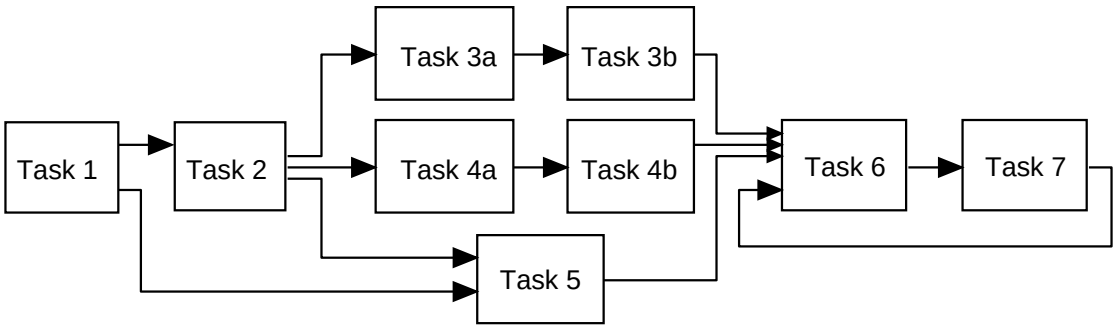



Figure 1.1: An example application dataflow

An example of such a directed graph is shown in Figure 1.1. This graph can be used to determine valid task execution schedules — for instance, Task 2 cannot run until Task 1 has completed, and Task 5 will depend on the output of *at least* one of Task 1 or Task 2, and

possibly both. To completely understand the algorithm, though, additional information is needed. The semantics of both tasks and edges must be known — does Task 1 produce data on both of its outputs, or just one or the other? And does Task 6 consume the three forward inputs at the same time, or individually? Furthermore, the *feedback loop* edge that connects Task 7 to Task 6 can clearly present a scheduling challenge, and may even introduce the risk of deadlock.

In this thesis, only *feed-forward* pipelines are considered. More advanced techniques may be employed to deal with topologies containing feedback loops, however these will not be presented in this thesis. A feed-forward pipeline is a sequence of tasks that contains only forward-directed edges; no edges may connect to tasks occuring earlier in the sequence. These pipelines may be expressed as acyclic directed graphs.

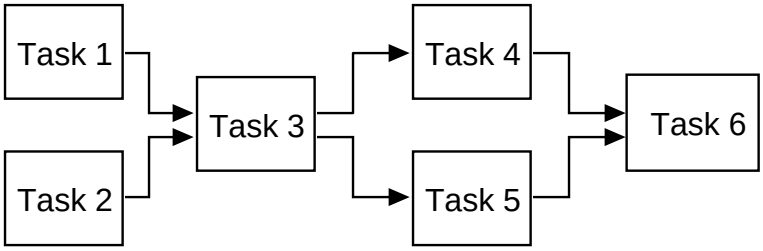



Figure 1.2: An example feed-forward pipeline

Figure 1.2 shows a simple pipeline of six tasks that will be used as an example. The process of determining what tasks 1 through 6 do, and how they are connected, is equivalent to steps 1(a) and 1(b) in the procedure from page 2. To create an executable application from this pipeline, the *processing architecture* must also be determined.

Processing architectures are graphs of connected *computation resources* (CRs). A CR is a generic term for a device that is capable of performing tasks, and includes general purpose processors (e.g., the Intel x86 series), embedded processors, network processors, field-programmable gate arrays (FPGAs), digital signal processors, and others. Identifying the CRs on which the user may potentially place the algorithm constitutes step 2 of the overall procedure.

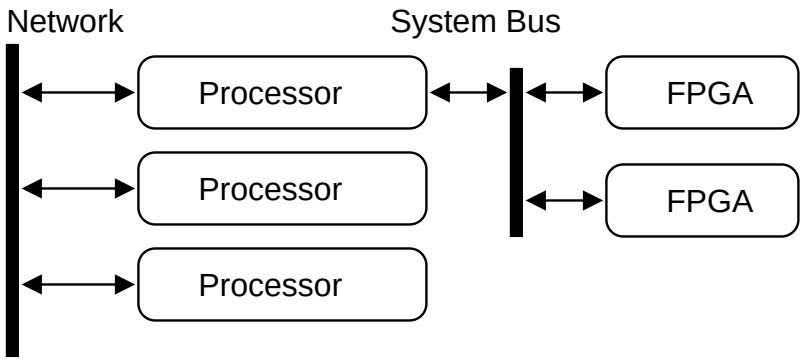



Figure 1.3: An example processing architecture

Also important to the behavior of a processing architecture are the connections, or *interconnect resources* (IRs). Identifying the IRs corresponds to step 3 of the application development process. IRs can be one-way or bidirectional, and can be point-to-point or shared. Sometimes processors are not directly connected to each other, but rather communication must be forwarded over one or more other processors. Additionally, real-world connection fabrics can be almost arbitrarily complex when resource sharing, non-uniform access, non-deterministic performance (such as the Internet), and other behaviors are taken into account.

An example of a processing architecture is given in Figure 1.3. In this figure, three processors share a switched network. Another two CRs, FPGAs, are connected to one of the processors using a system bus. For this example, it will be assumed that the processors are roughly equivalent when performing the same operations, while the FPGAs may have very different performance for the same task.

*Mapping* is the operation of assigning a specific task to a CR, or a specific interconnecting edge to an IR. A complete mapping of an application involves performing this step for every task and edge. In order to implement and execute an application, a complete mapping must be performed. Determination of a mapping is step 4 of the development procedure and completes the X Language program description.

The choice of a "good" mapping can become quite difficult, both in a computation sense (the minimum parallel processor total flow time problem is NP-Hard [19]) and in choosing the proper method and performance metric to evaluate what constitutes a "good" mapping. In general, a good mapping choice cannot simply consider the best resource available for a single task, due to a large number of conflicting factors such as:

- Relative performance of each task on different resources
- Relative performance of the different interconnections
- Resource contention within the processing resources
- Resource contention within the interconnection resources

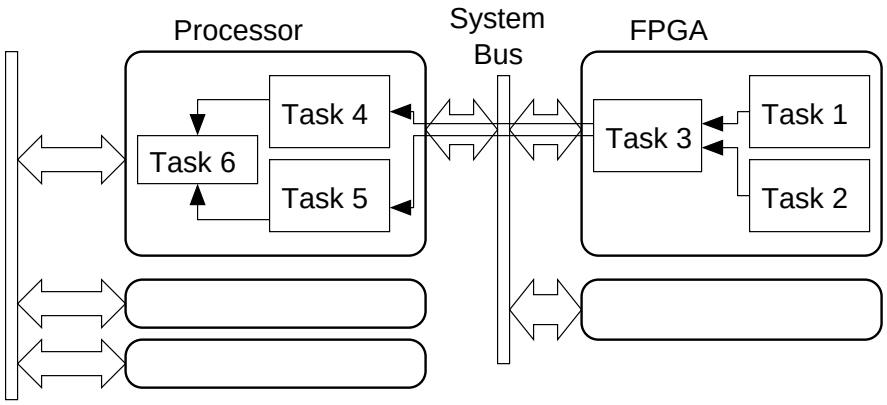



Figure 1.4: An example of mapping the algorithm to the processing architecture

Figure 1.4 demonstrates one potential mapping of the example application onto the example architecture. In this example, tasks 1, 2, and 3 have been mapped to one FPGA, and tasks 4, 5, and 6 have been mapped to the first processor. This mapping causes task 3 to communicate with tasks 4 and 5 over the system bus, while other task communications take place within the processor; in an FPGA, this might be a simple handshaking protocol, and on a processor this could be a function call or a job queue.

Figure 1.5 demonstrates another possible mapping of the same application and architecture. In this example, the tasks have been distributed more evenly across the processors.

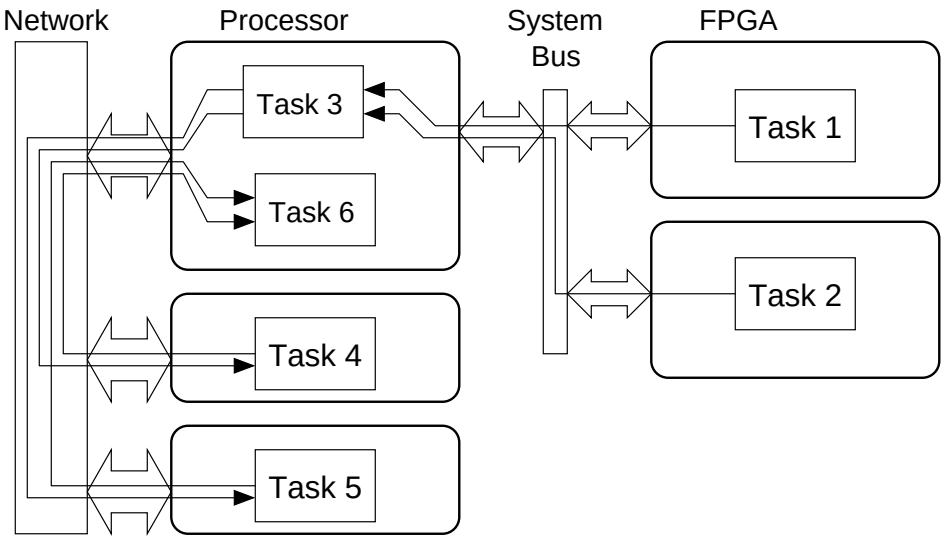



Figure 1.5: Another example of mapping the algorithm to the processing architecture

Depending on their relative performance, this may cause the tasks to execute more efficiently than in the first mapping; execution of tasks 4 and 5 have been distributed to two processors, and there is more potential for parallel task execution.

Note that the second mapping may not actually perform better than the first, even with better distribution of computation since there is much more communication over the network interconnect. Communication overhead, especially over the shared link, may offset the faster computation. This demonstrates a common tradeoff that occurs in parallel computing — the balancing of algorithmic parallelism versus the reduction of communications overhead. Determining which mapping results in higher overall performance requires consideration of the overall system performance including the performance of each task and communication link in the system. This analysis is performed in step 6 (page 3), and the results are used for step 7, which regenerates the mapping to improve performance.

A second level of resource association called *binding* is considered later. Binding concerns the deployment of CRs and IRs onto particular real-world devices (or simulations/emulations of such devices). Mechanisms to bind and deploy are not examined in depth in this thesis; our approach to this operation will be briefly discussed in Chapter 3.

The performance of streaming applications are generally evaluated in terms of two metrics: *latency*, the time from input to output of time-sensitive data, and *throughput*, the quantity of data processed per unit time. Other metrics such as power consumption and total implementation costs are also used. This thesis concentrates primarily on maximizing throughput, which is often the most important metric in high-performance applications without strict latency requirements.

## 1.2 Motivation

Auto-Pipe was created to overcome several difficulties in developing applications that are distributed across diverse computational nodes, particularly those that benefit from being decomposed into pipelines and other parallel structures.

The difficulties addressed by Auto-Pipe include:

- Implementing an application expressed in both hardware and software programming languages (often referred to as hardware/software *co-design*), or multiple software languages, is awkward and error-prone.
- Debugging and simulating such a mixed infrastructure (*co-simulation* or federation simulation) to ensure functional correctness.
- Developing efficient communication interfaces to transport data between CRs. This is tedious, time-consuming, and often unnecessarily repeats work done by others.
- Measuring the implications of different CR partitionings, performance-area tradeoffs, performance-memory tradeoffs, and various mappings. This is generally time-consuming and lacks sufficient tool support.
- Optimizing system performance over a large design space.

Due to these obstacles, designers are presently faced with a set of choices:

- Put a large amount of effort into developing, debugging, and optimizing the application across multiple platforms: This approach requires time, money, and programmers skilled in developing and optimizing for each of the platforms used.
- Limit the application to only general purpose CPU platforms for which development tools are available and which tend to be less expensive and easier: This results in an implementation that may not fully exploit the speed and unique performance capabilities of general purpose processors, programmable hardware (e.g. FPGAs), and special-purpose CPUs (e.g. DSPs).
- Limit the simulation and optimization of the co-designed system: This also may reduce the performance of the application. Developers choosing this option are often left with a system having unpredictable performance, and for which development efforts may concentrate on the wrong set of components.

In each of these choices, the developer ends up with a limited implementation of the application that cannot be easily extended to new processing platforms.

Auto-Pipe is an option that provides developers with a rich set of development and analysis tools to overcome these obstacles. It also provides an infrastructure to easily add support for new platforms and efficient communications links.

## 1.3 The Auto-Pipe System

The Auto-Pipe system and its components were created to aid developers of streaming applications. Auto-Pipe comprises a set of tools, compiler-level libraries, and user-level libraries to create an environment for the development of individual applications.

The three primary tool components are discussed below: the compiler, the simulation environment, and the performance optimizer. The X Language and compiler are the main subject of this thesis; the simulator and optimizer are currently under development and only their preliminary design is introduced here.

### 1.3.1 The X Language and Compiler

The primary means of expressing the programs created and analyzed using Auto-Pipe is the X Language. While its compiler, X-Com, does not depend on other Auto-Pipe tools, X has been developed specifically for the Auto-Pipe system and application development is greatly enhanced by use of its tools. The X Language is a coarse-grained dataflow coordination language where the actual tasks are expressed and executed utilizing other traditional and non-traditional programming languages. X programs describe a hierarchy of tasks, called *blocks*, and the interconnections of their interfaces, called *edges*. The language also provides a syntax to describe classes of computation resources on which tasks are to eventually execute and to describe the specific instances of these resources. A block-to-CR *mapping* may then be specified to indicate on which resources the tasks are to execute. The X Language has syntax to place portions of the application on different CRs, and easily change this allocation in order to explore the performance of the system under different partitionings.

As part of X-Com, an internal interface has been developed to connect modules that perform code generation. This interface is easily extended so that users can target new devices and languages without changing the dataflow synthesis and analysis details that form the core of X-Com and other tools.

Application programmer interfaces (APIs) have been created for a variety of programming environments to allow users to create their own blocks which can be used by X. Particular attention has been paid to the usability of these interfaces, as it is expected that most users of X and Auto-Pipe will want to create their own blocks for application-specific tasks (such as sensor or database I/O).

The output of X-Com is a set of "almost executable" source code files corresponding to each CR in the processing architecture. A second tool, X-Dep, the X application deployer, is used to perform the final linking steps and deploy the application to real hardware devices or simulations (or emulations) of devices. This tool does not affect the actual behavior of the

generated executable; it only makes the X-Com-generated code deployable to its intended real target. X-Dep is discussed further in Section 3.2.

### 1.3.2 The X-Sim Simulation Environment

Another core component of the Auto-Pipe toolset is X-Sim, a simulator for X applications partitioned across multiple simulated devices. X-Sim incorporates software and hardware simulations (of various fidelity) to characterize the performance of the system and of individual blocks, edges, and interconnects. These characteristics can be interpreted by the designer (using analysis tools) to locate performance bottlenecks and motivate design decisions. X-Sim also permits the testing of hypothetical devices and topologies to further drive the design process.

X-Sim comprises a set of steps surrounding the X compiler. When performing an X-Sim simulation, "simulation bindings" are passed to X-Com that indicate which components are to be simulated, along with the type of simulator and potential simulation options. For instance, the C generator might support simulation using either native processor execution or execution under SimpleScalar [5], where the latter could be passed options indicating what degree of simulation fidelity is desired. The output of X-Com is a set of code that may be directly compiled, cross-compiled, or specially linked (depending on the requirements of the simulator) through binding.

The code generated by the simulation-bound applications uses a simulation-mode interconnect mechanism. This mechanism uses an intermediate data file to store the data transmitted along an edge; this common format permits communication between different simulators. Depending on the simulation environment, it may also generate a file containing processing times and potentially other run-time information (e.g., instantaneous power, cache misses, etc.). After each intermediate data set is written, X-Sim uses a communication model corresponding to the edge's IR to model the time delay before the data is available to the receiving block. A second timestamp file is generated that reflects the transmission delay, and the data and timestamps are then provided to the simulator of the

receiving block. Since simulators of widely varying quality are supported, the simulator might or might not actually make use of the timestamps.

X-Sim is used to gather performance statistics and processing characteristics for each simulated device, to better understand the execution of the application. X-Sim will also be used with a performance analysis tool to further examine and optimize the data flow. This analysis is then used with the third major component of Auto-Pipe, the X-Opt performance optimizer, which is described in the following section.

### 1.3.3 The X-Opt System Performance Optimizer

The final main piece of the Auto-Pipe toolset is X-Opt, an optimizer for X applications. Given a feed-forward pipelined, streaming application, X-Opt will use a combination of the simulator and a throughput (or latency) optimization algorithm to find optimal or near-optimal task-to-CR and CR-to-device allocations.

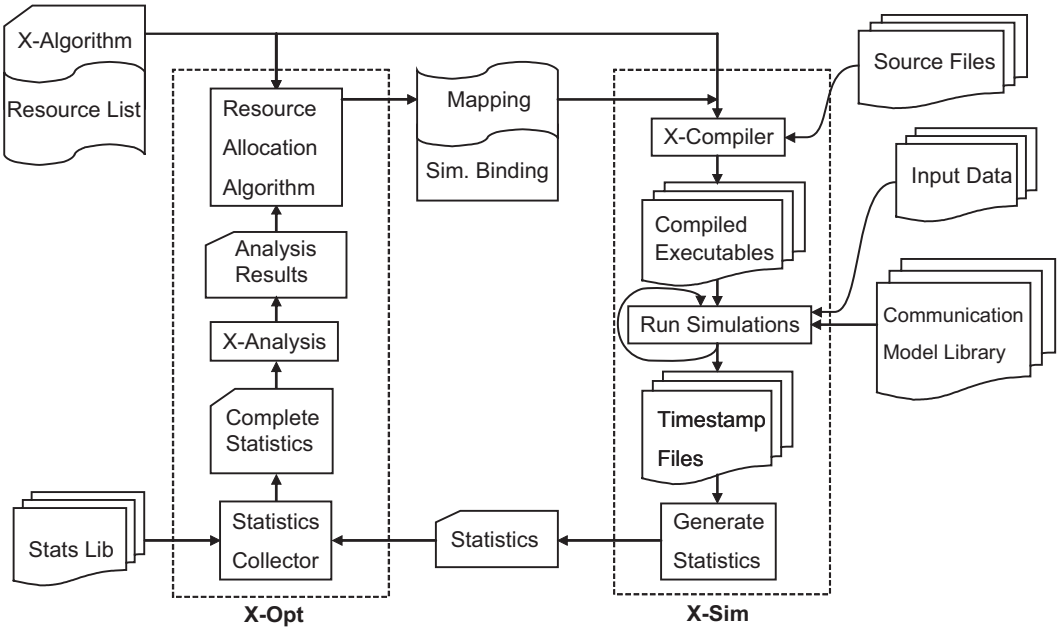



Figure 1.6: X-Opt Flow Chart

Figure 1.6 diagrams the design of X-Opt and its connection with X-Sim. X-Opt (or the user) creates an initial mapping and binding of blocks and edges to devices which are then simulated using X-Sim. X-Opt is responsible for ensuring that this mapping-binding is valid, with all blocks and edges properly mapped to resources. After gathering and analyzing statistics for this mapping, X-Opt revises its mapping-binding and repeats the simulation. This continues until a satisfactory result or some other end condition is reached.

This thesis does not deal directly with implementation of X-Opt and X-Sim. To demonstrate overall system capabilities, the "resource allocation algorithm" block for examples presented here utilize a human trial-and-error approach. The goal, however, is to eventually have a largely automated computer algorithm.

### 1.3.4 Auto-Pipe Development Flows

The X Language is open-ended and the Auto-Pipe tools permit the user to approach development of an application in a number of ways. There are four progressively more thorough development procedures or "flows" for the Auto-Pipe system discussed here which most users will likely use or expand upon for their development.

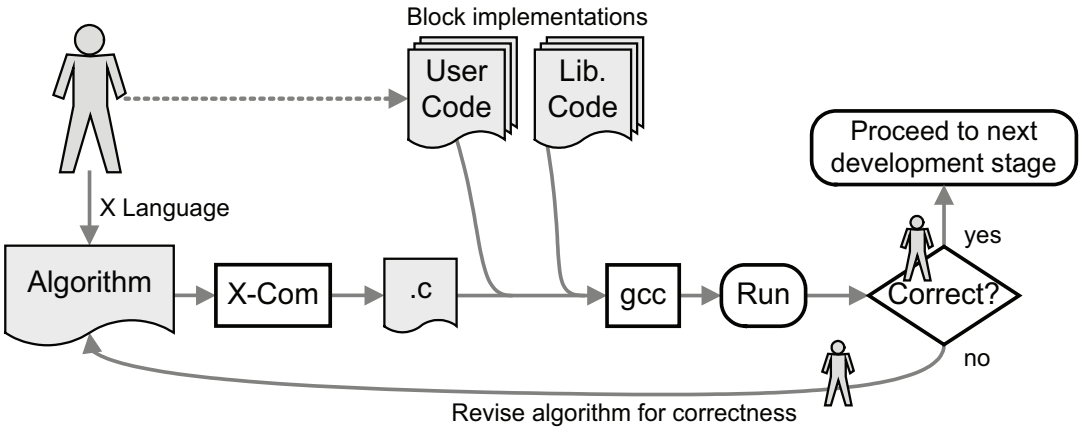



Figure 1.7: X-Com algorithm design flow

The most basic flow is depicted in Figure 1.7. Here, the user first creates an X Language file containing their algorithm partitioned into X blocks which are implemented in C (the

default for testing). This algorithm instantiates blocks from the available libraries and any user-defined blocks that are needed. X-Com processes the file and creates a top-level source file that can be compiled to a single-threaded executable that runs on the native machine. The user examines the output for functional correctness, redesigning the algorithm as necessary, until a implementation has been created that is correct.

Note that this flow does not yet create hardware components, nor does it use multiple computational resources. In general, block implementations are easier to write, test, and debug when written in software using single-threaded execution. Most users will desire to first develop their overall algorithm using this model, before introducing the complexities of hardware device programming and concurrent execution. It may not be desirable to directly deploy the result of this flow, as the generated program does not take advantage of any degree of parallelism. Rather, this is a useful initial testing stage before proceeding further.

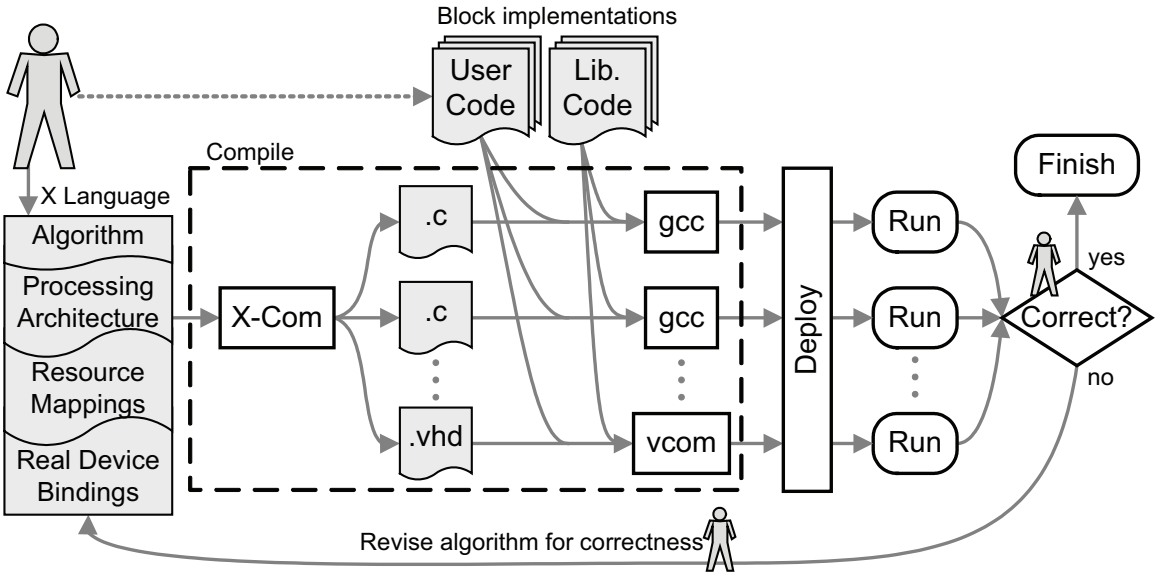



Figure 1.8: X-Com flow

The Figure 1.8 X-Com flow introduces user-selected processing architectures. In this flow, X-Com creates a top-level source file for each CR in an appropriate language (this mechanism is described in Chapter 2). Compilation of the generated source files is done using the appropriate language compiler for the target system, which may necessitate cross-compilation. After compiling, the user can deploy the finished application onto real compute and interconnect resources, execute the distributed application, and examine the output for correctness.

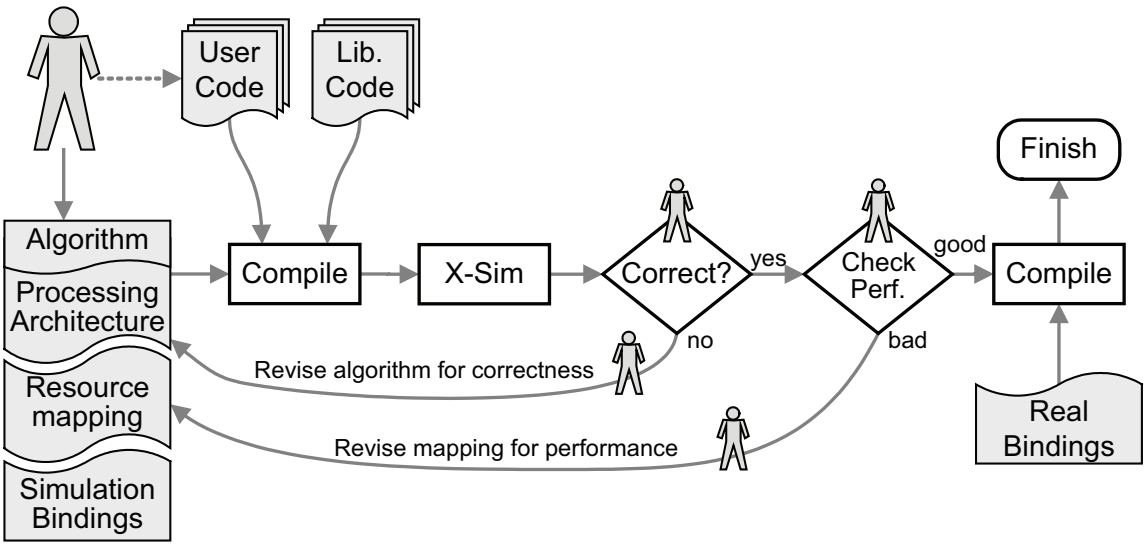



Figure 1.9: X-Com plus X-Sim flow
"Compile" incorporates X-Com and the compiler steps depicted in Figure 1.8

The first performance-improving flow involves the X-Sim simulator, depicted in Figure 1.9. In this flow, compilation proceeds as before, except that the generated programs are bound to simulated environments instead of real devices. This causes X-Com to create simulation-mode source files that enable X-Sim to analyze the communications between blocks and run the executables on simulation platforms (such as SimpleScalar [5] or ModelSim [6]). As before, the generated source files are compiled using their respective compiler, potentially a cross-compiler or hardware synthesis tool. In addition to the correctness check as in the previous flow, the user may also examine the X-Sim performance report, which will give insight into what components of the processing architecture are potential performance bottlenecks. From this information, the user may attempt to revise the mappings to achieve

greater performance, such as by improving the task partitioning. Once performance is satisfactory, the simulation bindings are replaced with their "real" equivalents so that a deployable set of executables are created.

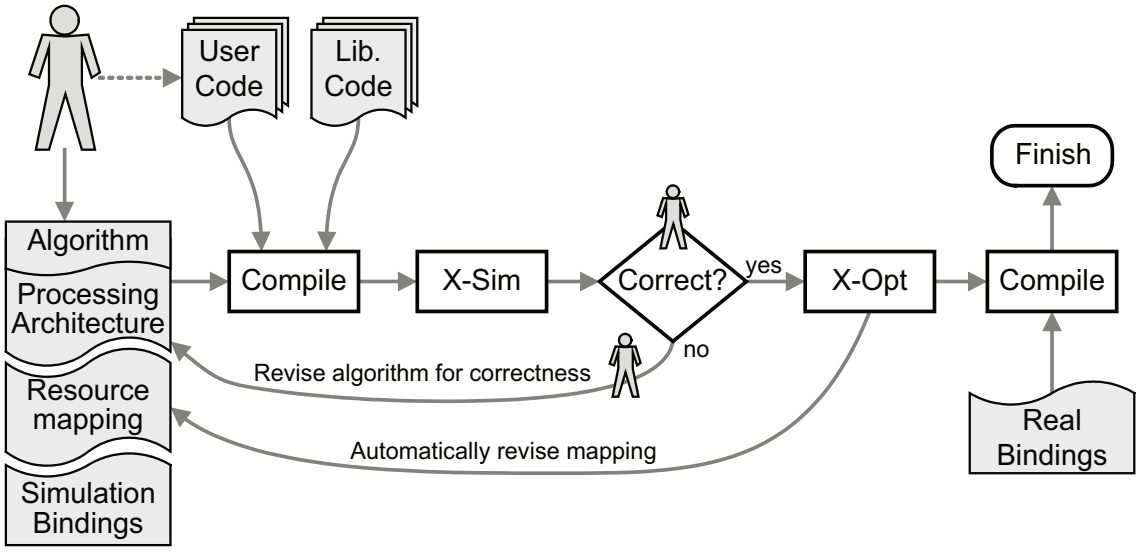



Figure 1.10: X-Com plus X-Sim plus X-Opt flow

Figure 1.10 modifies the flow of Figure 1.9 by replacing the user performance check with the X-Opt tool. X-Opt will explore the design space and be capable of quickly checking different performance characteristics. X-Opt will continually revise the mappings to test different configurations, until an end condition specified by the user is met (such as a minimum performance requirement or maximum exploration time).

## 1.4 Related Work

The sequential programming model is widely understood to be the most basic and ubiquitous way to develop an application on present-day machines. Other models exist, however, to exploit the physical and logical parallelism present on many modern processing platforms and thus potentially achieve higher performance.

Dataflow is a programming model that delineates the parallelism available in a program. Originally, dataflow was developed to exploit the massive parallelism available in some applications. It does this by departing from the globally updated memory model which had become a bottleneck for traditional von Neumann processors [16]. In dataflow, an application is expressed as a directed graph of processing tasks, each with all its inputs and outputs explicitly provided. In *fine-grain* dataflow, these tasks may be single instructions or logic operations. In *coarse-grain* (also called large-grain) dataflow, these tasks may be arbitrarily more complex, but usually describe a single high-level operation that is not efficiently or conveniently broken into smaller pieces [24].

Whereas early work concentrated on fine-grain dataflow to replace the classic von Neumann architecture, later work recognized that practically, better performance was reached on traditional machines through *hybrid* approaches, which use coarse-grain dataflow to direct internally von Neumann processes [25]. Modern dataflow architectures, especially in the digital signal processing (DSP) domain, have seen benefits from using such a hybrid approach [3]. Recently, Johnston, et al. have suggested that coarse-grain approaches "[offer] the most potential for improvements to dataflow programming" [16].

Dataflow languages tend to have many advantages, as well as limitations. By explicitly specifying the inputs and outputs of every task, the flow of data through the application may be optimized, which allows for an effective allocation of operations to resources. Certain applications, particularly DSP and audio/video media processing are easily expressed using dataflow, and several dataflow languages exist in these domains [12, 23, 33, 21].

Additionally, dataflow lends itself to visualization, which has a positive effect on the communicability of dataflow programs even to those not familiar with the language. This has led to the popularity of data flow visual programming languages (DFVPLs) such as LabView [14] with researchers in the natural sciences.

A recent development in dataflow has been the recognition of the *streaming* application domain. These are applications that operate on large sequential "streams" of consistently formed data such as audio, video, radio-frequency communications, network traffic, and

sensor network readings. The authors of the StreamIt language ascribe the following characteristics to streaming applications (paraphrased from [28]):

- The application operates on large or virtually infinite sequences of data.
- The algorithm performs a sequence of independent, self-contained transformation "filters" on the stream.
- The algorithm operates in a stable computation pattern, where the set of filters are "repeatedly applied in a regular, predictable order".
- The application will occasionally pass out-of-stream communications, such as error messages or changes to run-time configurations such as volume settings or filter coefficients.
- Users of the application expect high performance, in terms of resource efficiency, throughput, and latency.

Note that while this list is generally accurate, not all streaming applications rely on all these characteristics. Many limit or avoid the amount of out-of-stream communications passed through the system in order to simplify the portion of the application that needs to be fine-tuned. Also, it is quite common to leverage an increase in end-to-end latency in order to get an improvement in throughput through the use of pipelining techniques.

Given a sequential series of tasks and a pipeline of computation resources, it is possible to optimize the mapping of tasks to resources based on the performance (modeled or profiled) of each task. A variety of algorithms to perform this optimization in the context of network processors is presented in [7]. These algorithms are generalizable to the heterogenous systems supported in Auto-Pipe. Similar algorithms may be able to optimize for other metrics such as those mentioned in Section 1.1.

In order to optimize a mapping as mentioned above without a combinatorial explosion (every input data set, and every assignment of task to resource), the system must be modeled with certain assumptions. These analytically-based models can take many forms, the

simplest and most common of which assumes that every task takes a different but constant time to perform its operation. This model, which often uses the mean value of the task execution time (and is thus called the "mean value model"), is often sufficient when task execution times have low variance. This simple model may also be extended [18] to support tasks with arbitrary degrees of processing time variance and covariances with other tasks. Additional models are needed to take into account data-correlated processing times and other complexities.

Auto-Pipe incorporates a wide set of features from many areas of computer technology. Thus, Auto-Pipe and the X Language share features with many related projects, both commercial and academic. The X Language has common goals with a diverse range of programming languages in the linguistic domains of dataflow, hardware description, simulation, and application-specific codesign. However, X is the only language to date that incorporates the broad set of features discussed below. Further empowered by X-Sim and X-Opt, the Auto-Pipe system builds on previous work in pipeline optimization.

Table 1.1 provides an overview of selected related projects, including languages, compilation tools, and design environments. Check marks ($\surd$) indicate that the project (row) possesses that feature (column). The "Language" heading lists the languages (or language types) supplied by the user when creating applications. When multiple languages are used for different purposes within the project environment, the design or coordination language (or languange type) is listed first, followed by the implementation language(s). "Visual" refers to whether the dataflow graphs are visualized and/or edited through a graphical interface; "in" here means that dataflow graphs are input and edited visually, and "out" means that dataflow graphs may be visualized. "SW Gen" and "HW Gen" specify whether the parser or other tools can create components that are deployable to software or hardware using the application description. "Sim[ulation]," "Perf[ormance] Ana[lysis]," and "Optimization" indicate if the project supports reasonably useful algorithm simulation, application performance analysis, and analysis-based algorithm optimization, respectively.

Table 1.1: Overview of related projects

| Project name | Language(s) (design; impl) | Visual in/out | SW gen. | HW gen. | Het. codes. | Sim. | Perf. ana. | Optim-ization |
|---|---|---|---|---|---|---|---|---|
| GLU | functional; C | | √ | | | | | |
| Ptolemy | scripted; C, asm | √/√ | √ | √ | √ | √ | | |
| PtolemyII | XML; Java | √/√ | * | | √ | √ | √ | |
| LabView | visual only | √/√ | √* | √* | | | | |
| StreamC | C++-like; C++ | | √ | √ | | √ | | √ |
| StreamIt | C++-like; C | /√ | √ | √ | | √ | | √ |
| *X-Com only* | X; C, VHDL, ... | /√ | √ | √ | √ | | | |
| *Auto-Pipe* | as above | /√ | √ | √ | √ | √ | √ | √ |

* See text for details.

Granular Lucid or GLU [15] is a language that, similar to X, employs a hybrid dataflow-procedural structure. Rather than use a simple structural language, however, GLU uses Lucid as its coordinating language. Lucid [2] is a functional dataflow language whereby function evaluations form the "flow" of the data. GLU uses the concurrency and control mechanisms of Lucid, but with procedurally defined (written in C) basic functions and datatypes. As a functional language, GLU's coordination syntax is more sophisticated than X at expressing control flow. However, this is at some detriment to large or deep algorithms, or those with high flow complexity such as highly branched topologies.

Ptolemy II [8] is "a set of Java packages supporting heterogeneous, concurrent modeling and design." It supports the simulation of a number of computation model domains, including dataflow, discrete-event and continuous-time systems, finite state machines, and process networks, and many other domains and sub-domains. Ptolemy II uses a GUI called Vergil to create XML descriptions of interconnected tasks. These tasks, or "actors," are implemented using Java, and are invoked by the appropriately scheduled calls from customized computation model domain controllers. Some work on code generation from Ptolemy II models does exist [29], but it is not a primary goal of Ptolemy II, and the generators neither concentrate on high performance, nor do they yet support any targets other than a desktop Java environment. Ptolemy II is a very vast, complex, and diverse environment, encompassing many different development and research goals as well as the different computation models.

Ptolemy [27], or Ptolemy Classic, is a predecessor to (but not a previous version of) the Ptolemy II project. Ptolemy was created to aid development of embedded processing environments, particularly digital signal processing. Ptolemy is capable of generating code for a number of platforms; implemented generators exist for C, C++, VHDL (incomplete), and DSP assembly (on the Texas Instruments 320 series) targets. A variety of computation models are supported for simulations, however code generation targets may only use traditional data flow and some close derivatives. Ptolemy is now fairly dated, and execution of the tools are only supported on the Solaris and HP-UX operating systems. Ptolemy is not as complex as Ptolemy II, but it still encompasses an entire design methodology rather than just a combination of traditional tools.

LabView [14] is a popular proprietary programming environment among scientists and engineers. Users design dataflow graphs using the graphical language "G," where the tasks can be of varying degrees of complexity from simple adders to robust data visualization modules. Interconnects can contain the traditional set of scalar and array integer and floating point values as well dynamic types such as key→value maps. LabView programs are edited and executed in real-time, and offer performance not far from traditional procedural languages. The interface is intuitive for many applications such as DSP and sensor visualization, and is relatively quite easy to use — to the point that many proficient LabView users have little or no procedural programming experience. Recently, National Instruments (the creator of LabView) has developed a set of FPGA development boards and software support to target LabView designs to these devices.

StreamC [11] and StreamIt [28] are two projects that focus on the design of efficient streaming applications. Each language uses a different expansion of C++-style syntax to express interconnected tasks ("kernels" in StreamC of "filters" in StreamIt), which are expressed in line with the coordination language. The languages support common stream-related concepts such as synchronous dataflow consumption/production ratios, array composition and dimensions, and data access strides. Also of note is that each language was, in part, designed to aid development of a particular streaming multi-processor. StreamC can thus create parallel assembly code for the Stanford Imagine [1] processor, while StreamIt can

generate parallel C code for the MIT RAW processor. Additionally, a StreamC-to-FPGA compiler exists [11] that targets a particular FPGA development board, and makes use of some of the basic optimization techniques available to the main StreamC compiler.

The stream optimization procedures that X-Opt will support are very similar to those employed by StreamC and StreamIt. X-Opt will have to support more complex, heterogeneous systems and irregular (not just synchronous data flow) tasks, thus it will require somewhat more complicated modeling and analysis techniques. X-Opt will also support optimization over metrics beyond mean-value throughput.

X is unique as a *coordination* language that strives to support the high-performance, concurrent execution of an assortment of implementation languages on arbitrary combinations of resources. X, and thus the Auto-Pipe system, is as powerful and as capable as the platforms that it can support through the X-Com code generators. It is important that X-Com and the Auto-Pipe system as a whole grow to support more platforms as new languages and resources are developed.

Streaming programming languages can be expected to gain popularity as they become more robust and support a greater number of platforms, especially non-traditional computation resources such as graphics processors [4]. Current streaming languages use the data flow computation model, so a one-to-one relationship can be established between, for example, the StreamC task interface and X's task interface. Therefore, they have the potential of becoming a favorable X task implementation language for individual blocks or even compound blocks.

## 1.5 Overview of Thesis

This thesis presents the X Language and the X Language compiler, a significant subset of the Auto-Pipe design environment. Chapter 2 contains a discussion of the goals and design of Language X, and a formal specification of the X Language. Chapter 3 presents an

overview of the software components used by the X Compiler. In Chapter 4, sample X applications are provided and analyzed. Chapter 5 summarizes the contributions associated with this thesis, and future work that needs to be completed to fully reach the goals of the Auto-Pipe system.

# Chapter 2

# The X Language

The X Language is a dataflow coordination language specifically designed to incorporate traditionally programmed tasks developed for a wide variety of processing platforms, both software and hardware.

This chapter introduces the X Language in four parts. Section 2.1 discusses the goals which have guided the design of the X Language. Section 2.2 walks through the language in an increasingly complex manner to introduce the language in a practical manner. The last two sections are provided for reference and present a more detailed explanation of the language. Section 2.3 is a glossary of terminology that is used in the X Language. Finally, Section 2.4 contains a formal specification of the language.

## 2.1 Goals

The X Language was developed to meet the needs of domain-specific developers that have experience using traditional procedural languages. These users could be, for instance, researchers in the natural sciences who are competent in basic scientific C programming, but do not have detailed experience in developing and debugging inter-procedure communications and optimizing queuing systems.

There are three primary goals that have driven development of the X Language:

- The first main goal is that of aiding the user in *expressing* a wide variety of algorithms, resource types, and resource architectures. This goal also includes the converse of not trying not to impede the user's ability to express task and resource structures in the domains we support.

- The second main goal involves making X *easy to use*. Given the high degree of expressiveness of X, this goal requires that it be relatively easy to exercise the capabilities of the language. This applies not just to the ease with which the user writes X Language files, but also how easy it is to write modules that interact with the language (such as new resources and task implementations), and how clear and understandable the procedures are from a user's point of view.

- The third main driving goal is the support of using X to develop *high-performance* applications. It is expected that many users of X and the Auto-Pipe system will be seeking to improve the realized performance compared to traditional programming languages and processing devices.

The goal of linguistic expressiveness takes many forms in the X Language. An important and unique feature of the language is its ability to express complex architectures of *diverse* computation and interconnection resources. Particular attention was given to support development and deployment to both heterogeneous (e.g., FPGA and GPP) and widely parallel (e.g., clusters) processing architectures, especially algorithms organized in a pipelined manner. Additionally, X expresses *complex* topologies of resources, supporting the wide and deep hierarchies of resource interconnections found in real-life architectures.

An *intuitive* syntax which takes advantage of the well-organized nature of dataflow programs is an important aspect of the expressiveness of the language. Blocks, resources, and other entities are created to be *parameterizable* for convenient reconfiguration of the application. Data types are chosen to be easily recognized and understood by inexperienced programmers, yet support reasonably complex data structures. Also, the X Language is capable of expressing common simple topological operations such as the splitting/merging of composite data types into/from multiple edges; it was decided that more complicated

operations like multiplexing and reordering would be best handled by permitting the user to define their own blocks.

The X Language enables the goal of easing development by heavily supporting the exploration of many design decisions. Decisions such as algorithm organization, component parameterization, communication topology, and block and link mapping are all easily changed using the syntax provided by X.

An important aspect of supporting development with reduced effort is the reduction of tedium in writing X Language descriptions of systems. A balance is found between reducing the quantity of code needed to express language structures (resulting in fatigue and "copy-and-paste" errors), and retaining a certain degree of redundancy to prevent mistakes. Code reuse — whether extending or just reusing old X modules — is supported to further reduce development time and tedium.

X has been developed with modestly experienced programmers in mind. These include developers who are capable of traditional development of domain-specific (e.g., science, engineering, networking) applications, but would like to improve their performance by employing less-traditional technologies (e.g., FPGAs), diverse architecture organizations, and detailed simulation, analysis, and optimization tools.

As discussed in Section 1.4, *hybrid dataflow* is becoming a particularly favorable approach to creating high-performance applications. Supporting this approach furthers the goal of high performance. This involves ensuring that the language does not hinder algorithms which are appropriately written in a large-grain dataflow manner [1].

While the performance of the task implementations is certainly the most fundamental aspect of performance, there are a tremendous number of languages which can be used to achieve performance at that level. The unique approach of X is that it does not try to

[1] Some applications benefit from a completely Von Neumann architecture or fine-grain dataflow and are indeed hindered by large-grain dataflow, however these are outside the domain of this language and are generally not found in streaming applications. They can still be expressed, however their performance will not be improved by adjustments at the mapping and communication layer.

outperform all or indeed any of these implementation languages, and instead seeks to improve the performance of the coordination of tasks.

Much of the actual performance gain from using X is through the capabilities of X-Opt and the rest of the Auto-Pipe tools. Since X and Auto-Pipe were developed specifically to support each other, the performance gains of these two go hand-in-hand.

## 2.2 Tutorial Walkthrough

Chapter 1 introduced the concepts of tasks, processing architectures, and mapping. Data-flow graphs are made of a set of interconnected tasks, which combine together to form an algorithm. A processing architecture is the set of computation resources and interconnect resources (connecting the CRs) that will execute the tasks and transmit data between the tasks. The tasks are assigned to resources in the architecture through mapping.

This section will incrementally expose the syntax of the X Language through the use of examples. This walkthrough contains four sections which will introduce the specification of algorithms as interconnected blocks, the specification of resources comprising a processing architecture, and the mapping of an algorithm onto an architecture.

Figure 2.1 is a high-level view of the steps involved in the X Language description portion of application development. As indicated by the figure, the algorithm and processing architecture are developed independently, and then brought together through mapping. Once completely mapped, the X Language description of the application can be compiled, and the actual block implementations are brought in (this portion is discussed further in Chapter 3).

Sections 2.2.1 and 2.2.2 present two different algorithm descriptions. Section 2.2.3 presents a set of resource class hierarchies and instantiates the resources to create a processing architecture. Section 2.2.4 introduces the mapping operation and develops two complete mappings of the Section 2.2.2 algorithm onto the Section 2.2.3 architecture.

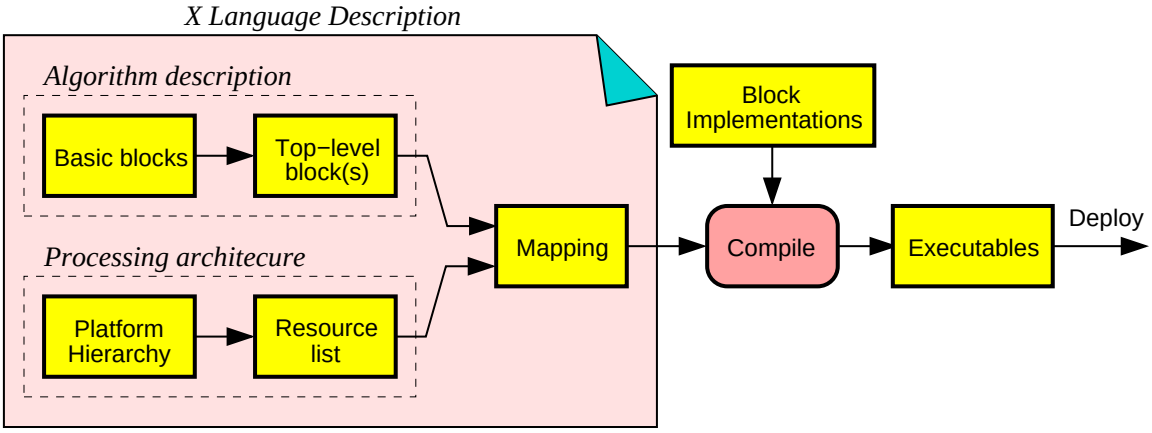



Figure 2.1: Overview of X Language description

### 2.2.1 A simple algorithm

The key concepts; *block, data type, compound block, block instantiation,* and *edge* are discussed first. This section will create a small algorithm, starting with a single block and expanding to combinations of blocks. The blocks are combined through the instantiation of multiple blocks within a compound block, and connecting them together with edges.


```
1 block RNG {
2         output FLOAT32 y;
3 };
```

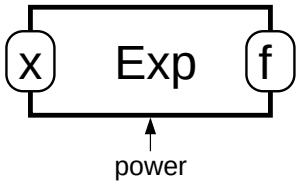
Figure 2.2: A basic block

A single *block*, `RNG` (i.e., Random Number Generator), is presented in Figure 2.2, alongside the X Language code used to create it. Blocks are the basic computation elements expressed in the X Language. Since the actual implementations lie outside the language description, the only distinguishing characteristic of a block is its identifier (`RNG`) and its port specification (output port `y`, of type `FLOAT32`).

The block identifier is `RNG` (line 1), and will be understood here as a random number generator, although it is up to the actual implementation(s) of the block to agree to the appropriate semantics of how the block functions. The actual implementation may be, for example, a set of C functions implementing a random number generator. The language support for

associating implementations with blocks is discussed further in Section 2.4 and in Chapter 3.

`RNG` has one output port, `y` (line 2) and associated data type, `FLOAT32`. `FLOAT32` is a 32-bit IEEE floating point type, and it is one of the basic data types supported by the X Language. For a complete list of basic data types, refer to Section 2.4.4. The port identifier, `y`, is used to connect to other blocks, as will be seen later in this section.




```
5 block Exp {
6         input FLOAT32 x;
7         output FLOAT32 f;
8         config FLOAT32 power = 1.0;
9 };
```

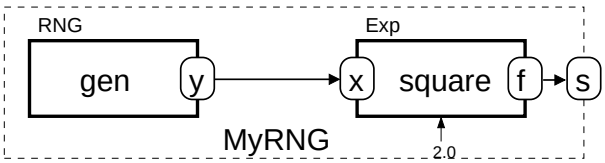
Figure 2.3: Another block

Another block, `Exp`, is defined in a similar manner as `RNG` (Figure 2.3). The keyword `input` on line 6 is used to create an input port, `x`, much like the output port in the previous example. Again, the datatype is a `FLOAT32` value.

On line 8 is another new keyword, `config`. The `config` statement syntax is much like a port declaration. It describes a block configuration parameter that is used to parameterize blocks at compile-time. In this case, `power` is the name of the configuration option, and it has been provided a *default value* of 1.0. The default value is a value that is passed to the block when no overriding value is specified during instantiation of that block. The default value is optional; for instance, the "`= 1.0`" portion could be omitted. In that case, the user would be *required* to provide a value for `power` when instantiating `Exp`.

Shown with a dashed border in Figure 2.4 is `MyRNG`, a *compound block*. A compound block is one that contain "subblocks" and edges within itself. Note that compound blocks can still have inputs and outputs, and indeed `MyRNG` contains a `FLOAT32` output named `s`.

`MyRNG` contains two blocks, `gen` and `square`, which have been instantiated on lines 14 and 15. A *block instantiation* is used inside a compound block to create subblocks of a given block type and with a unique (to the compound block) identifier (e.g., `gen` and `square`).



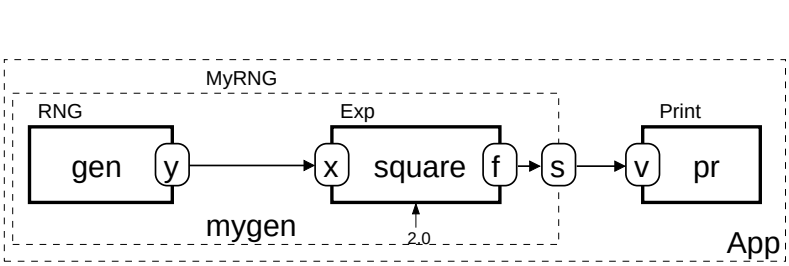

```
11 block MyRNG {
12         output FLOAT32 s;
13
14         RNG gen;
15         Exp square(power=2.0);
16
17         gen.y -> square.x;
18         square.f -> s;
19 };
```

Figure 2.4: A compound block

Configuration options may be passed to instantiated blocks, using the syntax seen on line 15. If a default configuration option is not available within the subblock, then configuration *must* be provided.

Edges connect the blocks together, from block output to block input. Line 17 creates an edge from the `y` output of the `gen` block to the `x` input of the `square` block. Line 18 creates an edge from the `f` output of `square` and connects it to `MyRNG`'s output port `s`.

`MyRNG` is itself a block which can be instantiated like any other, and will behave in accordance with the combination of its subblocks.




```
21 block App {
22     MyRNG mygen;
23     Print pr;
24
25     mygen -> pr;
26 };
27
28 use App;
```

Figure 2.5: A complete algorithm

Figure 2.5 demonstrates a compound block, `App`, which contains the previous `MyRNG` block and a new block, `Print`. Because `App` has neither inputs nor outputs, it may be used as an *algorithm*, which here means a fully self-contained arrangement of blocks that implements an application. The `use` statement on line 28 tells the compiler to indeed instantiate `App` in this manner (i.e., as a complete algorithm).

Note that the language intentionally requires that all data sources and data sinks are contained and described within the top-level algorithm. Source- and sink-less algorithms can easily be developed within a compound block, and then attached to various sources and sinks in a top-level "wrapper" instantiating the filter block.

Notice that on line 25, no ports were specified in edge declaration. This is because X supports the idea of a default input and/or output port, which may be inferred *only* when there is only one port that will function in the given context.The exact same behavior could have been reached with the explicit statement:

```
25        mygen.s −> pr.v;
```

Figure 2.6 provides the full code listing of the above example.

### 2.2.2 Another algorithm

The key concepts presented in this section include *array data type, block array, edge label, split edge,* and *merge edge*. This section expands upon the block structures in the previous section by introducing arrays of blocks, edges with array data types, and special edges that merge/split arrays from/into their individual elements.

Figure 2.7 depicts a more complex algorithm with new language constructs that will be introduced in this section. The full code listing of this algorithm is available in Figure 2.8.

The `Cross` block description on lines 5–8 of Figure 2.8 introduces the *array data type*. Whereas scalar data are created from a set of X "basic data types," the `ARRAY` keyword can create homogeneous arrays of a specified data type and length. The statement on line 21,

```
21        output ARRAY<SIGNED8>[2] p;
```

creates an output port `p` that is an array of two `SIGNED8` values. On line 30, `SendData` uses a similar statement to specify its input port, an array of two `UNSIGNED32` values.

This example also introduces the *block array*, used in line 6 of the description of `Algo`:

```
6         RunSum sum[2];
```

```
1  block RNG { // generate a normal random number
2          output FLOAT32 y;
3  };
4
5  block Exp { // raise <x> to <power>
6          input FLOAT32 x;
7          output FLOAT32 f;
8          config FLOAT32 power = 2.71828;
9  };
10
11 block MyRNG { // generate a χ^2 random number
12         output FLOAT32 s;
13
14         RNG gen;
15         Exp square(power=2.0);
16
17         gen.y −> square.x;
18         square.f −> s;
19 };
20
21 block App {
22         MyRNG mygen;
23         Print pr;
24
25         mygen −> pr;
26 };
27
28 use App;
```

Figure 2.6: The complete Section 2.2.1 example

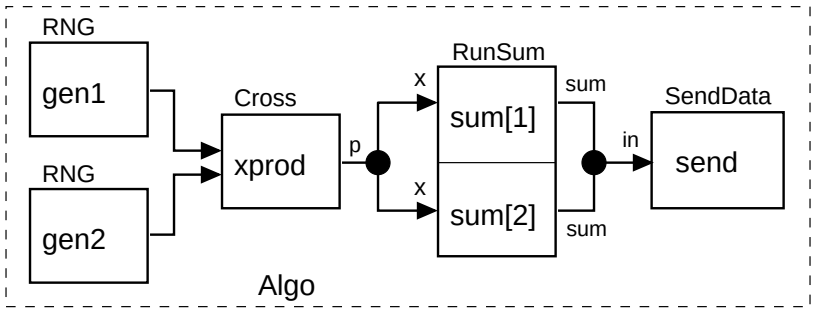



Figure 2.7: A more complex algorithm

```
1  use Algo app;
2
3  block Algo {
4          RNG gen1, gen2; // create two random number generators
5          Cross xprod;
6          RunSum sum[2]; // create an array of two running summers
7          SendData send; // send is a data sink
8
9  ea:    gen1 -> xprod.a;
10 eb:    gen2 -> xprod.b;
11 ec:    xprod =< { sum[1].x, sum[2].x }; // split xprod's output into two SIGNED8s
12 ec:    sum >= send;  // merge the two UNSIGNED32s into send's input
13 };
14
15 block RNG {
16         output FLOAT32 y;
17 };
18
19 block Cross {
20         input FLOAT32 a, b;
21         output ARRAY<SIGNED8>[2] p;
22 };
23
24 block RunSum {
25         input SIGNED8 x;
26         output UNSIGNED32 sum;
27 };
28
29 block SendData {
30         input ARRAY<UNSIGNED32>[2] in;
31 };
```

Figure 2.8: The complete Section 2.2.2 example

Lines 9–12 describe the edges in `Algo`. Note the addition here of *edge labels*, at the beginning of each edge construction. The statements `ea:`, `eb:`, and `ec:` attach a label, which does not need to be unique, to each edge. Note that the last two edges are given the same label, `ec`. These will be used later (Section 2.2.4) to refer to the edges for the mapping operation.

Two new types of edges are introduced in the `Algo` block description. The first is the *split edge*, on line 11.

```
11          xprod =< { sum[1].x, sum[2].x };
```

This statement creates an edge from the `p` (default) output of `xprod` that splits the data type `ARRAY<SIGNED8>[2]` into its two component `SIGNED8` types, providing them to the elements of the ordered array. That is, the first element of the array goes to the `x` input of `sum[1]`, and the second element goes to the `x` input of `sum[2]`.

The second new type of edge is the *merge edge*, seen on line 12.

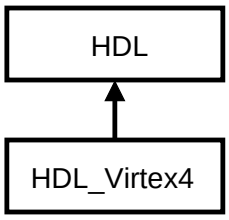
```
12          sum >= send;
```

The merge edge follows similar syntax to the split edge but in reverse, with a list of scalar outputs provided to an array input. In this case, we use a convenient method to express a merge edge when the source blocks are all members of the same array. This edge connects the default `sum` output of `sum[1]` and `sum[2]` to the default `in` input of `send`, which is of type `ARRAY<UNSIGNED32>[2]`. Note that this could instead have been written explicitly to achieve the same behavior:

```
12          { sum[1].sum, sum[2].sum } >= send.in;
```

### 2.2.3 Platforms, Linktypes, and Resources

This section explores the following key concepts: *platform*, *subclass*, *linktype*, *computational resource*, and *interconnect resource*. The concepts presented here are used to first express the class-space of computational and interconnect resources, and then to create processing architectures from such resources.




```
1  platform HDL;
2
3  platform HDL_Virtex4 : HDL {
4          config STRING part;
5  };
```

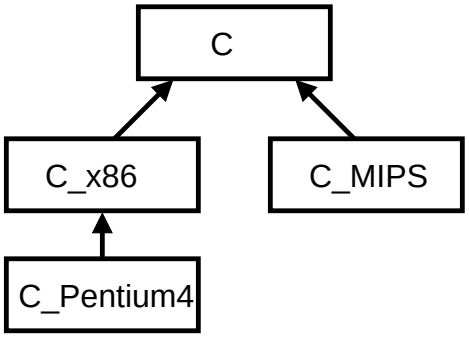
Figure 2.9: Example `HDL` platform hierarchy

Figure 2.9 shows the declaration of the `HDL` *platform* hierarchy. In line 1, the `platform` statement informs the compiler that `HDL` is an available platform. Platforms are classes of computational resources; that is, all computational resources are an instance of a particular platform.

On line 3, the `HDL_Virtex4` platform is declared as a *subclass* of the `HDL` platform. If a platform is a subclass of another platform, it is capable of implementing anything that its parent class is able to implement. For instance, if `HDL` can implement the block `Foo`, then `HDL_Virtex4` can also implement `Foo`.

Platforms can contain configuration options using the same syntax used for block configuration options. An example of this is on line 4. Here, `part` identifies a configuration parameter of type `STRING`, that must be provided in order to create a resource instance of the platform. Platform configuration options are used to affect compile-time code generation. `part`, for instance, specifies a hardware part description that is used by the HDL code generator to customize the generated hardware for a particular FPGA device. Default configurations are also permitted, just like for block configurations.

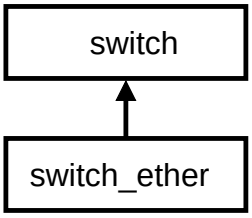
Figure 2.10 presents a hierarchy of platforms based on the `C` platform. `C_x86` and `C_MIPS` are subclasses of the `C` platform, and `C_Pentium4` is a subclass of the `C_x86` platform. Thus, an implementation of a block which has been designed specifically to execute on an instance of `C_x86` will also execute on an instance of `C_Pentium4`, but not on an instance of `C` or `C_MIPS`. The X Language compiler detects permissable mappings and informs the user if an invalid mapping is attempted.




```
10 platform C;
11
12 platform C_x86 : C {
13         config STRING hasMMX = "no";
14         config UNSIGNED16 freq; // in MHz
15 };
16
17 platform C_MIPS : C;
18
19 platform C_Pentium4 : C_x86 {
20         config STRING hasMMX = "yes";
21 };
```

Figure 2.10: Example `C` platform hierarchy

Notice also that on line 13, `C_x86` contains the configuration option `hasMMX`, which defaults to the string value ‘‘`no`’’. In the `C_Pentium4` platform, however, this value is overridden by the new default value ‘‘`yes`’’. Any resources of the platform `C_Pentium4` will thus be parametrized with `hasMMX=‘‘yes’’`. However, if line 20 were not present, then the default `C_x86` value specified on line 13 would be operative. In general, a single code generator will apply to a large tree of platforms (e.g. `C_x86` and its subclasses) thus providing a convenient way to configure the large number of code-affecting parameters occuring in both software and hardware devices.




```
25 linktype bus;
26
27 linktype bus_pci : bus {
28         config UNSIGNED16 freq = 66, width=32;
29 };
```

Figure 2.11: Example `bus` linktype hierarchy

Whereas computation resources are instances of platforms, interconnect resources are instances of *linktypes*. In Figure 2.11 we see an example of a linktype hierarchy. Linktypes are declared using the `linktype` statement, with the remaining syntax being identical to the `platform` statement for platforms. As computation resources are quite distinct from

interconnect resources in terms of their implementation, the linktype/platform distinction has been created to reduce confusion.

To create a *computation resource*, we must first have created the appropriate platform. Then, the `resource` statement is used to instantiate that platform. For example:

```
resource proc is C_Pentium4 ( freq=3400 );
```

instantiates the `C_Pentium4` platform type as a specific computation resource with identifier `proc`, and configures its `freq` parameter to the value `3400`. Note that `freq` here is a parameter inherited from the `C_x86` platform, which must be specified in order to instantiate the resource (since there is no default value).

Similarly, we can describe two computation resources with different identifiers, `bigfpga` and `lilfpga`, and different configurations, but that are of the same platform type:

```
resource bigfpga is HDL_Virtex4 ( part="XC4V8000" );
resource lilfpga is HDL_Virtex4 ( part="XC4V100" );
```

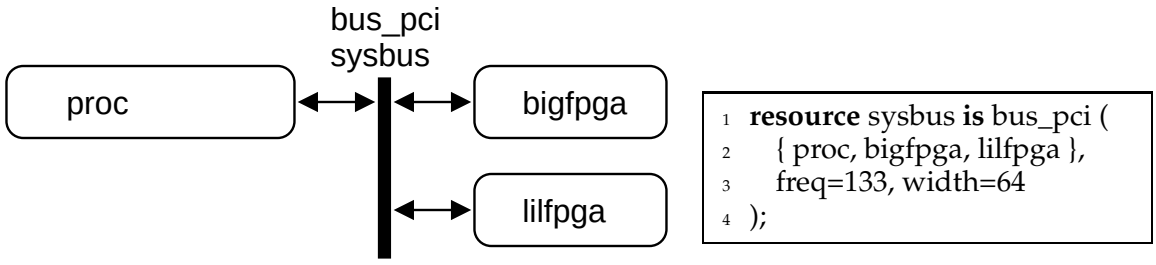
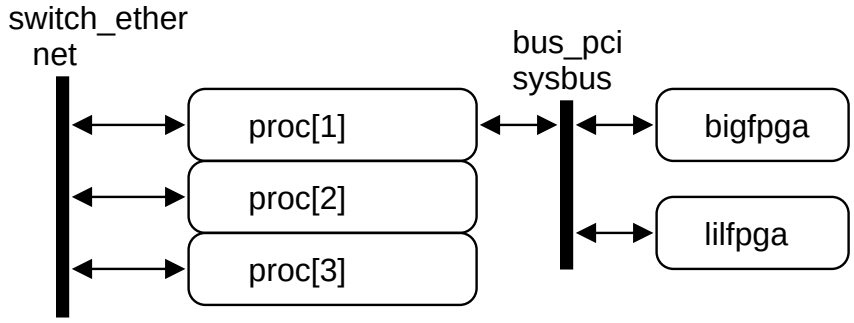



Figure 2.12: IR instantiation and small processing architecture example

Given the previous computation resource instantiations, we can now instantiate the *interconnect resource* found in Figure 2.12. In this example, the `resource` statement is used to instantiate `sysbus`, which is of linktype `bus_pci`. Interconnect resources are parameterized by configuration options, just like computation resources, the first parameter is always a list of resources connected to the interconnect. In the example, `proc`, `bigfpga`, and `lilfpga` are each connected to the `sysbus` interconnect resource.

Interconnect resources may also connect to other interconnect resources, although whether this is valid, modelable, or implementable depends on the particular interconnect.




```
1  resource proc[3] is C_Pentium4 {
2    (freq=3400), (freq=2800), (freq=2800)
3  };
4
5  resource net is switch_ether (
6     { proc[1], proc[2], proc[3] },
7     rate = 1000;
8  );
9
10 resource sysbus is bus_pci (
11    { proc[1], bigfpga, lilfpga },
12    freq=133, width=64
13 );
```

Figure 2.13: Larger processing architecture example

A final example of a processing architecture is found in Figure 2.13. This processing architecture is composed of five computation resources and two interconnect resources.

Lines 1–3 describe a *resource array* of three `C_Pentium4` type resources. Resource arrays are provided for convenience, as it is expected that users will often having homogeneous or nearly-homogeneous sets of resources available to them (clusters of processors, on-board FPGA arrays, etc.) Note the change of syntax in configuring a resource array; configuration is given as a brace-enclosed list of the usual parenthesis-enclosed configuration format.

This processing architecture connects each member of the `proc` resource array to the `net` interface. Resources are connected to `sysbus` in the same manner as the previous example.

### 2.2.4 Mapping

This section presents the ideas of *mapping* and *complete mapping*. A complete mapping, made up of individual block-to-CR and edge-to-IR mappings, describes the assignment of an algorithm to a processing architecture.

The following examples of mapping assume the processing architecture of Figure 2.13, and the block descriptions of Section 2.2.2 (summarized in Figures 2.7 and 2.8).

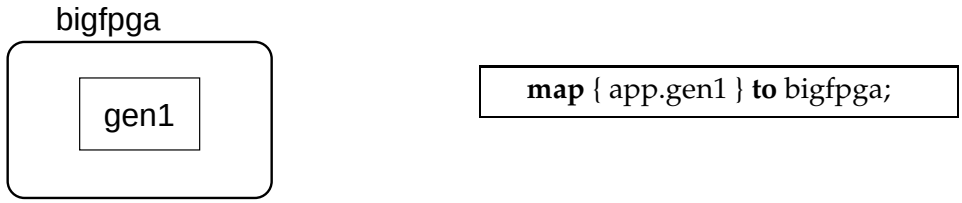



Figure 2.14: Mapping to a computation resource

Figure 2.14 performs a single *mapping* of a block onto a resource. The `map` statement maps a list of fully-specified blocks to a computation resource, or fully-specified edges to onto an interconnect resource. In this example, `app.gen1` specifies the `gen1` block contained in the `Algo` block named `app` by the `use` statement (on line 31 of Figure 2.8).

A single mapping does not necessarily constitute an implementable assignment of blocks to resources; each block in every used algorithm must be mapped, as will be seen later in this section.

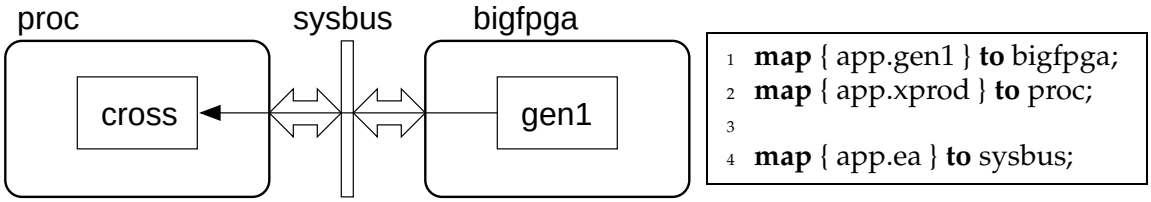
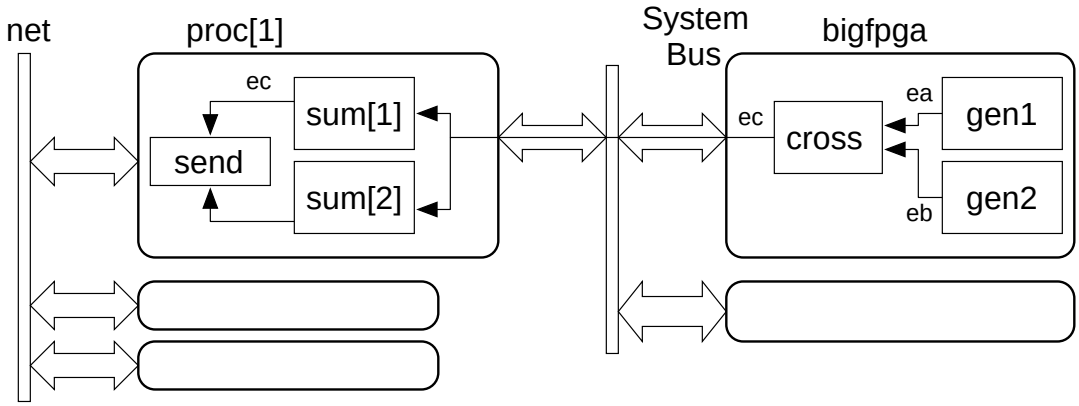



Figure 2.15: Mapping to an interconnect resource

Figure 2.15 demonstrates the mapping of `app`'s `gen1` to `bigfpga` and `xprod` to `proc` using the syntax of the previous example. It also shows a mapping of edge `ea` onto an interconnect




```
1 map { app.gen1, app.gen2, app.xprod } to bigfpga;
2 map { app.sum, app.send } to proc[1];
3
4 map { app.ec } to sysbus;
```

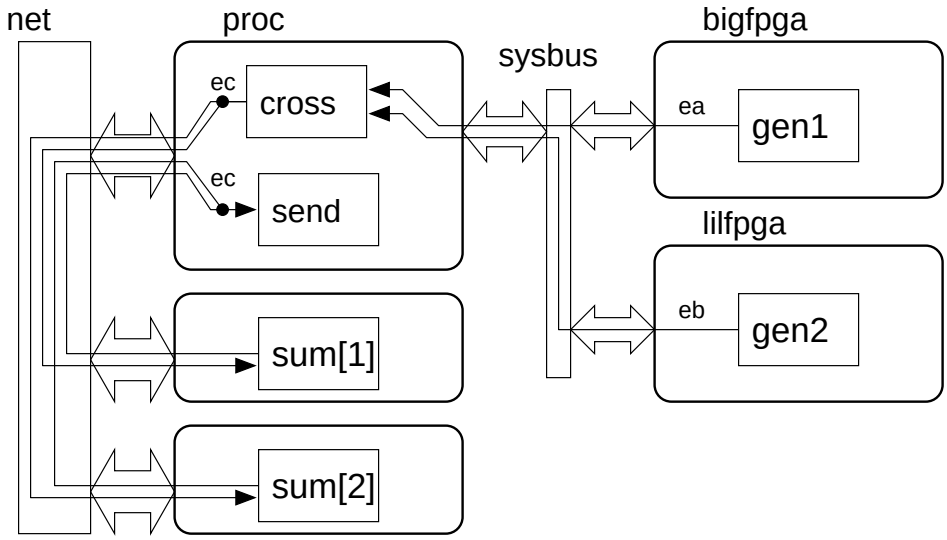
Figure 2.16: A complete mapping

resource. The edge from `gen1` to `xprod` is mapped onto `sysbus`, using the label assigned to the edge in the edge's construction (line 25 of Figure 2.8).

Figure 2.16 provides a *complete mapping* (different from the *partial mapping* of Figure 2.15) of the algorithm onto the processing architecture. Here, all blocks have been assigned to computation resources, and the edges connecting blocks on different resources have each been mapped to the appropriate interconnect resource.

In line 2 of the example, the use of `app.sum` causes all members of the `sum` array to be mapped. The compiler expands any occurance of a block array identifier without subscript to mean all the members of that block array.

Also notice that not all edges must be mapped. The edges in label `app.ea` are contained within `bigfpga`, thus there is no requirement to map them to an interconnect resource. Furthermore, note that while `app.ec` describes both the edges from `xprod` to the `sum` array and the edges from the `sum` array to `send`, the `map` statement does not cause the latter merge edge to actually map to `sysbus`. This is because the merge edge is contained within `proc[1]`, and thus are ignored for mapping by the compiler.




```
1 map { app.gen1 } to bigfpga;
2 map { app.gen2 } to lilfpga;
3 map { app.xprod, app.send } to proc[1];
4 map { app.sum[1] } to proc[2];
5 map { app.sum[2] } to proc[3];
6
7 map { app.ea, app.eb } to sysbus;
8 map { app.ec } to net;
```

Figure 2.17: Another complete mapping

An alternative complete mapping example is provided in Figure 2.17. Here, blocks and edges are distributed more evenly (in terms of quantity) among the resources. `app.gen1` and `app.gen2` are mapped to the two `HDL_Virtex4` resources, and their outputs are connected to `app.xprod` on `proc` by mapping their respective edges `app.ea` and `app.eb` to `sysbus`. The single mapping statement on line 8 connects `app.xprod` to the `app.sum` blocks (now on different resources) and back to `app.send` over the `net` resource.

This concludes the tutorial. All constructs of the X Language required to produce an implementable application have been presented above. Other factors remain, however, before a complete application can be synthesized and deployed. All blocks occurring in the algorithm must have a corresponding implementation in a supported language, and each implementation must be created for the platform to which the block is mapped. Additionally, a deployment must be described that associates each resource with a real-world device. These mechanisms are discussed further in chapter 3.

## 2.3 Terminology

To differentiate the many different layers of abstraction that are necessary when using the X Language, a specific terminology is used when discussing applications. The terms found in alphabetic order in this section are specific to the X Language. They are largely based on those used in related literature and academic projects.

- *Algorithm*: An algorithm here refers to the particular arrangement (a directed graph) of blocks and edges which make up the X Language description of an application. The algorithm is the highest-level compound block which is instantiated by the compiler.
- *Atomic block*: An atomic block is a block that does not contain any internal blocks or edges. Atomic blocks are generally the blocks to which physical implementations are eventually associated.
- *Block*: A block is an X Language description of a processing task. The description contains any number (including zero) of each: inputs, outputs, and configuration. Blocks may be atomic blocks, which may be associated with implementations and eventually placed on devices, or compound blocks, which contain blocks and edges and are used to organize tasks consisting of smaller subtasks.
- *Compound block*: A compound block is a block that itself contains instances of blocks and edges, usually connected to its own inputs and outputs.
- *Deployment*: Deployment is the specification, or the specification process that involves associating a computation or interconnect *resource* (IR) with a specific computation or interconnect *device*. Deployment is an injective operation; that is, each resource is deployed to a unique device, but not all devices need contain a resource. Deployment takes place in the X-Dep tool, *after* the X Language description compilation; it is discussed further in Chapter 3.

- *Device*: A device is a tangible object of a specified platform or linktype. That is, a device corresponds to a particular real-world or simulation-world entity. Interconnect resources (IRs) and computation resources (CRs) are *deployed* to interconnect devices and computation devices outside the X Language description (see Chapter 3).

- *Edge*: An edge is a virtual interconnect from one block's output port to another block's input port. The X compiler collapses composite edges, maps them to IRs, and generates their implementations.

- *Implementation*: An implementation is a platform-specific function (e.g., a C function or HDL entity) which is used to implement a block's or link's functional task. A mapping indicates which implementation a block or edge will use.

- *Instance*: Instantiation is the operation of creating an entity of a particular type. In X, a block instance is a named object of a particular block type within a compound block (or at the top level). Compound blocks may contain multiple instances of the same block, as long as they do not share the same name (including subscript).

  Instantiation is also used in the processor architecture. A computation resource (CR) is an instance of a platform, and an interconnect resource (IR) is an instance of a linktype.

- *Linktype*: Similar to a platform, a linktype is a class of interconnect resource (IR) recognized by the X Compiler, over which data may be passed between implementations of X blocks. For example, an ethernet switch, PCI system bus, or on-chip shared memory interface could be used as a linktype.

- *Mapping*: Mapping refers to either the specification, or the act, of assigning a block to a computational resource, or an *edge* to an interconnect resource.

- *Platform*: A platform is a class of computation resource (CR) recognized by the X Compiler for which the compiler may generate an executable. Due to the construction of the compiled environment, a language is often a necessary attribute of the platform. Examples include "C-x86," "C-Alpha," and "HDL-Virtex4." Platforms are organized in a hierarchy; for instance, an implementation written for the "C-x86"

platform may run on the "C-Pentium4" platform, but not the "C-Alpha" platform. Computation resources are specific, named objects of a specified platform.

- *Resource*: Resources refer to both computation resources (CRs) and interconnect resources (IRs). Computation resources, such as "a C-Pentium4 resource" are instances of specific platforms. Interconnect resources, such as "an ethernet switch" are instances of specific linktypes.

## 2.4 Language Specification

This section describes the syntax and semantics involved in creating a system specification compliant with the X Language. Most of the specification is presented here, however some less important details (comment styles, etc.) have been left out. [30] contains the verbose specification and is updated as the language evolves.

An Extended Backus-Naur Formalism (EBNF) notation [2] is used throughout, with some additional style changes made for clarity. To summarize:

- A symbol on the left-hand side of the ::= is defined by its substitution on the right.
- Symbols in upright **boldface** are non-terminal symbols (defined further in this section), and begin with an uppercase letter.
- Symbols in ***slanted boldface*** are terminal symbols (without a further definition provided).
- The remaining `plainface` words, and those found in single quotes ' ' are X Language literals (keywords, delimiters, etc.).
- The pipe | character delineates symbol substitution choices.
- Parentheses ( ) group a set of symbols into one logical symbol.

[2] There are a variety of different notations claiming the title "EBNF" (some of which are supported by different standards organizations). The notation in this section is based on [31].

- Square brackets `[ ]` group a set of *optional* symbols into one logical symbol.
- An asterisk `*` follows a symbol that may be replicated zero or more times.
- A plus sign `+` follows a symbol that may be replicated one or more times.
- White space has been ignored to make the notation more readible.

To aid interpretation of the symbols, literal *constant* symbols end with `C`, *top-level* statement symbols end with `S`, `block`-level statement symbols end with `BS`, and `platform`-level statement symbols end with `PS`.

### 2.4.1 Lexical conventions

```
char       ::=  nextline | digit | letter | somepunctuation
comment    ::=  // (char)* nextline | /* (char)* */
natural    ::=  (nonZeroDigit) (digit)*
integerC   ::=  [-|+] (natural | (0)*)
floatC     ::=  (digit)* [.]  (digit)* [e (digit)+]
stringC    ::=  " (char)* "
identifier ::=  (letter | @) (letter | digit | _ | @ )*
```

"C++ style" comments are allowed.

Integer and floating-point constants are permitted in certain circumstances. All values are entered in decimal (base 10) format unless otherwise specified, with optional exponentiation using the `e` character (e.g., `11.2e2` is equivalent to 1120). Entering floating-point constants when integers are expected produces undefined behavior.

String constants consist of a series of one-byte ASCII characters, surrounded by the double-quote character `"`. Only printing characters (ASCII values above decimal 31 and below decimal 127) excluding the double-quote character are supported, unless otherwise specified.

Identifiers consist of a sequence of letters, numbers, underscores '`_`', and at '`@`' characters An identifier may not begin with a number or underscore character. Identifiers, keywords,

and numerical constants are case-insensitive. String constants may be case-sensitive in some cases; this behavior is specified where relevant.

The following keywords may not be used as identifiers:

```
array      bind         block         constant
config     device       impl          input
linktype   map          output        platform
resource   struct       target        typedef
use        @blockrank   @devicerank   @starttime
@unique
```

### 2.4.2 Pre-processing directives

X-Com, the X Language file compiler, uses the C Preprocessor (cpp) before parsing the file directly, thus any directives supported by the native cpp are available to the user. Of particular note are the cpp statements `#include`, `#if[def]/#else/#endif`, and `#define`. Use of these directives is recommended to improve readability and reduce redundancy in the X code. cpp identifiers may also be passed in on the command line to facilitate automated construction of systems using the X Language.

### 2.4.3 Statements

**TopLevel** ::= (**BlockS** | **ConstantS** | **LinktypeS** | **PlatformS** | **ResourceS** | **MapS** | **TypedefS** | **UseS**)* *EOF*

X Language files are created from the top-level declarations and statements enumerated in the above grammar. Synthesizable X files contain at least one block and at least one `use` statement indicating an architecture to implement. The following sections examine the top-level statements in further details.

### 2.4.4 Data type syntax

X data types aid in the parsing and type-checking of block parameters and the type-matching of edges connected to block ports and other edges. Statements discussed below include `typedef`, used to define new types for convenient referencing, and `constant`, used to create named constants of a given value and type.

**BasicType** ::= `(signed | unsigned) (8 | 16 | 32 | 64) |`
`float (32 | 64 | 128) | string`

Native X Language data types are constructed from a set of basic data types, including floating-point numbers and signed and unsigned integers, each of various bit widths.

**DataType** ::= `array <` **DataType** `> [ (` ***natural*** `| '*') ] |`
`struct <` **DataType** `(,` **DataType**`)* > |`
**identifier** `|` **BasicType**

Homogeneous arrays of data types are constructed by using the `array` keyword. `array` requires a non-negative length to be specified when creating the data type, or `*` indicating variable length. Variable length arrays are of indeterminate length and their handing by the system is platform-specific.

To create heterogeneous data structures (i.e., consisting of multiple data types), the `struct` keyword is used. This data type is provided with an *ordered* and *unnamed* list of types that are contained in the structure. As an example, a data type containing two 16-bit unsigned values, an array of eight 8-bit signed values, and another 16-bit unsigned, in that order, would be declared with:

```
struct<unsigned16,unsigned16,array<signed8>[8],unsigned16>
```

Note that, as in the above example, `array` and `struct` data types may be nested to create arbitrary data types of larger dimension.

**TypedefS** ::= `typedef` **identifier** **DataType** ;

User-named data types are created using the `typedef` statement. X-Com does not distinguish these named types from their fully expanded contents.

**ConstantS** ::= `constant` **DataType identifier** `=` **Constant** `;`
**Constant** ::= `(`**integerC**`|`**floatC**`|`**stringC**`)` `|` **identifier** `([` *natural* `])*`
`{` **Constant** `(,` **Constant**`)*` `}`

`constant` creates a named constant of a specified type, and sets it to a value or an array of values. Array constants are surrounded with braces `{ }` and their elements delimited with commas. Array constants may be nested (e.g., `{{1,2},{3,4}}`).

### 2.4.5 Platform syntax

The `platform` statement is used to identify platforms, their class hierarchy, the configuration options for resources of their type, and the available block implementations for that platform.

**PlatformS** ::= `platform` **stringC** `[:` **stringC**`]`
`[` `{` `(`**ImplPS** `|` **ConfigPS**`)*` `}` `]`
**ImplPS** ::= `impl` **identifier stringC** `;`
**ConfigPS** ::= `config` **DataType identifier** `[=` **Constant**`]`
`(,` **identifier** `[=` **Constant**`])*` `;`

The `platform` statement associates library and user-supplied implementations with blocks. The first string constant specifies the name of the platform. The platform can optionally derive the implementations and configuration of another platform by specifying a second string constant.

Specific implementations are attached to blocks using the `impl` statement. The block name is specified, followed by the function identifier. Function identifiers are specific to their native language.

`config` works similarly by indicating possible configuration options; the set of valid configuration options is particular to the code generator used for that platform.

### 2.4.6 Block syntax

**BlockS** ::= `block` **identifier** `{ (` **PortBS** | **ConfigBS** | **BlockInst** |
| **EdgeBS** | **SplitBS** | **MergeBS** `)* } ;`

Blocks are the abstract processing elements with which algorithms are specified in the X Language. The `block` statement encloses a description of a single block. Describing a block does not create the block; only blocks specified by the `use` statement, or subcomponents of those blocks, are actually created.

**PortBS** ::= `(input|output)` **DataType identifier**
`(,` **identifier**`)* ;`
**ConfigBS** ::= `config` **DataType identifier** `[=` **Constant**`]`
`(,` **identifier**`[=` **Constant**`]) ;`

Within a `block` statement are port declarations containing the type and name of input and output ports, and one-time configuration inputs. These are indicated by the `input`, `output`, and `config` keywords, respectively. Each declaration is followed by the expected data type and a unique identifier. `config` ports may optionally include a default value provided by a constant value or a named constant with optional indexing.

**BlockInst** ::= **identifier BlockIdent** `[`**BlockOpts**`]`
`(,` **BlockIdent** `[`**BlockOpts**`])* ;`
**BlockIdent** ::= **identifier** `['[' natural ']']`
**BlockOpts** ::= `'('` **identifier** `=` **Constant**
`(,` **identifier** `=` **Constant**`)* ')'`

Blocks may also contain other blocks by declaring an instance using the block's identifier and an instance name. Configuration may be provided by following the identifier with arguments in the form `SomeBlock(config1=const1, config2=const2)`. Same-type blocks with different names may be declared by separating the identifiers with commas. Same-type arrays of $n$ blocks may be declared by immediately following the identifier with `[`$n$`]`. The array `b[3]`, for instance, will consist of block instances `b[1]`, `b[2]`, and `b[3]`.

### 2.4.7 Edge Syntax

```
EdgeBS        ::=  (Port -> (DefaultEdge ->)* Port) ;
DefaultEdge   ::=  DefaultPort (-> DefaultPort )*
Port          ::=  DefaultPort | BlockIdent . identifier | identifier
DefaultPort   ::=  BlockIdent
SplitBS       ::=  (DefaultEdge ->)* Port =< CompoundPort ;
MergeBS       ::=  CompoundPort >= (DefaultEdge ->)* Port ;
CompoundPort  ::=  Port | { Port (, Port)* }
```

Edges may be created between block instances within a compound block. The most explicit form for an edge is:

```
blockA.outportX -> blockB.inportY;
```

However, for convenience and readability, "default" ports may be used when the input or output is unambiguous. For example, if the only output of `blockA` is `outportX` and the only input of `blockB` is `inportY`, then the above example can be simplified to:

```
blockA -> blockB;
```

Additionally, unambiguous strings of blocks may be connected in one statement by using their default ports. Note that this is only possible if all interior blocks in the statement satisfy the requirement of having only one input and one output. Referring again to the above example, if `blockA` also has only one input and `blockB` has only one output, then a legal string of edges would be:

```
blockC.outputW -> blockA -> blockB -> blockD.inportZ;
```

The `=<` split operator allows `array` and `struct` data types to be divided into parts and distributed to multiple ports. A port list, usually a list of ports surrounded by braces and separated by commas, is provided in place of a single port for the destination. The number and order of elements in the port list must completely agree with the data type being split.

The >= merge operator is similar to split, but in the opposite direction. A port list is merged into a single port having a compound data type. If the merged port data types are different,

then a `struct` will be automatically inferred. If the types are the same, then either a `struct` or `array` will be formed, depending on the destination data type. As with split, the number order of the data types must completely agree.

Block array members must be referred to using an index subscript (i.e., `[n]`) unless they are used in a split or merge operation. In the case of a split, a non-indexed array block identifier may be used as the destination block. Similarly, in the case of a merge, a non-indexed array block identifier may be used as the source block. For example, if `E` is an array of five blocks with a scalar data type output `y`, and `F` is a single block with one input that is an array of five elements of the same data type, then

```
{ E[1].y, E[2].y, E[3].y, E[4].y, E[5].y } >= F;
```

is equivalent to

```
E.y >= F;
```

The input to a compound block can be provided directly to the input of a block within that same compound block. The same holds for outputs.

The split operation `=<` and merge operation `>=` are used to trivially separate and collate compound data types. Splits allow an edge to connect an output of type `array<`$T$`>[`$n$`]` on a source block to an input of type $T$ on an array of $n$ blocks. Similarly, merges allow outputs of type $T$ on an array of $n$ blocks to connect to a single input of type `array<`$T$`>[`$n$`]`. When a split or merge operate on a `struct<>` data type, the semantics are the same over the heterogeneous data types.

The behavior of splits and merges on edges is intended to be as transparent as possible. Once data is transmitted on the source of the edge, it is to be ready on the destination of the edge without preference to any single element, in the same manner as an equal number of simple one-to-one edges.

This particular method of splitting/merging data was included in the language because it was assumed to be a very common, straightforward operation that would quickly become tedious without compiler support. Other methods of splitting and merging data (time-multiplexing as well as array concatenation) exist as well. These methods were assumed to be too numerous and not straightforward enough to include in the language. In these cases, it is necessary to create a block implementation to perform the more complicated operation, by creating a block with the necessary input and output ports and hand-writing the operation in each desired implementation language. More split and merge techniques may be incorporated into the language if it becomes clear that they are commonly encountered and too inconvenient to write by hand.

### 2.4.8 Generation syntax

The remaining syntax, found below, is used to define the resources on which structures from the X algorithm are placed, and to perform the mapping operation for code generation.

**UseS** ::= `use` **BlockInst** [**identifier**] ;

The `use` statement is used to indicate all block hierarchies that are to be actually synthesized by X-Com. Only a single instance of the highest-level block containing all subcomponents should be used to create a single fully connected algorithm. Multiple use statements are permitted to implement multiple top-level blocks. An optional name may be given to the architecture; otherwise, the name will be the same as the instantiated block's.

**ResourceS** ::= `resource` **ResIdent** : **stringC**
[ **ResOpts** | { **ResOpts** (, **ResOpts**)* }] ;
**ResOpts** ::= '(' (**identifier** = **Constant** ,)* ')'
**ResIdent** ::= **identifier** ['[' *natural* ']']

Actual computation and interconnect resources (CRs and IRs) available to the code generator are specified using the `resource` statement. This statement identifies a name for the resource or resources, and the type of resource (from `platform` or `linktype`). In this

statement, the user may also provide configuration information. For scalar (non-array) resources, only a single configuration of the form `(configA=valueA, configB=valueB, ...)` is accepted. Resource arrays may be configured identically using a single configuration, or differently by nesting configurations within an additional pair of braces. An empty (`{}` or missing) configuration is also accepted in all cases. Resource configuration must specify all configuration items from `platform` or `linktype` that lack a default value.

**MapS** ::= `map { [`**FullIdent** `(, `**FullIdent**`)* ] } to` **ResIdent** `;`
**FullIdent** ::= `(`**BlockIdent** `.)*` **BlockIdent**

Blocks and edges in the used algorithm(s) are mapped to resources using the `map` statement. Using `map`, a CR identifier is associated with a set of one or more blocks, or an IR identifier is associated with a set of one or more edges. Blocks and edges are identified as fully specified children of the toplevel block instances provided by the `use` statement. Multiple `map` statements for the same resource identifier are permitted and will be merged. If a non-terminal block (i.e., a block containing other blocks) is specified, then all contained blocks (if a CR) or edges (if an IR) will be assigned to that resource as well.

### 2.4.9 Behavior

The X Language is strongly, statically typed with respect to the data types of blocks' ports. Types are checked during compilation and may not necessarily be re-checked when synthesizing block implementations in the native languages (although most languages provide some partial support for this).

Any named types (corresponding to `typedef` statements) with the same fully expanded type may be used interchangeably. For instance, in the following example, types `T` and `U` are identical:

```
typedef array<unsigned8>[8] S;
typedef array<S>[4] T;
typedef array<array<unsigned8>[8]>[4] U;
```

However, simply having the same number of elements of the same basic type will not guarantee type compatibility. In the statement below, the type `V` is not compatible with either type `T` or `U`, above, even though they all contain 32 `unsigned8` elements:

```
typedef array<array<unsigned8>[4]>[8] V;
```

An exception to the type checking is allowed in the case of variable-length array outputs connected to static-length array inputs. An output port of type `array<T>[n]` may be connected to an input port of type `array<T>[*]`, but the reverse is not permitted.

To increase the flexibility of the X Language and enable more complex structures to be created, all constants are evaluated after the initial parsing of the language file. This includes but is not limited to the special constants, block and constant array indices, and `constant` assignments.

The utility of this is apparent in the block instantiaton expression:

```
constant array<unsigned8>[5] c_array = {5, 3, 1, 4, 2};
Block myBlock[5] (c = c_array[@BLOCKRANK]);
```

`myBlock[1]` will be configured with c=5, `myBlock[2]` with c=3, and so on.

An important feature in the behavior of the `platform` statements is that implementations for a specific platform may be distributed across any number of equally identified `platform` statements. For instance, if an implementation for a new block `Foo` has been written for platform `Bar`, then a new `platform "Bar" { impl Foo ‘‘someFunction’’ }` statement may be written without modifying any previous `platform "Bar" { ··· }` statements:

This ends the specification of the X Language. This chapter introduced the X Language terminology and constructs, from algorithm and architecture design to mapping, by incremental example and by formal specification. The next chapter will discuss the implementation and use of X-Com, a compiler for the X Language, and the two currently supported generation targets, `C` and `HDL`.

# Chapter 3

# Using the X Compiler

The central program that processes X Language files into deployable executables is X-Com, the X compiler. This chapter examines the design and function of X-Com and its code generation components. The introduction of X-Com components is divided into three categories, corresponding to the three primary development roles that users of X-Com will take. This division is discussed in the following section.

## 3.1 Development Roles

X-Com may be used in a variety of capacities, and as such, there are numerous types of users who will use the compiler and its codebase. These users can be grouped into three broad categories or "roles":

1. Application authors, who develop applications entirely in the X Language and simulate or deploy them on supported processing architectures.
2. Block implementation programmers, who write the platform-specific code (e.g., C or HDL code) to perform the tasks encapsulated within X Language blocks.
3. System extension programmers, who build upon the X-Com codebase to enable greater functionality and device support.

The knowledge required to perform each role is incremental; block implementation programmers must be competent in the steps performed by application authors, and system extension programmers must be capable of the block implementation interfaces as well as authoring procedures.

Application authoring involves developing X algorithms by creating networks of blocks and edges, describing combinations of resources into processing architectures, and deploying the application to real devices or simulations of such devices. Application authoring is envisioned as the most common operation when doing X application development. Thus to further overall ease of use, little knowledge of the entire system is required of these developers. This role is discussed further in Section 3.2.

Block implementation involves writing, for instance, C or HDL code implementations which perform the tasks described by blocks. Users taking on the block implementation role must be knowledgable of the API used to realize the input port, output port, and configuration constructs in the respective implementation language(s). In general, it is also beneficial to the users to understand at a high level the way in which the blocks are then connected to each other (e.g. in terms of implied structure in HDL implementations and scheduling and memory management in C implementations). This role is discussed further in Section 3.3.

System extension involves development of the X-Com codebase itself. Users who extend the X-Com system must know to some degree how the compiler functions, particularly the overall order of operation and the design of the code generation components. We expect that the most common type of system extension will be creating support for new interconnect resources, followed by support for new classes and subclasses of platforms. This role is discussed further in Section 3.4.

## 3.2 X Application Authoring

Users authoring applications entirely within the X Language need only assume the basic X dataflow model introduced in Chapter 1. In this model, blocks execute independently of one another, and edges transmit data directly from output port to input port. All edges incorporate "infinite" queues, which simplifies the syncronization and coordination required of the algorithm. Block port semantics (e.g., how many data are consumed/produced in a "firing" of a block) are not controlled from the X Language, however this information must be communicated between the X application author and the block implementor.

X application authors develop an X Language file description of the algorithm and the processing architecture. Authors then run X-Com on the language file to generate nearly-executable code for the general processing architecture, then run the X-Dep deployment tool on that code to create and deploy the final executables.

During execution of X-Com, the X Language input file is parsed into an internal representation. Syntax errors and many semantic errors are detected and reported. These include such errors as data type or array inconsistencies and missing edges. The precise mapping of program entities (blocks and edges) to code generation classes is performed, using the guidelines provided in the input file. Using this mapping, code is generated for each computation resource which can be transformed into the appropriate executable format for that platform.

### 3.2.1 Deployment

Whereas *resources* are abstract entities to which X Language objects are mapped, *devices* are the real-world entities on which the application is actually executed. As mentioned in Chapter 1, the X-Dep deployment tool is used to link the code generated for compute resources by X-Com into programs that can be executed on either real devices or simulated/emulated devices.

An X Language file is used to describe the devices (both computation and interconnect) available to the system. This file is generally kept separate from the application description file, as it does not change except with the installation or reconfiguration of the devices in the real-world architecture. Also, by keeping these separate, it becomes easier for mapped X applications to be shared among developers with different sets of devices. For instance, an application mapped to a cluster of ten (generic x86) processors can be trivially redeployed to any cluster with at least ten such processors, because the local programming details such as IP and bus addresses, and simulation details such as clock frequency, can be kept within the deployment description.

The X-Dep tool is currently under development, and its syntax is not yet settled. Current plans use a nearly identical syntax to the resource declaration. This enables users to move device configuration into either resource or device description in an arbitrary manner, increasing the flexibility of parameterization.

The actual mechanism used to perform the deployment is an automatically generated Makefile script. The X-Dep Makefile will invoke the final executable compilers with the linking options required to generate device-specific executables. It will also provide rules that run the X-Sim simulation of the application, or distribute and execute the application on the real system.

### 3.2.2 Usage

X-Com is invoked in the follow manner:

```
xcom source1.x [source2.x [source3.x ...]  ]
```

The X-Com executable (`xcom`) is passed a list of X Language source files. These files are each parsed, in order, to define and instantiate the entities used by the compiler. Recall that the C pre-processor is applied to each source file before parsing, so each individual X Language file may also include other X Language files (i.e., with the "`#include`" directive).

Future work may likely introduce numerous command line options, such as defining symbols and setting configuration parameters from outside the X Language description. See Section 5.3 for more on future work.

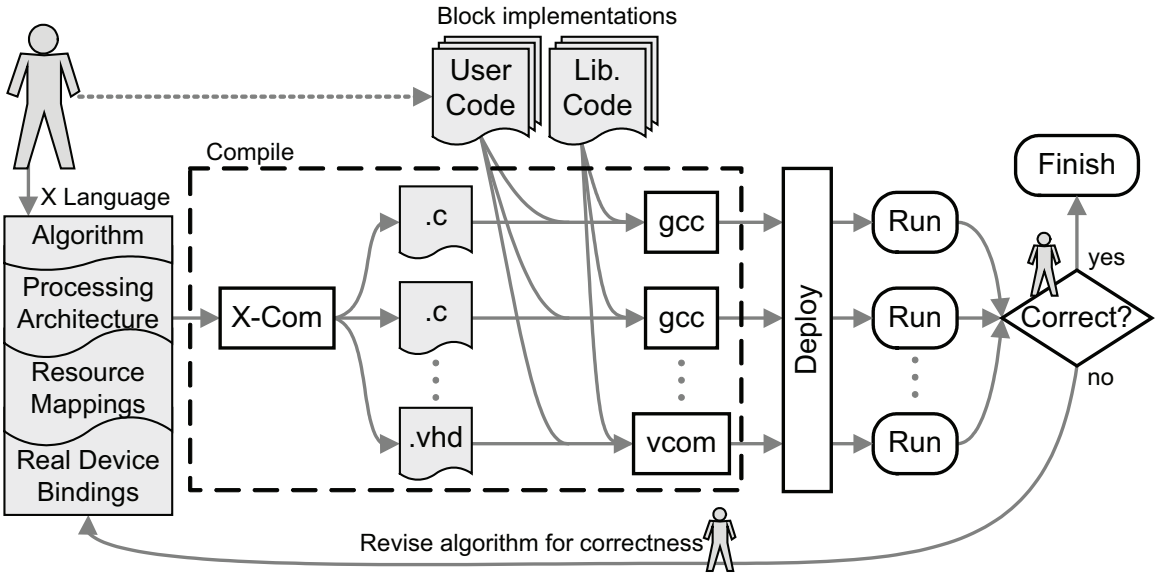



Figure 3.1: X-Com flow (same as Figure 1.8)

After the input files have been parsed, X-Com generates a source file output for each compute resource (e.g., `processor_1.c`, `fpga_3.vhdl`). These outputs are then be compiled by the appropriate compiler (e.g., the GNU C Compiler, or an HDL synthesis compiler). Compilation generally requires a language-specific "helper" file defining useful operations (e.g., `X.h` for C and `xpkg.vhdl` for HDL). It also requires the proper linking to system-specific libraries and system resources. This device-specific compilation step, the deployment of executable files, and the execution of the system, are currently performed by a manually generated Makefile. Future work, discussed further in Section 5.3, will employ the X-Dep deployment tool and a Makefile generator to automate the processor of device-specific compilation, deployment, and execution.

### 3.2.3 Support Tools

Additional tools are provided to aid the user in developing X applications. Currently, these support tools include an algorithm graph visualizer, and a generator for block implementation skeletons. More productivity support tools may be developed in response to feedback from users.

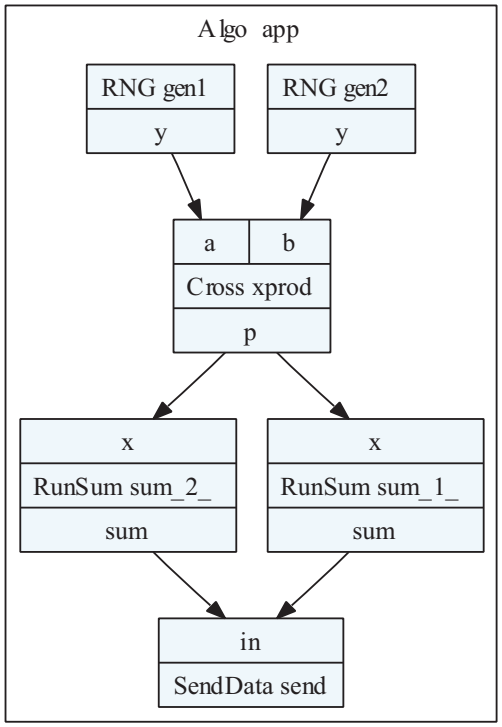



Figure 3.2: Visualization of the Section 2.2.2 example using X-Viz

The graph visualizer, X-Viz, takes X Language descriptions as inputs, and outputs a "dot" format file for use with the *graphviz* graph visualization package [10]. Figure 3.2 shows an example of using X-Viz on the example from Section 2.2.2. X-Viz displays the blocks and edges of the instantiated (i.e., with the "use" statement) algorithms, and can optionally show edge labels and data types, and the final post-parsing structure before code generation (see Section 3.4). Figure 3.3 demonstrates a portion of the visualization of a large algorithm.

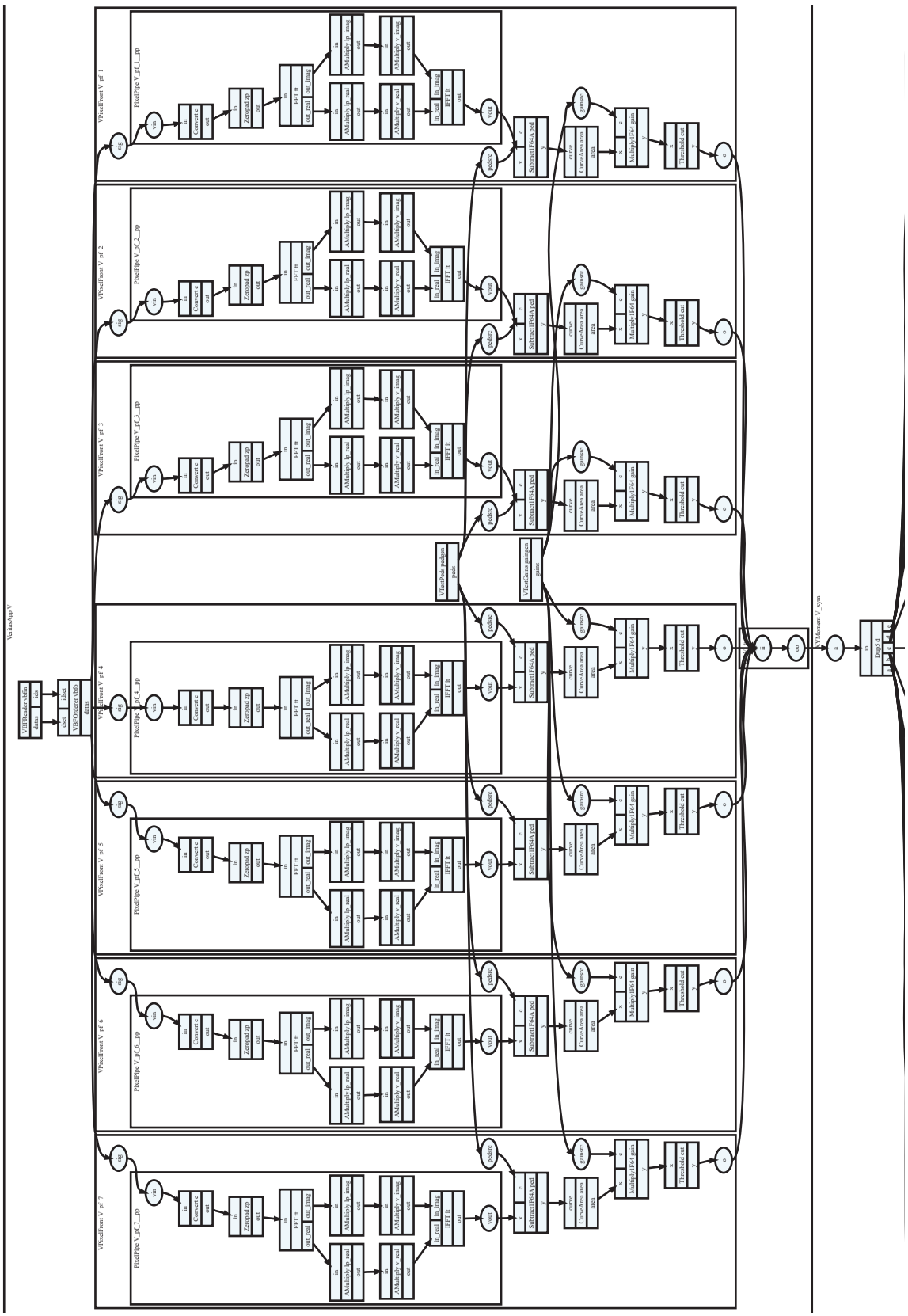


Figure 3.3: A large algorithm visualization using X-Viz

Another tool is available that helps users quickly get started writing block implementations. This tool, currently named "`impgen`," generates the skeleton code that provides the API for a specified X block in C or VHDL. It also can optionally rename input and output variables to their X port names, which is particularly valuable when using array structures in VHDL. Using `impgen`, one can avoid the tedious writing of implementation-specific interfaces and data structures using the information already available in the X block description. Users then simply fill in the remaining functions (in C) or process body (in VHDL) to perform the appropriate action. `impgen` produces code for the most general case, so it is up to the user to make the minor alterations for their particular implementation when desired.

## 3.3 Block Implementation Programming

Block implemenation programmers assume the same basic dataflow model as application authors, but no longer treat blocks as a simple abstract task. An individual block implementation must conform to the abstract task in two ways:

- Implementations must provide certain structures and language constructs that express the interfaces (inputs, outputs, and configuration) of their respective block using the appropriate language-specific API.
- Implementations must semantically operate in a way that is expected and explicitly or implicitly expressed to application authors. For instance, it must be clear to application authors if a block with two inputs consumes either input as soon as it is available, or if the block consumes both inputs at the same time.

The latter can be communicated implicitly (e.g., a simple "add" should consume all its inputs simultaneously to produce one output) or explicitly (e.g., through comments in the X block declaration). The former is the topic of the next two sections, which describe the APIs used for interfacing block implementations written in C and HDL.

### 3.3.1 The C Code Generator and API

The C code generator is the component of the X compiler responsible for creating the top-level files that implement a mapping of C-compatible (e.g. C, C++, and wrappers around certain languages) block implementations to "C" platforms. These platforms may include most modern software resources, simulation of resources, and the default "generic" mapping used for testing. Currently this is the only software generator, although more may be developed in the future (see Section 5.3). This section examines the application programmer interface (API) for user-supplied code, an overview of the code generator's function, and some semantics and limitations of the current implementation.

The API for a block consists of four functions and a state-storing structure. The code generated by X-Com links to each of these functions, so they must be defined in the user-supplied or library code. Figure 3.4 provides an example of these constructs. The functions perform initialization (`X_BlockA_init`), finalization (`X_BlockA_destroy`), and data-synchronous and data-asynchronous operations (`X_BlockA_push` and `X_BlockA_go`). The data-synchronous push operation is called when an external entity transmits ("pushes") data on an edge connected to one of the block's inputs; most function-like operations use push alone. The data-asynchronous go operation is called on a potentially indeterminate schedule; I/O and scheduler blocks will generally use asynchronous operation.

The state structure contains a performance-capturing structure (`clock`), port status register (`portmask[]`), and pointers to the block's interface ports (inputs, outputs, and configuration). The input variables (`iport`$N$) are made available by the API immediately before the push function is called, whereas an output variable (`oport`$N$) must be valid and available to the API before transmitting data on any output port. Configuration variables, named after their X block identifiers, are initialized before any block functions are called, and may be used by the block at any point.

Memory management of communication data in X's C API is handled by the blocks themselves, with some assistance from the system. Data are transmitted from and to a block as

```
block BlockA {
        input FLOAT64 x;
        input FLOAT64 y;
        output FLOAT64 out;
        config UNSIGNED32 order;
};
```

```
struct X_BlockA_data {
        Xclock_t clock;
        // pointer to BlockA's send function
        // this is called to transmit data to downstream blocks
        void (*send)(int);
        // pointer to BlockA's release function
        // this is called to release inputs (and optionally free data)
        void (*release)(int,char);
        portmask_t portmask[1];   // bitmask of ports with available data

        FLOAT64 *iport0;   // input port: x
        FLOAT64 *iport1;   // input port: y
        FLOAT64 *oport0;   // output port: out
        UNSIGNED32 *order;   // config: order
};

// initialization -- called on startup
void X_BlockA_init(struct X_BlockA_data *) { ... }
// destruction -- called on completion
void X_BlockA_destroy(struct X_BlockA_data *) { ... }
// push -- called when data is received on any port
int X_BlockA_push(int p, struct X_BlockA_data *) { ... }
// go -- called by the main loop
int X_BlockA_go(struct X_BlockA_data *) { ... }
```

Figure 3.4: Example X Block and corresponding X Language C API structure

a pointer to memory containing the appropriate data structure, with typical expectations on how the pointer is treated:

- When data is received, the block may read, modify, resize, and free the segment (as long as it is only freed once).
- To transmit data, a pointer to the beginning of a valid allocated memory segment must be provided. After using `send` to transmit, that segment may no longer be accessed by the block.

- Block implementations with both input(s) and output(s) may choose to reuse input pointers as output as long as the data structures are the same (or they are resized/restructured appropriately). This can greatly reduce the overhead incurred from unnecessary memory management operations.
- As C does not implement garbage collection, it is necessary for any received memory allocations to be either freed or sent, to avoid "memory leaks."

All memory operations (allocate, free, and resize) are performed by the GLib (discussed further in this section) memory management functions such as `g_malloc`, `g_free`, and `g_realloc`. These functions are both syntax compatible and run-time compatible with the ANSI C memory functions (i.e., `malloc`, etc.), but the GLib wrapper aids in the debugging of the generated and user-supplied code and integration with GLib data types.

Two functions are provided to the block via its state structure: `send()` and `release()`. The release function is used to indicate that the block has finished using the data made available to it on a input port, indicated via the function argument. Calling the function will clear the specified port's bit in the portmask (if there is no more data on that port) and optionally free the memory buffer for the received data on that port. The send function pushes data out of a single port, indicated via the function argument. Data is indicated by the pointer provided in the port's `oport` variable; after sending, the block is no longer permitted to access the pointed-to memory. Figure 3.5 demonstrates a sequence diagram of typical edge communications using the send and release functions. The figure depicts the above function calls surrounding the BlockB push function. The surrounding functions are not shown at this point; these are discussed further in this section (see Figures 3.6 and 3.7).

Both generated C code and the API make use of the GLib [26] low-level portability and utility library. The generated code uses GLib data structures for memory management and queuing to facilitate the queueing system on which the data flow organization relies. Within the API, GLib data types are used for safety and robust management of non-trivial data structures such as arrays. In particular, the `GArray` structure is used for both constant

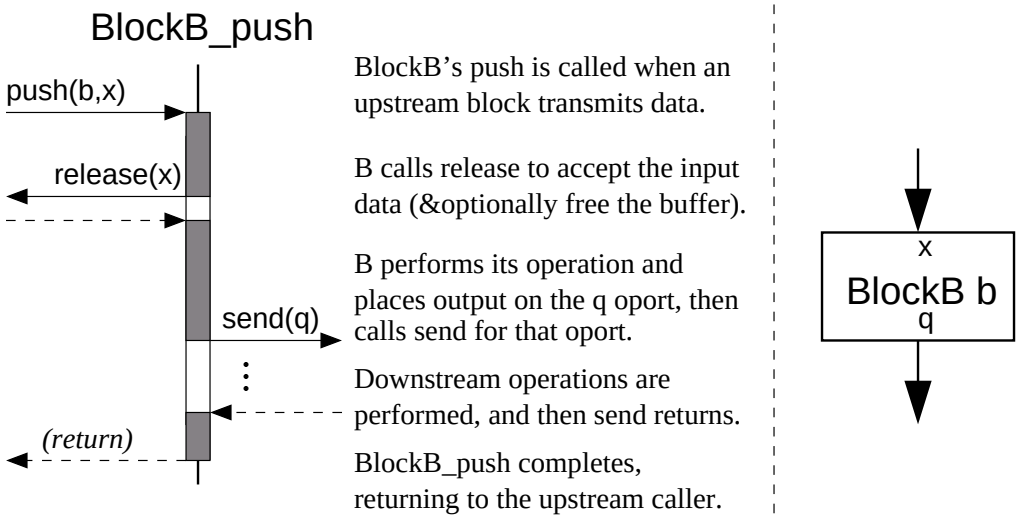



Figure 3.5: Sequence diagram of edge communication using the C API
With time in the vertical axis, the shaded portion indicates the thread of execution. Solid arrows are function calls, and dashed arrows are returns.

and variable sized arrays. Table 3.1 provides an overview of the currently implemented C data types, including their X counterparts.

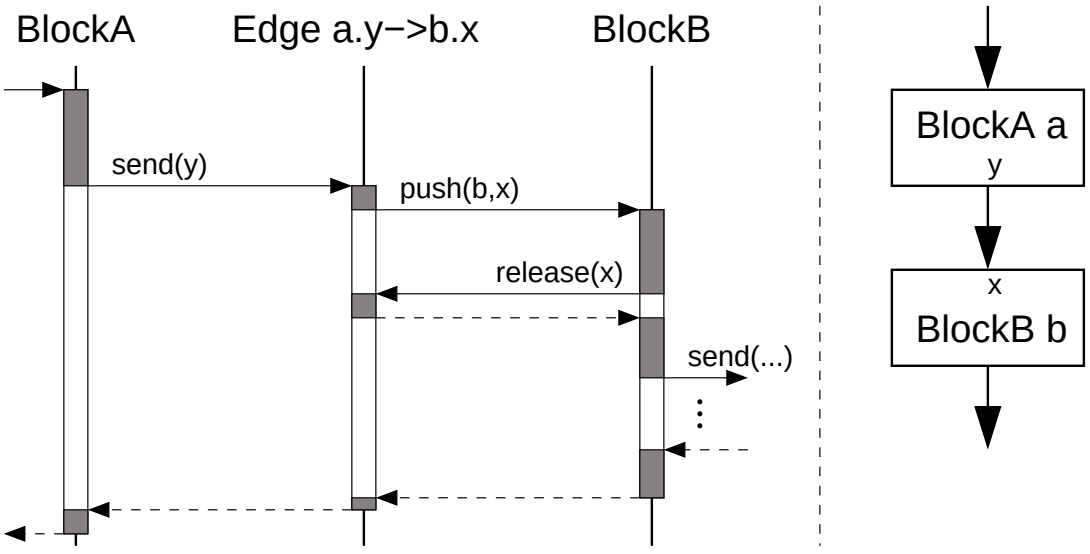



Figure 3.6: Sequence diagram of communication between two blocks in one processor

Table 3.1: C API Data Types

| X Type | C Type |
|---|---|
| `UNSIGNED8/16/32/64`<br>`SIGNED8/16/32/64`<br>`FLOAT32/64/128` | Pointer to `typedef`'d type of the same name |
| `STRING` | `STRING`, which is defined as `char*` |
| `ARRAY<BasicType>[`$n$`]` | `GArray*` with `len=`$n$ and elements of type BasicType |
| `ARRAY<ARRAY<>[]>[`$n$`]` | `GPtrArray*` with `len=`$n$ and elements `g_pointers` to `GArrays` |

Figure 3.5 provided a sequence diagram for the operations surrounding one block. Two blocks can be contained within a single processor, however, and Figure 3.6 provides an example of the connected functions created by such a mapping. The X-Com-generated code for each edge will connect the outputting block's `send` function to the inputting block's `push` function, and perform the necessary queuing operations in-between.

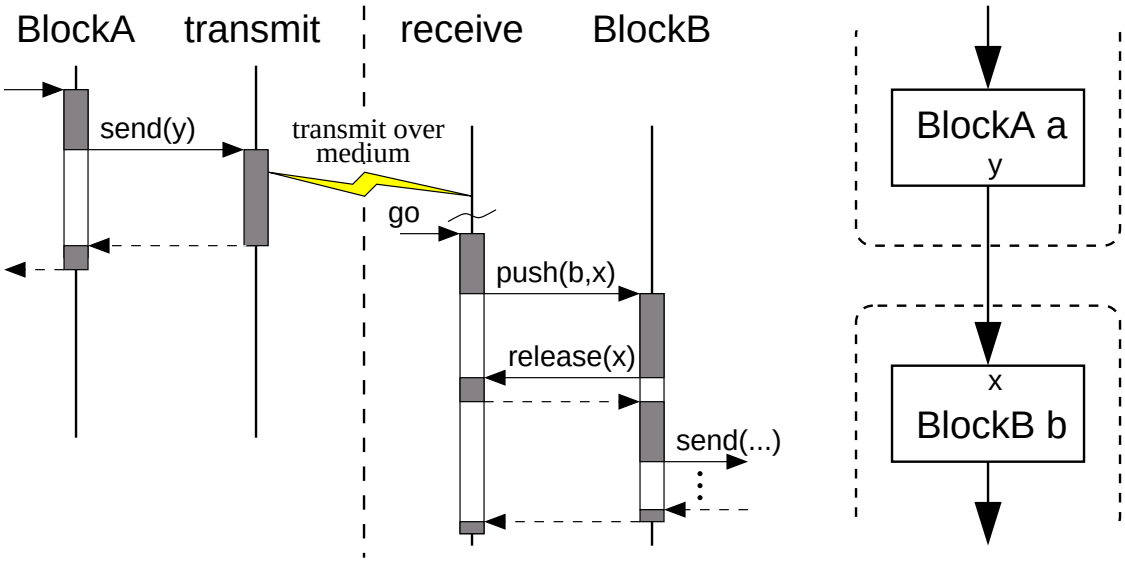



Figure 3.7: Sequence diagram of communication between two blocks on two processors

Figure 3.7 demonstrates how the communications between these two blocks would look if they were mapped to two different processors. Note the opportunity for parallelism that

has been introduced in this multi-processor execution sequence. In this diagram, BlockA's `send` calls the medium-specific transmit operation, which sends data to the second processor. This data is received by system buffers and is retrieved by the medium-specific receiving operation once its `go` function has been called (scheduled round-robin in the same thread as the other blocks' `go` functions). After it has been retrieved, BlockB's `push` is called in the usual manner.

### 3.3.2 The HDL Code Generator and API

The HDL code generator component generates VHDL top-level entities suitable for simulation or deployment on an FPGA. The generated code is a VHDL entity, however code written in any compatible HDL (e.g., Verilog or AHDL) may be instantiated, using the standard `port` and `generic` interface (e.g., `input`, `output`, and `parameter` in Verilog). Currently, the top-level entity is entirely self-contained, as inter-resource communications are only implemented using a simulated file system interface. Future interfaces such as a PCI bus DMA engine will be supported and connected to the device pins using the X-Dep tool, permitting the use of actual (non-simulated) FPGAs on supported development boards.

The HDL API to user-created blocks consists of a set of three signals, for each block-input, `input`, `read`, and `avail`; two signals for each block-output, `output` and `write`; and a back-pressure signal. Figure 3.8 depicts the interaction handshake of a block with one input (`x`) and one output (`f`) with the API. When data is available on the block's `input_x` input, the `avail_x` signal is asserted. The block then responds, when it is ready and has made use of the input data, by asserting `read_x`. In the clock cycle following the `read_x` response, the `avail_x` input goes back to low and `input_x` is no longer valid — unless more data is available on the same input, in which case `avail_x` will remain high during the cycle.

Following the processing within the block implementation, the block makes data available on its `output_f` port and asserts `write_f`. The data is queued on the block-output in the first rising clock-edge following the `write_f`, and the block must deassert `write_f` during that cycle unless it wishes to queue a second datum on the block-output.

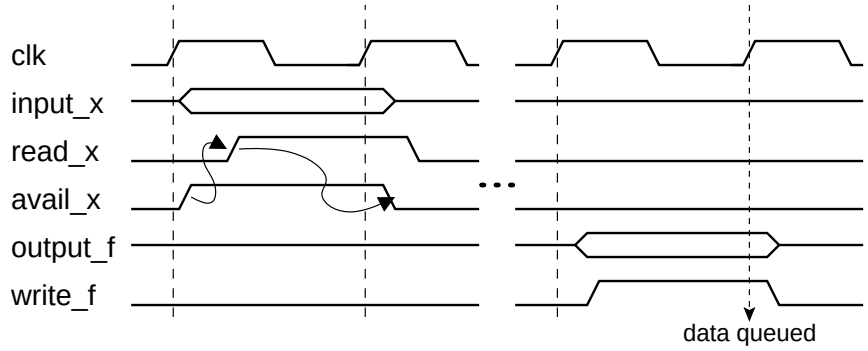



Figure 3.8: Timing diagram of the HDL API

The back-pressure signal to a block is created from the logical-OR of the "almost-full" signals on the output queues, and any other conditions that would require the block to stop producing output. The handling of back-pressure is simplified to the block writer, however, because the signal is also delivered to the entities containing the block's input queues. These entities will not deliver more data to the block under back-pressure conditions. This means that blocks without deep pipelining can actually ignore the back-pressure signal and rely on the API to handle those conditions, reducing the design complexity and area required within the block implementation.

An HDL example parallel to the C API sequence diagram (Figure 3.5) is provided in Figure 3.9. Here, the generated infrastructure connects BlockA to BlockB, with each block having one input and one output. The block-input and block-output signals for each edge are shown connected to each edge's FIFO queue structure. Not shown is the nontrivial logic, described functionally above, that generates the `avail` signals and the backpressure signals. Note that a common clock signal (`clk`) and reset signal (`rst`) are connected to every component. The current implementation requires that all communication take place under control of a common clock. Blocks may internally use a faster or slower clock, however they must still interact at the global rate.

The data types used in the HDL API and their corresponding X type are shown in Table 3.2. The basic data types directly follow from their names, with IEEE 754 and 854 floating

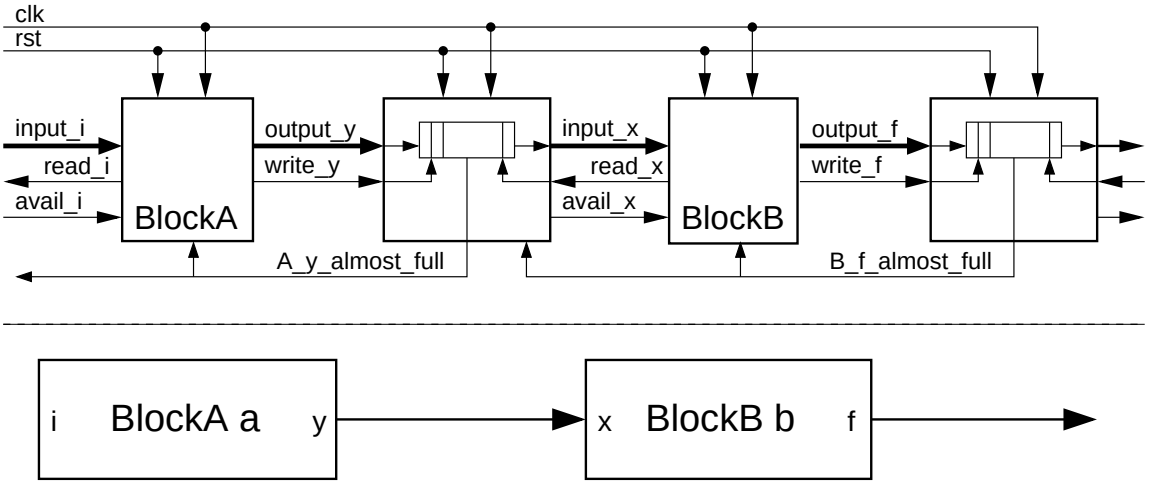



Figure 3.9: Interface diagram of edge communication using the HDL API

point values being implemented using the IEEE 1076.3 (VHDL200x) floating point extension to VHDL. Arrays of basic types are treated as wide concatenations of their contained data. Automatically generated utility functions are relied upon to make accessing their contents sufficiently convenient. Arrays of array types are transmitted in sequence; the first subarray is sent, then the second, and so on. The choice of treating multi-dimensional arrays differently from single-dimension arrays was made for convenience and efficiency in initial test applications. Further development will support a more flexible treatment of array data types in generated VHDL, with respect to the parallel or serial transmission of data.

Table 3.2: HDL API Data Types

| X Type | VHDL Type |
|---|---|
| `UNSIGNED8/16/32/64` | `unsigned(7/15/31/63 downto 0)` |
| `SIGNED8/16/32/64` | `signed(7/15/31/63 downto 0)` |
| `FLOAT32`<br>`FLOAT64`<br>`FLOAT128` | `float(8 downto -23)`<br>`float(11 downto -52)`<br>`float(15 downto -112)` |
| `ARRAY<BasicType>[`$n$`]` | `std_logic_vector` of concatenated data, ascending index |
| `ARRAY<ARRAY<>[]>[`$n$`]` | sequential packets of the subarray type |

Block configuration parameters in the HDL API are passed in at HDL compile-time as generics (VHDL) or parameters (Verilog). The types used for block configuration are the same as for ports, as described in Table 3.2. Note that this interface requires that blocks with run-time configuration be configured using standard input ports, rather than configuration "ports," as compile-time generics cannot change during execution.

## 3.4 X System Extension

Users who extend the X system are no longer able to treat the system as an ideal dataflow model, however they must ensure that their extensions do not break the uniform development interface and simplified programming model provided to other users.

Users in this class might be interested in modifying nearly any part of the X compiler. Certain operations, however, are expected to be relatively more common. Thus, efforts have been made when writing X-Com to make such extensions easier by employing clear and concise interfaces with the other components. Examples of common extensions include:

- Addition of new interconnect resource type implementations for existing platforms.
- Addition of new platforms based on other platforms.
- Addition of entirely new platforms.
- Moderate changes to C and HDL code generator templates.

This section presents the X-Com compilation procedure, the code generator interface, and some detail of the function of the C code generator. The HDL code generator, being structurally described and naturally parallel, is sufficiently described in the previous section. The last subsection contains an overview to the steps involved in adding new resource types.

It is important to note that the goal of this section is only to provide an introduction to the extension of X-Com. A user wishing to modify the X-Com source code will naturally have to read the relevant portion of the actual source code and become familiar with its function.

### 3.4.1 X-Com Function

X-Com operates in a series of steps to transform an X Language description of an application into a set of code ready to be deployed with X-Dep. These steps are summarized below:

1. Parse X Language file into an internal representation.
2. Form connections between identifiers and objects. This is done as a second pass to remove nonintuitive "declare before using" requirements such as in C.
3. Instantiate the block tree, with the indicated "use" blocks as the root nodes and instantiated blocks as child nodes to their containing blocks.

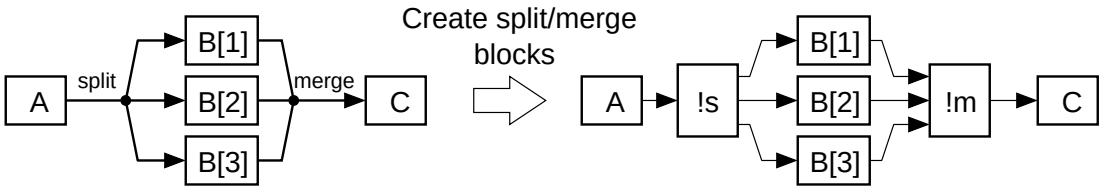



Figure 3.10: Edge collapse operation

4. Convert split and merge edges to internally generated blocks connecting simple edges, as shown in Figure 3.10. Note that the `!` character is used in the block name; this is not a legal character in user-specified blocks, so internally generated block names use this to avoid name collisions. The actual block names follow the pattern:

   `!split_Ablock_outport_Bblock_inport`.

5. Identify blocks as atomic or compound.

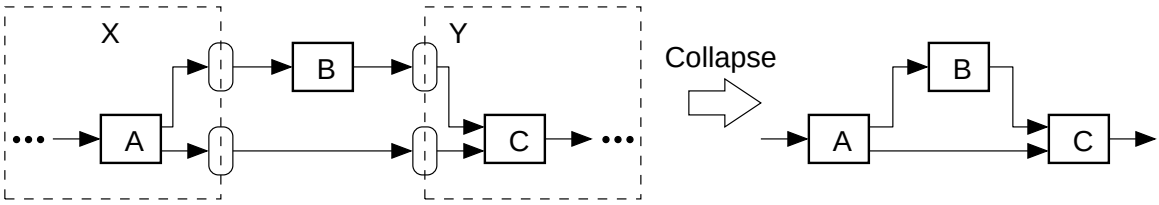



Figure 3.11: Edge simplification operation

6. Collapse edges to only single edges connecting atomic blocks as shown in Figure 3.11. This flattens the graph to a simple network of implementable blocks connected by point-to-point edges.

7. Instantiate and configure code generators for every resource, and pass the relevant blocks, internal edges, and interconnect edges to each generator.

Each code generator uses a simple interface, outlined in Figure 3.12 using an abbreviated sample of the abstract `Generator` class, from which all code generators are derived. The main compile operation determines which subclass of `Generator` (e.g. `C_Gen`, `HDL_Gen`) to create, and then instantiates such an object and attaches it to a computational resource. Then, resource configuration options are passed to the generator object with `addConfig()`, followed by the blocks (`addBlock()`) and edges (`addEdge()`) that are completely contained within that resource. Edges that cross resource boundaries are given to the generator, using `addInput()` and `addOutput()`, with a `tag` identifier that is unique for each edge using the same interconnect. Finally, the `generate()` function is called to cause the generation of source files.

### 3.4.2 The C Code Generator

C code generation uses the `Generator` interface as shown in Figure 3.12. When `addBlock`, `addEdge`, `addInput`, or `addOutput` is called, a reference to the corresponding `Block` or `Edge` object is recorded. A string is then generated for that object, describing (for edges) the edge data structure and send/release functions, or (for blocks) the state structure declaration and initialization. Functions are created for every edge endpoint contained in the

```
class Generator {
public:
    Generator(Resource *, string name);

    virtual void addBlock(const Block *) = 0;
    virtual void addEdge(const Edge *) = 0;
    virtual void addConfig(std::string name, const DataType *data) = 0;
    virtual void addInput(const Edge *, unsigned int tag) = 0;
    virtual void addOutput(const Edge *, unsigned int tag) = 0;

    virtual void generate(std::ostream &os) = 0; // os is used for runtime status output
};
```

Figure 3.12: The `Generator` code generation class (abbreviated)

computation resource, while any interconnection points are replaced by internal blocks which implement (in C) the appropriate data source or sink for that IR. The C generator's `generate()` function causes the creation of the complete C source file, combining together all the previously generated strings and creating the main execution loop.

When splits and merges are encountered by the compiler, they are passed as blocks to the generator, which can then decide how to implement the operation. The C code generator implements splits and merges as simple blocks that perform the appropriate operation and handle memory reallocation. Split blocks divide up the input `GArray` amongst the output ports. If the input is an array of basic types, then the individual outputs must each be allocated. Array-of-arrays types do not require reallocation, as each subarray is individually allocated and may thus be directly used. Merge operations operate by appending each input value to a new `GArray` or `GPtrArray`, and freeing the input subtypes if they are scalars.

Currently, the main execution loop consists of a round-robin execution of each block's `go()` function, the portion which executes independent of input. Each block may return a value indicating that it has completed such operations, and thus remove itself from the main loop. Generally after the first loop, only data source blocks will remain in the loop, as the input-synchronous blocks (which are often the vast majority) will have removed themselves. Once removed from the loop, there is no mechanism for a block to re-insert

itself back into the loop. It is important to note that there may be applications for which round-robin scheduling of the `go()` functions is not desired. The ability to specify other internal scheduling schemes is a topic for future research.

### 3.4.3 Adding Resource Types

Adding a new interconnect resource is a potentially very common operation, especially when considering embedded systems and prototyping devices (e.g., FPGA development boards) which can use a very diverse set of interconnect mechanisms to connect to other devices. Interconnect resources are created using the `LinkGenerator` class interface, which is a special case of the `Generator` class shown in Figure 3.12.

Creating a `LinkGenerator` requires writing the `addEdge` function, which updates the internal variables of the `LinkGenerator` and is called for every edge which connects using that interconnect. The `addConfig` function is also available here to parameterize the interconnect and the endpoints (e.g., unique addresses for each resource connected to a network).

```
class LinkCode {
public:
    virtual string genSend(const Edge *e, LinkGenerator *l, Generator *g);
    virtual string genRecv(const Edge *e, LinkGenerator *l, Generator *g);
    virtual string genTop(LinkGenerator *l, Generator *g);
};
```

Figure 3.13: The `LinkCode` I/O code generation interface (abbreviated)

The `LinkGenerator` is ignorant of specific types of compute resources; to generate I/O code for individual platforms, the `LinkCode` class is used, shown in Figure 3.13. Each `LinkCode` class corresponds to *one* linktype implementation on *one* platform. The class is registered with the compiler before code generation, so that when the `Generator` finds that it must output to an interconnect, it will retrieve the implementation of the output from the appropriate `LinkCode`'s `genSend()` function (similarly with inputs using `genRecv()`). `genTop()` is called once for every `Generator-LinkGenerator` combination. Each of these functions are permitted to modify the `Generator` object, so the internal data structures

(e.g., global variables, function prototypes, etc.) may be modified as needed for `LinkCode`'s code generation.

Adding new platforms is a more involved process, however they use the same basic interface described above and shown in Figure 3.12. The C and HDL code generators are easily extended using traditional class inheritance, as their code generation templates are distributed among a set of functions, so changes to single mechanisms (e.g., internal block communication, scheduling) requires minimal repetition of code. Creating an entirely new platform is possible, although naturally much more difficult. The existing code generators provide two good examples of generation for two very different types of programming language.

# Chapter 4

# Sample Applications

This chapter presents a set of example applications written using the X Language. For each application, the algorithm design is presented, as well as a selection of mapping arrangements. Where useful, application output is provided. Finally, each section examines the performance of the application under the selected mappings.

### Measurement Techniques and Issues

All native-execution timing measurements in this chapter were made using the `RDTSC` machine instruction to read the system clock. These times are accumulated in a set of variables associated with each block; this is part of the X-Com C API profiling mechanism. Block execution times are measured from the beginning of a `push` or `go` call, until a `send` call is made or the `push` returns — during this time, only block-specific code will be executing. The time to send or receive data from another device is similarly measured, although the time to check if data is available is not recorded.

Some performance measurements in this section are made using symmetric multi-processor systems running Linux. In such systems, the Linux scheduler could cause processes to switch from execution on one processor to execution on another in an effort to balance processor load, in a process called migration. Core-to-core migration was avoided by using the Linux-specific `sched_setaffinity()` system call to set the "affinity" of the process

to a single logical processor. This causes the Linux scheduler to only schedule the process for execution on a specified subset of the available logical processors.

As all HDL execution was done in a discrete event simulator (i.e., ModelSim), the times were manually recorded by inspection of the waveform. Later work will automate this step.

Some issues arise when comparing performance measurements of real-world systems such as those gathered from native execution in this section. The effects of multitasking present one such issue, where many kernel-space and user-space programs can take processing time away from a program being measured. This is particularly an issue using the native profiling technique when an initial clock time is read, but another program is scheduled before the final clock time is recorded. The time spent running the intervening program is then measured as part of the measured block execution time. In the measurements taken in this section, this effect usually accounted for a very small portion (no more than 0.5%) of the total execution time [1]. During some measurements, however, ModelSim was running alongside the native execution. In these cases, the outlying processing times were pruned from the results, and the total execution time was extrapolated from the "clean" measurements. As a side effect, the overhead (the remaining time not accounted by any block) was not measureable and was thus left out from the results.

Where it is helpful to understanding the system, the issues involving the accuracy of measurements and nonideal performance scaling are discussed further in each application section as they arise.

[1] This is an estimated figure, based on the proportion of noticeably outlying processing times in each measurement run.

## 4.1 Triple-DES Encryption

The Triple-DES application is a short example design that performs encryption on streams of data. Encryption involves transforming unsecured "plaintext" information into encoded information under the control of a key. Triple-DES is a variation of the Data Encryption Standard (DES) that employs three DES blocks in sequence to increase the effective key size.

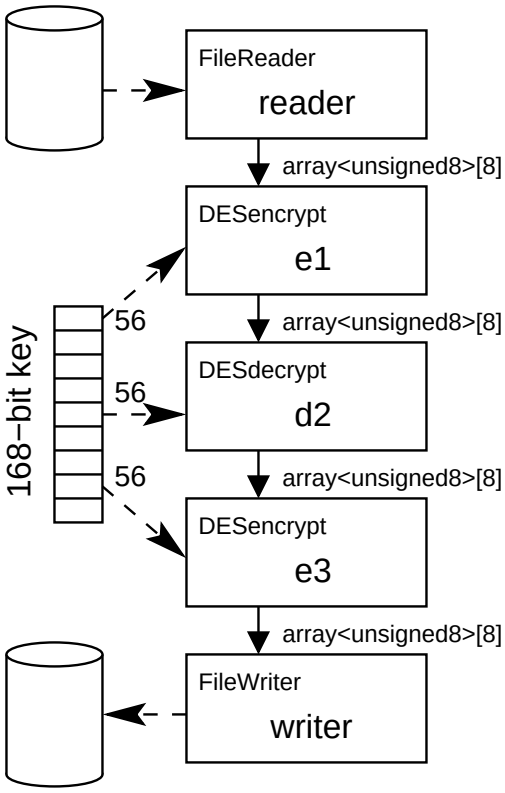
In a single DES block, a 56-bit key encrypts 64 bits of data at a time. The block divides the data into two halves which are permuted and XORed with a portion of the key. That result is reorganized and the process is repeated seven more times, eventually using all portions of the key.

To perform Triple-DES encryption, three DES blocks each operate on the data stream in sequence, giving an effective key size of 168 bits. The three blocks can be three DES-encrypt blocks, or a DES-encrypt followed by a DES-decrypt and then another DES-encrypt. The implementation of Triple-DES in this section uses the latter.

### 4.1.1 Design

Figure 4.1 depicts the design of the Triple-DES application, alongside its implementation in the X Language. To provide a sink and source to the stream, the `FileReader` block reads data from disk and outputs it a "`phrase`" (an array of 8 bytes) at a time and the `FileWriter` writes the same data type to disk. Each DES block takes the `phrase`, encrypts (or decrypts) the data using the key preloaded from a specified keyfile, and outputs it to the following stage.

To implement the application, three mappings were created (Figure 4.1) using different combinations of resources. Figure 4.2 provides the code for each of these mappings. The first mapping creates a single `C_x86` resource named `proc`, to which the entire algorithm is mapped. The second mapping also creates an `HDL_Sim` resource named `hwsim` and places




```
typedef ARRAY<UNSIGNED8>[8] phrase;
block DESencrypt {
   input phrase in;
   output phrase out;
   config STRING keyfile;
};
block DESdecrypt {
   input phrase in;
   output phrase out;
   config STRING keyfile;
};
block FileReader {
   output phrase out;
   config STRING file;
};
block FileWriter {
   input phrase in;
   config STRING file;
};

block top {
   FileReader rd(file="plaintext.dat");
   FileWriter wr(file="encrypted.dat");
   DESencrypt e1(keyfile="key1");
   DESdecrypt d2(keyfile="key2");
   DESencrypt e3(keyfile="key3");

   rd -> e1 -> d2 -> e3 -> wr;
};
```

Figure 4.1: Triple-DES algorithm flow and X code

the middle `DESdecrypt` block, `d2`, on this resource. The third mapping places all of the DES blocks on the `hwsim` resource.

### 4.1.2 Performance

The performance results found in this section were obtained through compiling the algorithm and various mappings (from Figure 4.1 and Figure 4.2, respectively) using X-Com.

The resulting processor executables were then executed on a machine containing a 3.4GHz Intel Pentium 4 processor, using the Cygwin environment running on Windows XP.

```
// mapping 1
use top;
resource proc is C_x86;

map { top } to proc;
```

```
// mapping 2
use top;
resource proc is C_x86;
resource hwsim is HDL_Sim;

resource io is File;

map { top } to io;
map { top.rd, top.wr, top.e1,
      top.e3 } to proc;
map { top.d2 } to hwsim;
```

```
// mapping 3
use top;
resource proc is C_x86;
resource hwsim is HDL_Sim;

resource io is File;

map { top } to io;
map { top.rd, top.wr }
      to proc;
map { top.e1, top.d2,
      top.e3 } to hwsim;
```

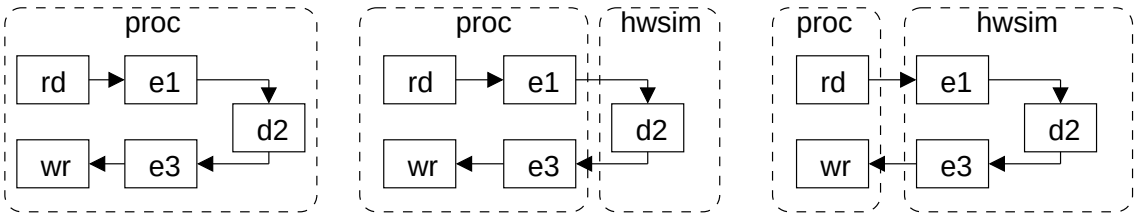



Figure 4.2: Triple-DES mappings

Hardware executables (each a VHDL entity) were executed in the ModelSim simulator, assuming a clock rate of 100MHz for hardware simulation. Communication times to send data between `proc` and `hwsim` were modeled as a constant 850MB per second, divided evenly between the sending and receiving streams [2]. The actual overhead measurements are ignored during these conditions, due to the increased inaccuracies in measuring native execution times while a simulator (i.e., ModelSim) is running.

All performance results below show the *aggregate* execution times of encrypting a single 67MB file.

Where multiple resources are used in this example (i.e. `hwsim` and `proc`), they form a pipeline. Therefore, the performance graphs presented in those cases are drawn as two bars, where the overall execution time is the maximum of the bars.

[2] This data rate has been measured in similar types of streaming applications, using a 100MHz PCI-X bus and incorporating the overhead to initiate DMA transfers.

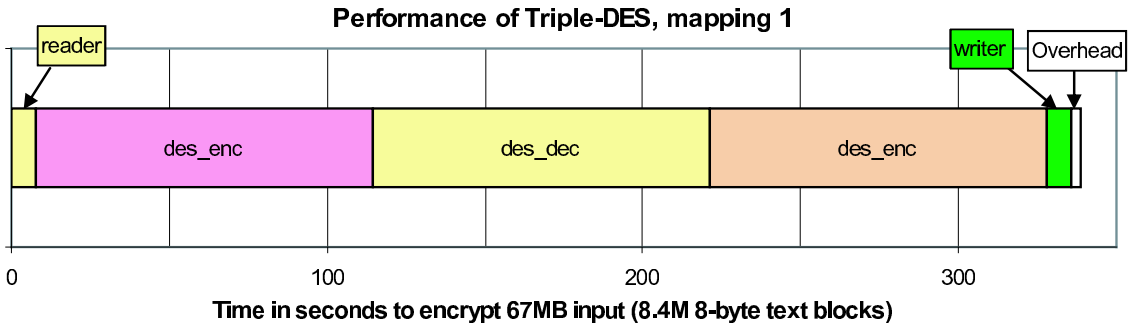



Figure 4.3: Results of mapping 1

Figure 4.3 presents the performance results of executing the first mapping on a single processor. The total runtime was 338.9 seconds, of which about 7.5 seconds were spent in each file I/O block, and 107 seconds were spent in each DES block.

The overhead time, measured as the remaining time not spent in any block, was measured at 3.3 seconds aggregated over the entire run.

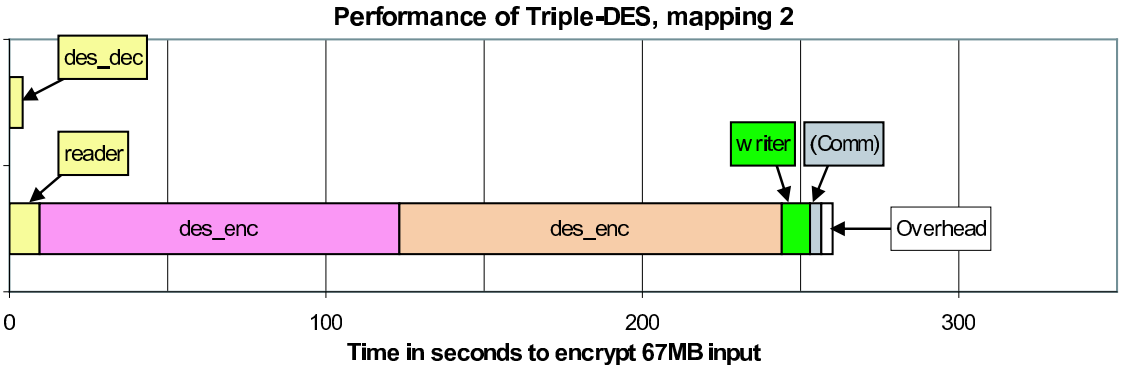



Figure 4.4: Results of mapping 2

Figure 4.4 presents the performance results of executing the second mapping, which simultaneously simulates the `d2` block in ModelSim and executes the remainder of the algorithm natively. Communication times between these two parts were modeled as described above. Overhead was not measurable in this case, due to the simultaneous simulation and execution, so this figure retains the overhead measurement from the previous results.

Total run time (using the simulated results) is 256.8 seconds, limited by the execution on the processor. The simulated hardware, running in parallel to the processor execution, spent 0.0197 seconds operating on data and 4.067 seconds in communications. The total run time under these conditions is a speedup of 1.32 over the first mapping, compared to a theoretical speedup of 1.46 if there was no communication delay.

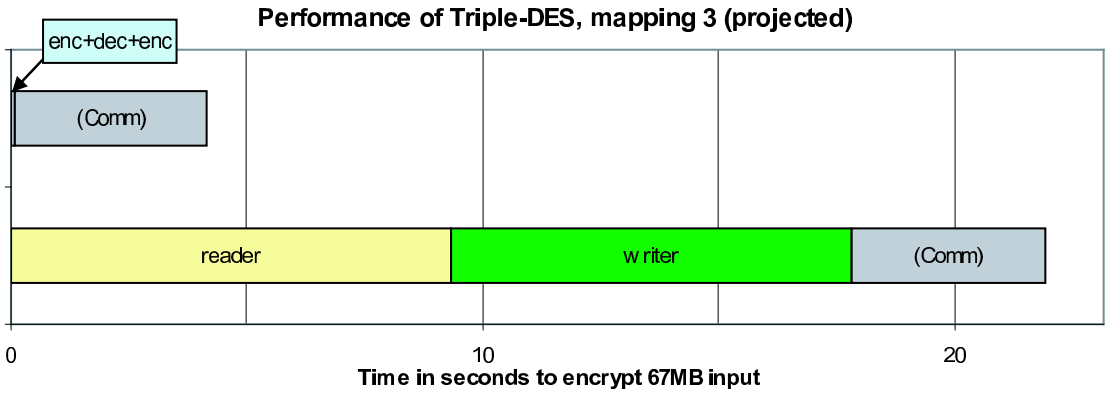



Figure 4.5: Results of mapping 3

Figure 4.5 presents the performance results of executing the third mapping under the same conditions as used in the previous mapping. In this mapping, all DES operations are performed in the hardware simulation, which is capable of pipelining the operations and thus running them in parallel. These results assume the same communication delays as before, and the removal of the non-communication overhead.

The total run time of this configuration is 21.9 seconds, and is still limited by the execution time of the processor. The simulated hardware again spent 0.0197 seconds opreating on the data, due to the pipelining within the hardware of the three DES blocks. This configuration attains a speedup of 15.5 over the single processor configuration, compared to a theoretical speedup of 18.5 if there was no communication delay.

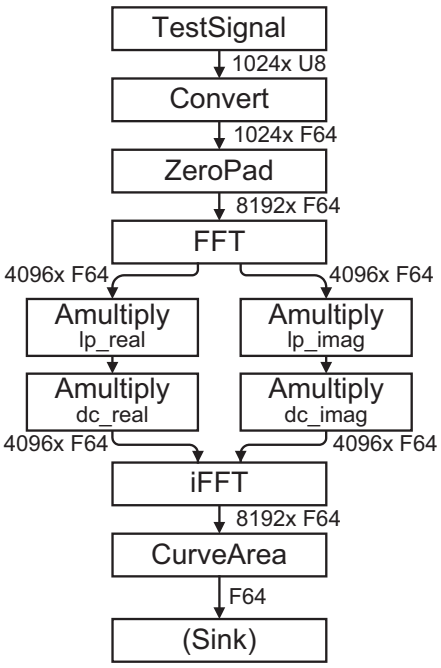
These performance results demonstrate the effectiveness of executing portions of the Triple-DES application on hardware resources, even with the inclusion of system bus communications.

## 4.2 Signal Cleaner

The Signal Cleaner application is a test design that creates a cleaned, high-resolution signal from a lower-resolution signal. This particular design was chosen because a similar structure plays a significant role in the gamma ray event parameterization application, to be presented in Section 4.3.

### 4.2.1 Design

Figure 4.6 depicts the layout of the signal cleaner algorithm as a nine-stage pipeline. The corresponding X Language code is shown on the right of the figure.




```
block SigClean {
        TestSignal genwave(noise=20.0);
        Convert convert;
        ZeroPad pad;
        FFT fft(width=8192);
        AMultiply lp_real(coeffs=LPFILT_R);
        AMultiply lp_imag(coeffs=LPFILT_I);
        AMultiply dc_real(coeffs=VFILT_R);
        AMultiply dc_imag(coeffs=VFILT_I);
        IFFT ifft(width=8192);
        CurveArea sum;
        Sink drop;

        genwave -> convert -> pad -> fft;
        fft.out_real -> lp_real -> dc_real
                -> ifft.in_real;
        fft.out_imag -> lp_imag -> dc_imag
                -> ifft.in_imag;
        ifft -> sum ->drop;
};
```

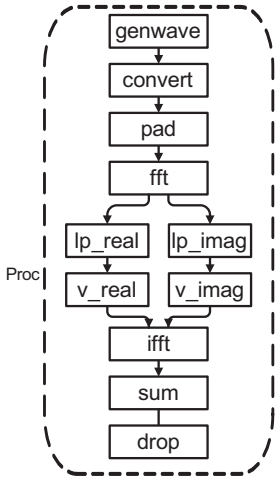
Figure 4.6: Signal cleaner algorithm flow and X code
U8 and F64 refer to the X types UNSIGNED8 and FLOAT64, respectively.

In this algorithm, signals are generated by `TestSignal`, a block that approximates the input from an analog-to-digital converter. `TestSignal` creates samples of log-normal shaped

pulses with random background noise, and outputs one sample per block cycle as an array of 1024 unsigned 8-bit values. This output is sent to `Convert`, which converts the array into 1024, 64-bit floating-point values. `ZeroPad` then pads this array up to 8192 elements by inserting seven zero elements between each original floating-point value. This array is transformed from a time-domain signal into a frequency domain signal using `FFT`, a block which performs the fast Fourier transform (FFT). `FFT` outputs two half-sized signals, the real and the imaginary part of the frequency domain. Four `AMultiply` blocks perform two element-wise array multiplies on each of the two frequency domain arrays, to do a low-pass filter and a deconvolution. The cutoff point of the low-pass filter is chosen to correspond to the high frequencies introduced by the zero-padding operation; the combination of zero-padding and this particular filter performs a sinc interpolation, a high-quality upsampling interpolation technique. The deconvolution filter corrects for errors introduced by the transduction of the electrical signal from its physical source. The convolved frequency-domain halves are converted back into a time-domain signal by the inverse FFT block, `IFFT`. Finally the total charge of the pulse is found with the `CurveArea` block, that takes the sum of the elements in the array. The result is then dropped in the `Sink` block. In a real application implementing the signal cleaner, the edge into the `Sink` block would connect to further blocks that make use of this value (e.g., Section 4.3).

Four mappings of the signal cleaner onto a networked set of processors have been chosen here for analysis. The first mapping, depicted pictorally and with the corresponding X Language code in Figure 4.7, places the entire algorithm onto a single processor.

The second and third mappings, depicted in Figure 4.8 and Figure 4.9, partition the algorithm onto two processors. Mapping 2 places the `AMultiply` filter blocks in the first processor, while mapping 3 places them in the second processor. The fourth mapping, similarly shown in Figure 4.10, maps `genwave` and `convert` to the first processor, `pad` through the filter blocks to the second processor, and the remaining blocks to the third.

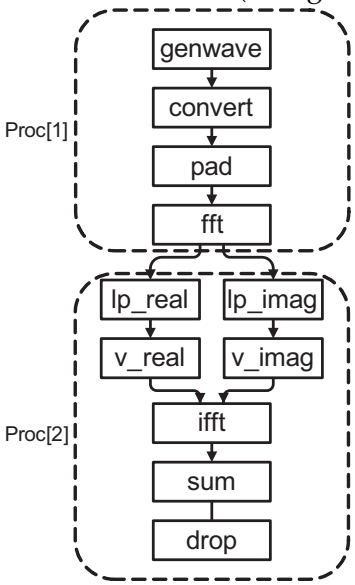



```
resource proc is C_x86;

use SigClean t; // t is the instance name for
                // compound block SigClean

map { t } to proc;
```

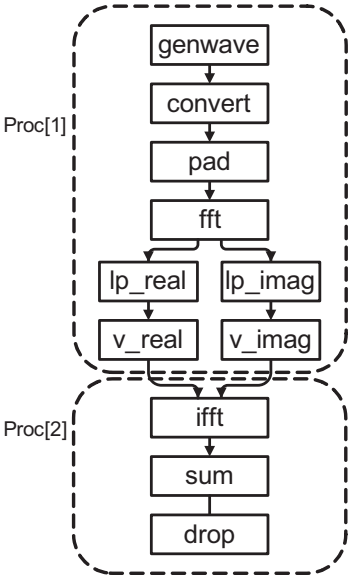
Figure 4.7: Mapping 1
(Using instance names rather than block type names)




```
resource proc[2] is C_x86;
resource net is TCP (
   { proc[1], proc[2] }
);

use SigClean t;
map { t } to net;
map { t.genwave, t.convert,
          t.pad, t.fft }
   to proc[1];
map { t.lp_real, t.lp_imag,
          t.dc_real, t.dc_imag,
          t.ifft, t.sum, t.drop }
   to proc[2];
```

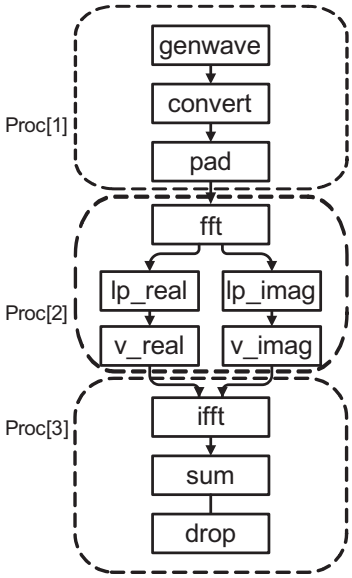
Figure 4.8: Mapping 2




```
resource proc[2] is C_x86;
resource net is TCP (
   { proc[1], proc[2] }
);

use SigClean t;
map { t } to net;
map { t.genwave, t.convert,
          t.pad, t.fft,
          t.lp_real, t.lp_imag,
          t.dc_real, t.dc_imag }
   to proc[1];
map { t.ifft, t.sum, t.drop }
   to proc[2];
```

Figure 4.9: Mapping 3




```
resource proc[3] is C_x86;
resource net is TCP (
   { proc[1], proc[2], proc[3] }
);

use SigClean t;
map { t } to net;
map { t.genwave, t.convert }
   to proc[1];
map { t.fft, t.lp_real,
          t.lp_imag, t.dc_real,
          t.dc_imag }
   to proc[2];
map { t.ifft, t.sum, t.drop }
   to proc[3];
```

Figure 4.10: Mapping 4

### 4.2.2 Performance

The four mappings (figures 4.7 through 4.10) of the Figure 4.6 algorithm were compiled using X-Com. The resulting executables were then evaluated on a set of one, two, or three machines connected with a Gigabit Ethernet switch. The machines each contain an AMD Athlon 64 X2 4400+ processor (dual core, although only one core of each was used) with 1MB L2 cache, 2GB of system memory, and a Broadcom Corp. BCM5271 PCI-Express Gigabit Ethernet controller. All machines were running Red Hat Enterprise Linux AS 4, with Linux kernel 2.6.9-22 for x86-64 architecture.

Figure 4.11 and Table 4.1 each show the result of running Signal Cleaner on a single machine. The segments of the bar in the figure shows the aggregate time spent executing each block. All times are normalized to show the time to process 100,000 input waveforms.

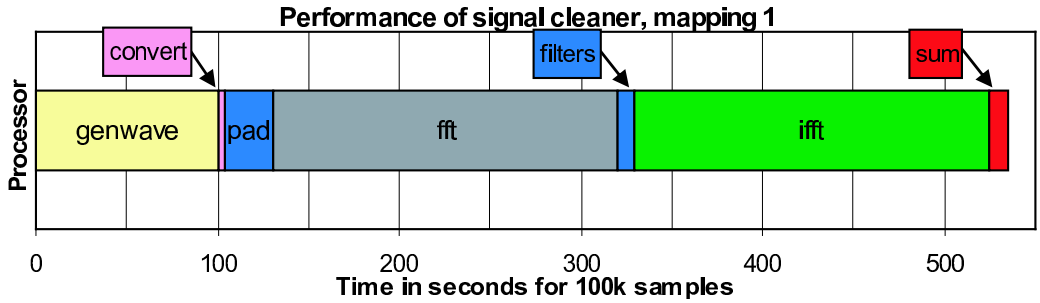



Figure 4.11: Results of mapping 1

Table 4.1: Results of mapping 1

| Block name | Cumulative time |
|---|---|
| genwave | 100.8 seconds |
| convert | 3.16 |
| pad | 27.0 |
| fft | 189.4 |
| (filters) | 8.96 |
| ifft | 194.8 |
| sum | 10.9 |
| drop | 0.044 |
| Overhead | 0.037 |
| Total | 535.1 |

97.7% of the total time of 535.1 seconds were spent in five of the blocks: `genwave` (with 100.75 aggregate seconds per 100,000 samples), `pad` (27.0), `fft` (189.4), `ifft` (194.8), and `sum` (10.9).

0.036 aggregate seconds (less than .01%) were spent in unaccounted "overhead" time, measured from the beginning of the first block `go()` call to the last `push()`. This negligible overhead time corresponds to the overhead of the X round-robin scheduler and inter-block communication manager in this application.

One can use the results of this first mapping to attempt to create an efficient partitioning of the algorithm onto two equal processors. Ignoring the communication overhead, this can be done analytically by finding the partitioning that comes closest to evenly dividing the processing times. To minimize the effects of communication overhead, even though the value is not known, such a partitioning is done under the additional constraint of making only one "cut" of the pipeline. This is the technique performed by pipeline optimizations such as those found in [7] (which can also improve the partition if communication time *is* known and employ more complex performance models). In this example, this is trivial to do manually, and the result is the second mapping. Ideally, and with no communication overhead, the execution time would be the larger of the sums of the Table 4.1 execution times for blocks on each processor: 320.4 seconds for the first stage and 214.7 seconds for the second, thus 320.4 seconds for the whole application.

Figure 4.12 shows the result of running Signal Cleaner on two machines, using the organization shown in Figure 4.8. Again, times shown are aggregate over the total execution and normalized to 100,000 samples.

78.7% of the total distributed time of 677.8 processor-seconds were spent in the same five blocks as before: `genwave` (with 103.7 aggregate seconds per 100,000 samples), `pad` (28.9), `fft` (190.2), `ifft` (201.0), and `sum` (9.6). The total runtime of 338.9 seconds represents an improvement of 57.9% over the time of 535.1 seconds in mapping 1. It is 5.8% slower than the ideal execution time of 320.4 seconds calculated earlier. This is due to the increase

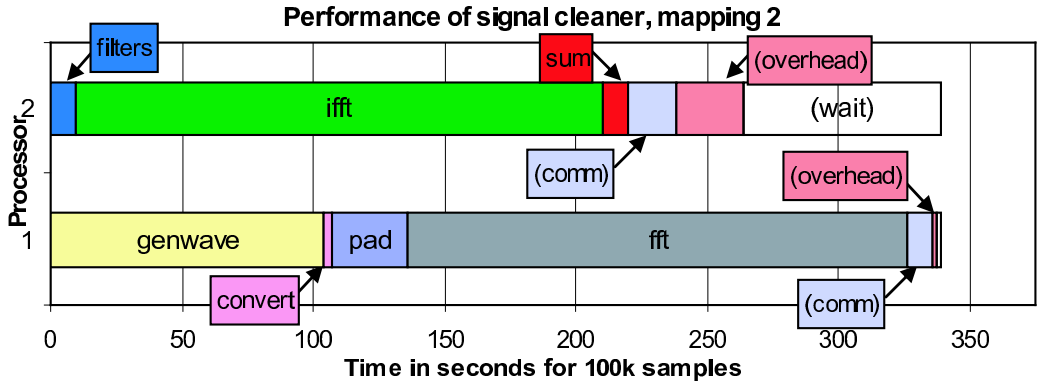



Figure 4.12: Results of mapping 2

in time spent performing communications between the processors, additional memory management overhead, and other effects of partitioning the algorithm.

Overhead time, including X scheduling and inter-block overhead as before, was measured at 0.006% of the runtime in processor 1, and 7.7% in processor 2. This overhead time also contains a portion of the time spent busy-waiting on communications, thus it may be somewhat inflated in these figures.

Communication time, the time spent transmitting data over TCP, was measured at about 2.9% of the runtime in processor 1, and 5.3% of the runtime in processor 2.

The remaining time (about 22.1%) found in processor 2 is wait time. This is wasted execution time, created by an imperfectly balanced pipeline where the rate of production in the first process is slower than the rate of consumption in the second. Processor 2 is able to complete the processing of its data quickly, so it spends a significant amount of time waiting for data to arrive.

This demonstrates an important aspect of pipelining: the total system performance of a pipeline (here, the two-parallel-stage pipeline formed by the two processors) is limited to the time spent by the longest stage.

Figure 4.13 shows the result of running Signal Cleaner on two machines, using the organization shown in Figure 4.9. This organization is similar to the previous mapping, except the filter blocks (`AMultiply`) have been mapped to processor 1 instead of processor 2.

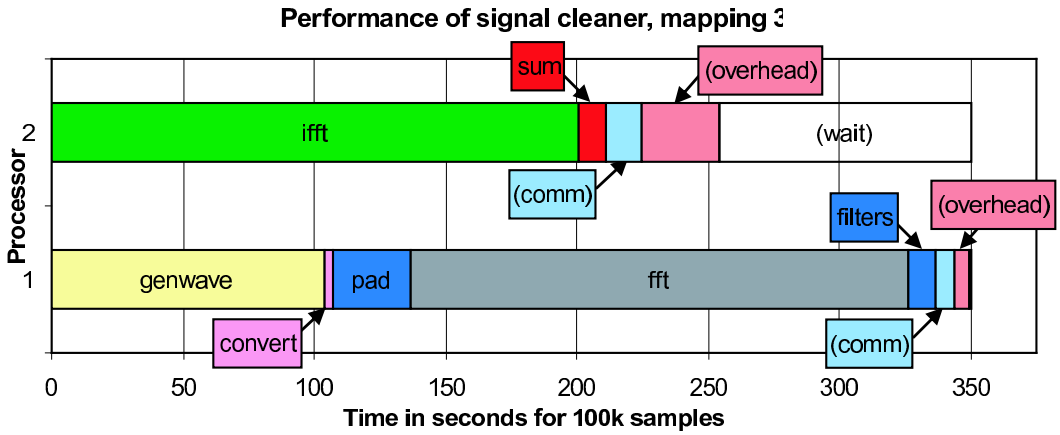



Figure 4.13: Results of mapping 3

In this mapping, the five larger blocks take up 76.1% of the total distributed time of 701.0 seconds. Each of these larger blocks executed in within 0.5% of the time recorded in the previous mapping. The total runtime of 350.5 seconds represents an improvement of 52.7% over the time of 535.1 seconds in mapping 1.

The time spent outside blocks follows a pattern similar to the previous mapping. Overhead time here was measured at 1.5% in processor 1 and 8.4% in processor 2. Communication time was measured at 2.1% in processor 1 and 4.0% in processor 2.

Unaccounted time was measured at 0.6% in processor 1, and 27.4% in processor 2. In this case, moving the filters from processor 2 to processor 1 caused the imbalance between the two stages to grow. The filters account for 9.7 seconds in mapping 2 and 10.0 seconds in mapping 3. The total time has increased by 11.6 seconds going to mapping 3, which suggests that the relative slow-down in performance is directly and primarily caused by the approximately 10 seconds spent in the added filter blocks. This clearly demonstrates the system-wide performance effect of the slowest stage discussed in the previous mapping.

Figure 4.14 shows the result of running Signal Cleaner on three machines, using the organization shown in Figure 4.10.

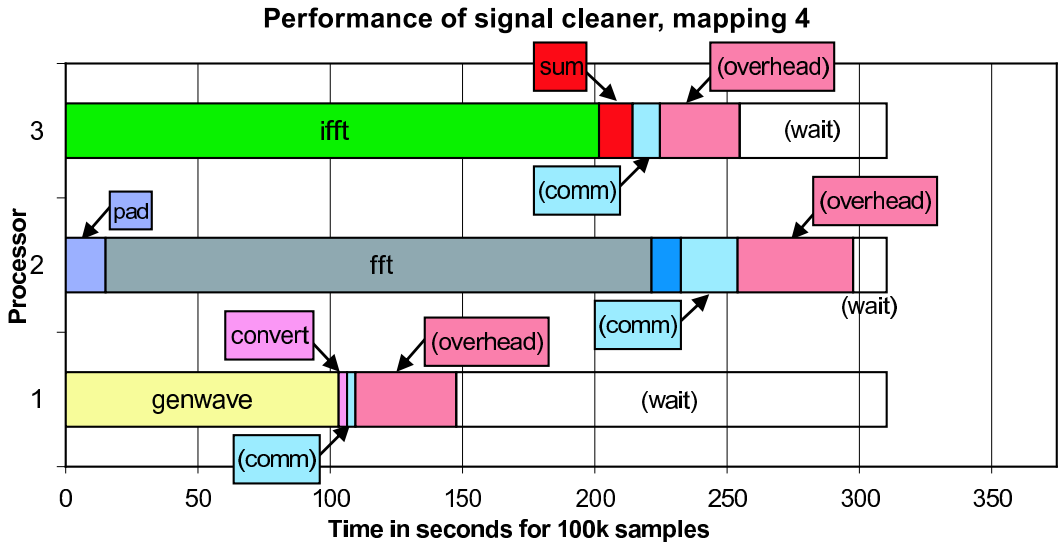



Figure 4.14: Results of mapping 4

This mapping increases the available process-level parallelism by introducing a third processor to the pipeline. The improvement on application performance is apparent: runtime for 100,000 samples is reduced to 310.3 total seconds, compared to 338.9 seconds in mapping 2, the next-best arragement. This is an improvement of 57.9% over mapping 1, and 9.2% over mapping 2.

The block-accounted portion of Figure 4.14 indicates that ideally, even better performance could be seen with optimal data availability — as low as 230 seconds (a 133% improvement over mapping 1, or 47.3% over mapping 2).

The significant increase in unaccounted (overhead) time is most likely due to increase in wasted time spent blocking on transmitting data between the processors. This effect is compounded in this mapping because, for instance, processor 2 waiting to send to processor 3 will stall the receiving of data from processor 1, and vice versa. Much of this overhead can be overcome with the threading of inter-resource communications, or the

increased use of nonblocking I/O with queuing on the outputs. Future implementations of TCP (and other linktypes) will improve the performance in these conditions.

## 4.3 Gamma Ray Event Parametrization

The third example application is from the field of astrophysics, and demonstrates the effectiveness of the X Language in describing large algorithmic structures in real scientific applications. This application, called gamma ray event parameterization, involves the processing of large amounts of sensor data with the goal of identifying and characterizing interstellar cosmic rays, particularly high-energy gamma rays.

Gamma rays are emitted from a wide variety of interesting extraterrestrial sources. Currently, astrophysicists believe that common gamma ray sources may include:

- Pulsars (very quickly spinning neutron stars)
- Expanding gas surrounding supernova explosions
- Supermassive black holes in galactic nuclei
- Neutron star collisions (uncertain)
- Hypernovae, very large supernovae resulting in black holes (uncertain)

The first three phenomona are relatively periodic sources of gamma rays, while the last two are potential candidate sources of gamma ray bursts (GRBs). This distinction is important, because it represents two separate scientific applications which use the same equipment and processing techniques. Ground-based gamma ray telescopes generally examine the periodic sources during their nightly observations. GRBs, however, are very rare events that are seldom detectable with any advanced warning. In order to analyze GRBs, a "rare event search" is performed over large spans of the database of nightly observations. While

the first application can be performed online (i.e., during the observation itself) if the application throughput can keep up with the sensor rate, the second application must be performed over potentially very large storage-based databases.

A handful of ground-based (e.g., HESS [13], VERITAS [32], MAGIC [20]) and space-based gamma ray observatories (e.g., GLAST, not yet deployed) have been developed in recent years following the success of past gamma ray observatories such as the Fred Lawrence Whipple Observatory and EGRET [17]. The ground-based systems all follow similar detection techniques, employing relatively large (10 to 16 meter) parabolically-arranged arrays of smaller mirrors focused on arrays of hundreds of photomultiplier tubes (PMTs); many employ multiple (usually four) telescopes of this configuration to perform stereoscopic observation.

In these ground-based experiments, the high-energy gamma rays are not directly measured. Rather, the lower-frequency photons produced by the gamma rays' interaction with atmospheric particles are measured using what is called the Imaging Atmospheric Cherenkov Technique. With this technique, high-energy gamma rays interact with the Earth's upper atmosphere, forming electron-positron pairs which subsequently generate showers of electrons moving at velocities near the absolute speed of light. As they fall to earth, the fast-moving particles are slowed to the speed of light in the atmosphere, resulting in electromagnetic shockwaves called Cherenkov radiation. These shockwaves appear as faint glowing streaks of light in the blue through near ultraviolet portion of the spectrum. In an experiment, the streaks of light reflect off the mirror arrays and strike the PMTs. The PMTs then convert this light, which may be on the order of single photons and last only 3–10 ns, into electrical pulses which are then be converted into digital waveforms by a high-frequency "flash" analog-to-digital converter (FADC).

The measured Cherenkov radiation can be on the order of 100 total incident photons from one gamma ray, and this is recorded against a strong background of diffuse light from starlight and city light. In the VERITAS telescopes, for example, when the ADC samples are only 2ns long, the individual measured photoelectrons register around 2–4 digital

counts in the FADC, against background light of 15–20 digital counts and a variance of 5–10. As a result, a significant amount of signal processing in each channel and image processing in each telescope is required to "clean" the recorded waveforms and discriminate the Cherenkov pulses from background light fluctuations.

The processing of waveform data into telescope event characteristics is performed in two main stages, per-channel (PMT) signal processing and per-event image processing. Signal processing needs to perform the following steps on *each* channel:

1. Up-sample the waveform, using sinc ($\sin(x)^{-1}$) interpolation.
2. Deconvolve the waveform using a matched filter, which corrects for the PMT capacitance and analog transmission defects caused by transducing the photons and transmitting the signal to the FADC.
3. Window the waveform to select only the expected location of the charge pulse on that channel.
4. Subtract the "pedestal," or baseline noise value from the waveform.
5. Scale the waveform, as necessary, to correct for differences in the gain between multiple PMTs.
6. Find the area under the curve of the cleaned, scaled waveform; this is the charge incurred by the Cherenkov photon, if any.
7. Threshold the charge, to filter out charges within some factor of the typical variation in background light.

Note that steps 1, 2, and 6 comprise the signal cleaning application presented in Section 4.2.

Once this has been performed on every channel, a clean image of significant charges detected in the event will be created. Then, image processing must be performed:

1. Calculate the moments $M[x], M[x^2], M[y], M[y^2], M[xy]$ over the charges in the image, using the $(x, y)$ coordinates of each channel's corresponding PMT in the array.

2. Compute the Hillas parameters using various algebraic combinations of the five moments above. The Hillas parameters are a set of values describing the image cast onto the telescope by cosmic particles. For instance, a gamma ray typically looks like an elongated ellipse, and Hillas parameters will characterize the ellipse's width, height, distance from the center of the array, and angle relative to the center of the array.

3. Store the Hillas parameters for each event for use in the final step described below.

The procedure to process the digital waveforms, with an emphasis on an optimal matched filter for the VERITAS electronics, is discussed further in [9].

The final step in scientific analysis of the experiment is *cutting*, a less computationally-intensive process that filters images by their parameters and generates statistics of the filtered events. This step does not require sophisticated computational resources, and the filtering parameters are likely to change quite drastically for each scientific objective. For this reason it is left out of the X Language application design.

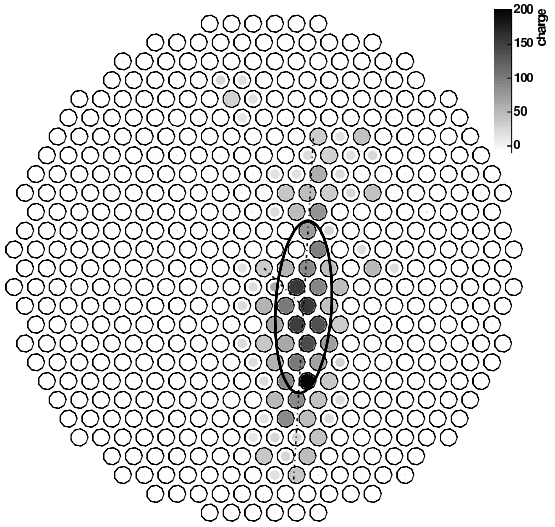



Figure 4.15: VERITAS PMT array with sample event superimposed

Figure 4.15 depicts a schematic of the PMT array found in each telescope of the VERITAS system. Each VERITAS telescope contains an array of 499 PMTs arranged in a packed-circle pattern as seen in the diagram. Superimposed over the schematic is a simulation of a single gamma-ray event, with the total charge per pixel indicated by the shade of the filled circle within each PMT. Image processing of the event results in a set of parameters, particularly those that describe the hollow ellipse that rougly contains the pixels of high charge.

The above steps describe the general processing steps required for the Imaging Atmospheric Cherenkov Technique. A processing pipeline designed for a specific telescope using this technique must be parameterized to account for the particular set of physical sensors used to capture the data, such as the particular packed-circle pattern above. The VERITAS events are digitized only after passing multiple levels of triggering (per-channel, per-telescope, and across telescopes), and the data for each channel in an event are available at the same time. VERITAS consists of four telescopes (currently two are operational), however the stereoscopic analysis is well-suited for the cutting step, performed after the computationally-intensive portion of the analysis.

The following section will examine the particular approach used in designing an X application to perform gamma ray event parameterization for the VERITAS telescopes. X blocks performing many of the above steps are connected in a pipeline suitable to deployment on pipelined processing architectures. Following the application description is a set of performance results found by different mappings of the algorithm within a multi-core, multi-processor system.

### 4.3.1 Design

Figure 4.16 depicts the design of the VERITAS gamma ray event parameterization algorithm developed in the X Language. Two of the 499 channel processing pipelines are shown, one of which is annotated with the data types transmitted on the edges.

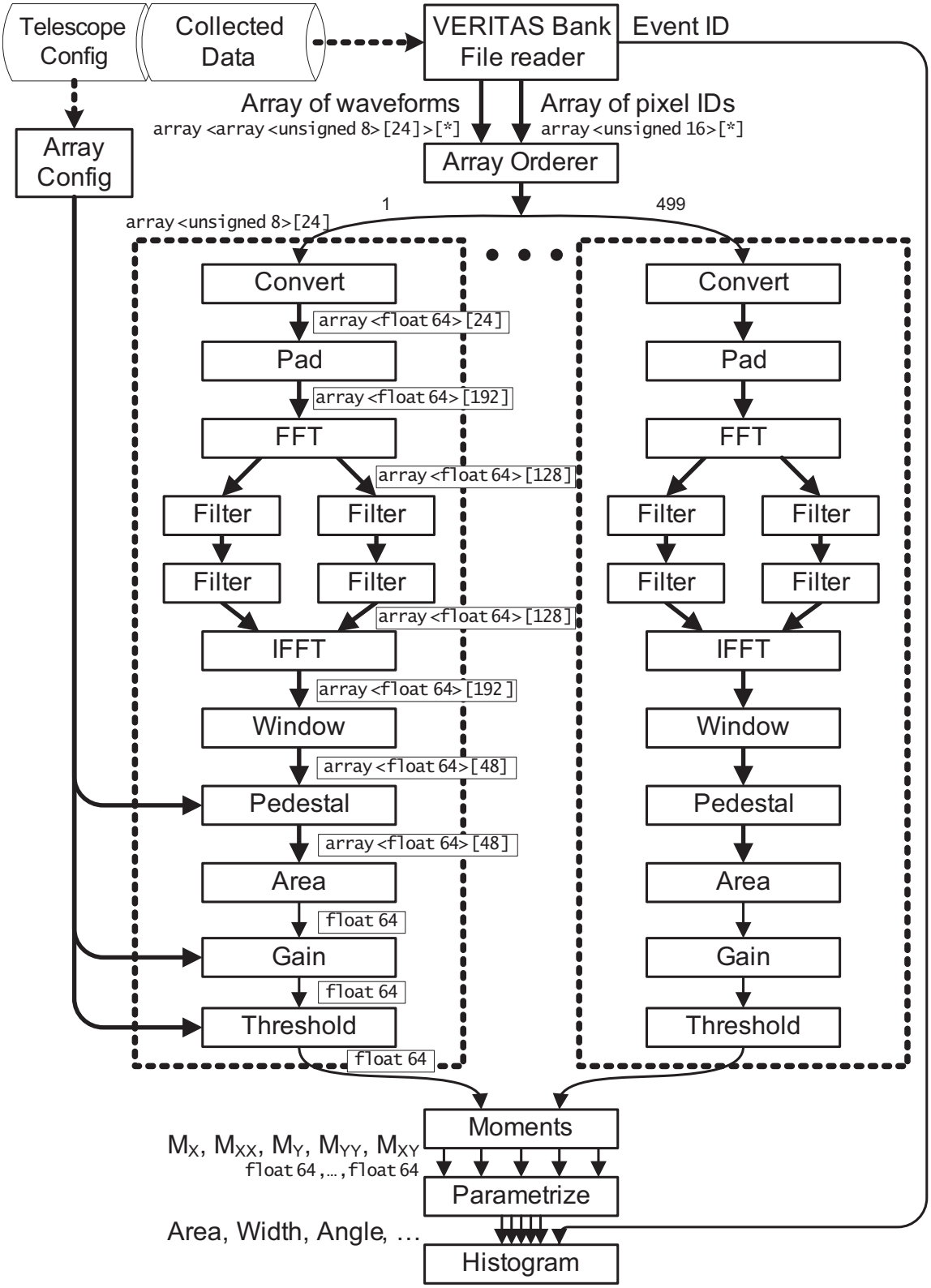



Figure 4.16: VERITAS gamma ray signal processing pipeline in X

One input to the application is a telescope configuration file, containing the pedestal value, gain coefficient, and charge threshold for each PMT in the telescope for that run. The "Array Config" block in Figure 4.16 distributes these values to the three corresponding blocks in each of the 499 signal processing pipelines. The $(x, y)$ coordinates of the PMTs do not change between telescopes, so they are instead provided as compile-time configuration values to the moment generator.

The second input to the application is the data file, in VERITAS Bank File format, containing waveforms collected during one "run" of the experiment. A typical 28-minute run will yield approximately 1.4 GB of data, varying with the triggering rate. The "VERITAS Bank File reader" block wraps around a C++ event reader which uses the VERITAS-provided Bank File library. It reads each event from the file (uncompressing if necessary) and creates two variable arrays, one containing the waveforms themselves and the second containing the channel numbers for each waveform.

The two variable arrays are read by the "Array Orderer" block and distributed among the 499 signal cleaning pipelines (contained in the dashed lines in Figure 4.16).

The pipeline contains the signal processing steps described at the beginning of this section. To perform interpolated upsampling and deconvolution, the signal is first padded and then converted into the frequency domain. Zero-padding followed by a perfect low-pass filter in the frequency domain directly corresponds to the sinc-interpolation operation. The second filter in the frequency domain performs the matched filter. It is convenient to do this in the frequency domain as the complicated matched filter operation can be quite complex as a FIR filter in the time domain. Note that the two filter operations are performed in two smaller pipelines; this is because the FFT and IFFT operations treat the real and imaginary portions as two distinct arrays for compatibility with other library operations. The remaining time-domain operations directly follow the previously described outline of steps: the waveform is windowed around the pulse, the pedestal is subtracted, the curve area (charge) is computed, and the resulting charge is multiplied by the PMT gain.

The output of each channel's signal processing pipeline is a cleaned charge value, corresponding to the charge from high-energy photons in the event. Any below-threshold charges (less than typical variance) are zeroed to further clean the event image. The first, second, and correlated moments of all these charges are then computed, and Hillas parameters are found using their values. Finally, the Hillas parameters for each array event are received by the "Histogram" block, which bins each parameter in each event for scientific analysis of the whole run.

Four different mappings of the X blocks onto one to four processors were created to demonstrate the performance of different partitioning techniques. Each mapping is described below and is depicted along with a simplified version of the gamma ray event parameterization algorithm (Figure 4.16).

- The trivial mapping, mapping 1 (Figure 4.17), simply places all blocks on one processor.
- Mapping 2 (Figure 4.18) places the front blocks (reader, config, order) and one half of the signal processing pipelines on one processor, and the remaining blocks on another.
- Mapping 3 (Figure 4.19) places the front blocks and the first five stages of each of the pixel pipelines (convert, zeropad, fft, two filters) on one processor, and the remaining blocks on another.
- Mapping 4 (Figure 4.20) is similar to mapping 2, but across four processor resources. It places the front blocks and 90 of the signal processing pipelines on one resource, the back blocks and 90 pipelines on the second, 160 pipelines on the third, and 160 pipelines on the fourth. These proportions were chosen (by hand) to roughly provide equal amounts of computational work to each processor.

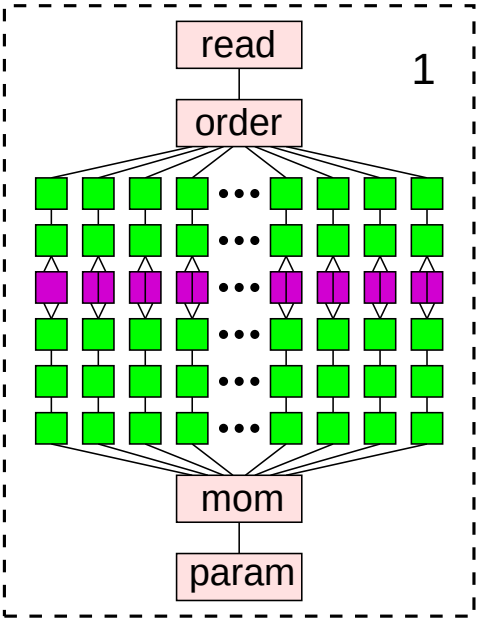



Figure 4.17: Mapping 1

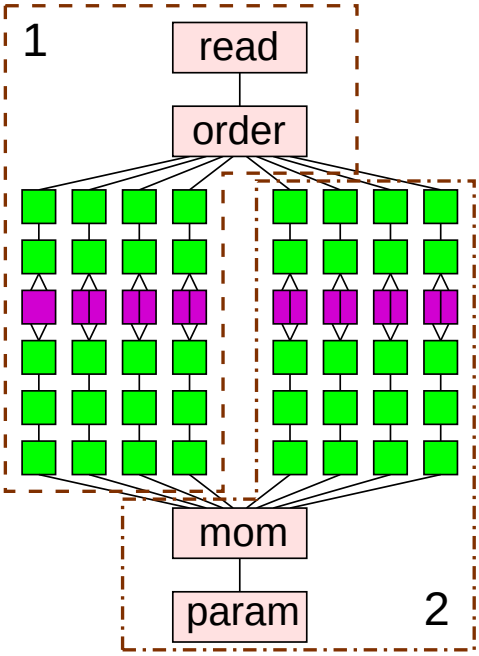



Figure 4.18: Mapping 2

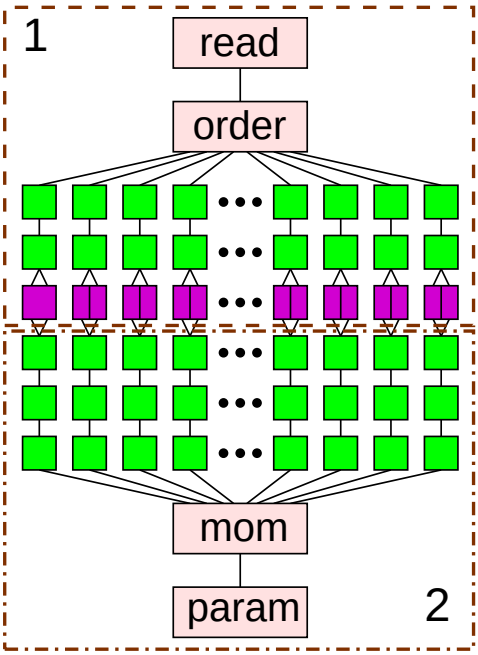



Figure 4.19: Mapping 3

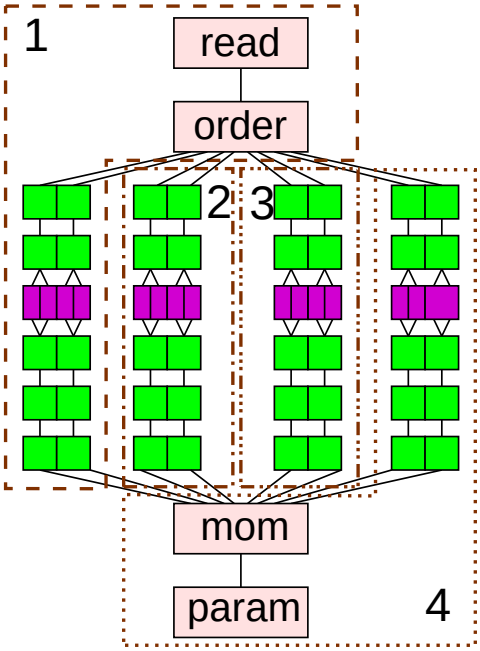



Figure 4.20: Mapping 4

### 4.3.2 Performance

The mappings found in the previous section section were all evaluated by executing the X-Com generated programs on a single machine. This machine contains 8GB of memory distributed to two AMD Opteron 270 processors, which are *dual-core* devices and thus total four cores, operating at approximately 2000 MHz, with 1MB of L2 cache per core. The machine was running Red Hat Enterprise Linux AS 4, with Linux kernel 2.6.9-34 for x86-64 architecture. As before, the processor mapping was maintained by use of the `sched_setaffinity` Linux system call to restrict each program to scheduling on only one of the four logical processors.

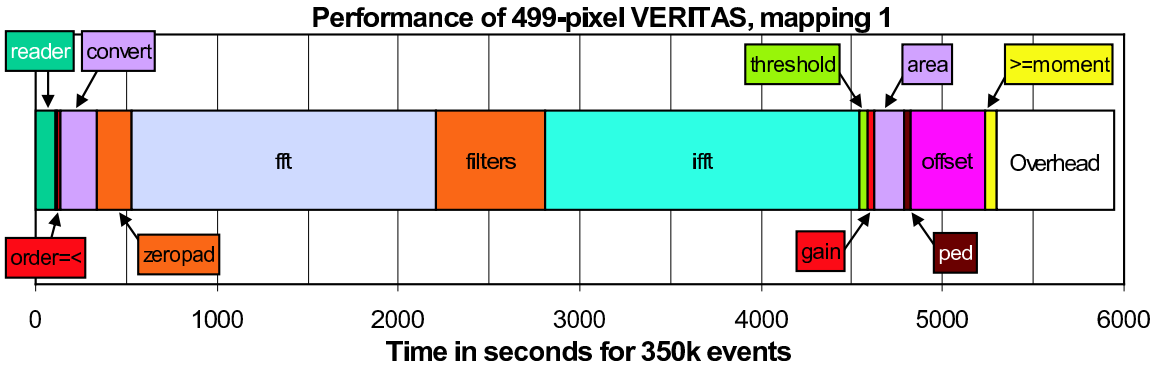



Figure 4.21: Results of mapping 1

Figure 4.21 presents the results of running the gamma ray event parameterization application with the trivial, single-context mapping. Processing 350000 events takes a total of 5945 seconds. The FFT and IFFT blocks are clearly the most computationally-intensive operations in the application, taking up 3420 cumulative seconds, or 57.5% of the total runtime. The filters (four elementwise array multiplications per channel) also take a significant portion of the runtime, with 593 seconds (10.0%). Overhead for this mapping is 642 cumulative seconds (10.7%), corresponding to the time not spent in any individual processing block.

Figure 4.22 presents the results of running the same application and dataset of 350000 events on mapping 2. The total runtime was 4776 seconds, 80.3% of the time to run the

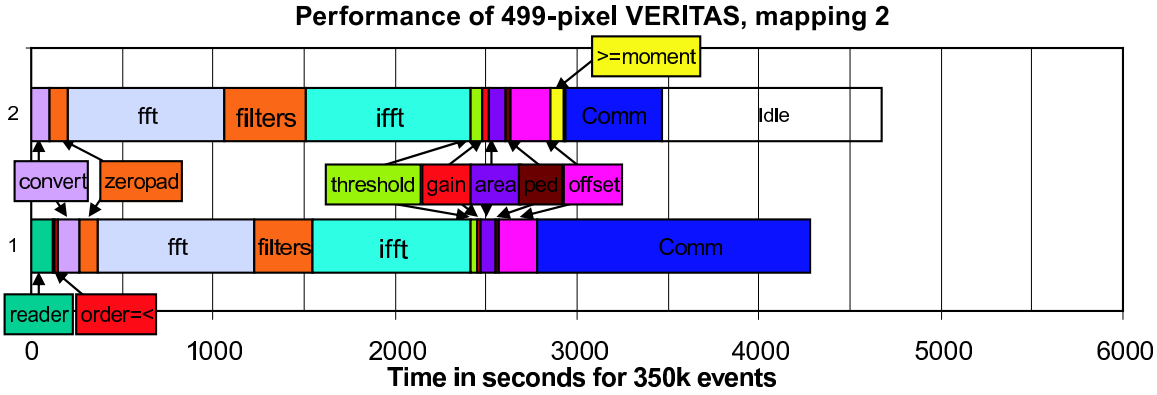



Figure 4.22: Results of mapping 2

trivially mapped version. While this mapping was developed to almost equally partition the application across two processors, it falls short of the ideal of 50% of the execution time of the first mapping. This is due to a number of factors, such as the increase in individual block processing time, the imbalance of the pipeline due to these different processing times, and the introduction of a significant communication delay.

Each class of blocks in this mapping takes somewhat longer to execute than in mapping 1, from 2–3% longer for the FFT/IFFT blocks to 110% longer for the threshold block. This is probably due to the increased strain on memory due to multiple simultaneous contexts.

The idle time for this mapping was 1211 cumulative seconds in the second core, or 25.4% of that core's runtime. This indicates that the second core was starved for data, and that in an ideal case, an approximate 12.7% reduction in runtime could be achieved with a better-balanced pipeline.

Communication time is a major component of the runtime of mapping 2, at 1505.5 seconds (31.5%) of the first core's total runtime. The large magnitude of communication time is likely due to inefficiencies of the Unix socket communication code. Better queueing and threading are possible techniques to improve this time, discussed further in Section 5.3. Even with improvements, however, the communication involves the transmission of very many small (8–192 byte) packets, for which the messaging overhead of individual Unix socket packets is not efficient.

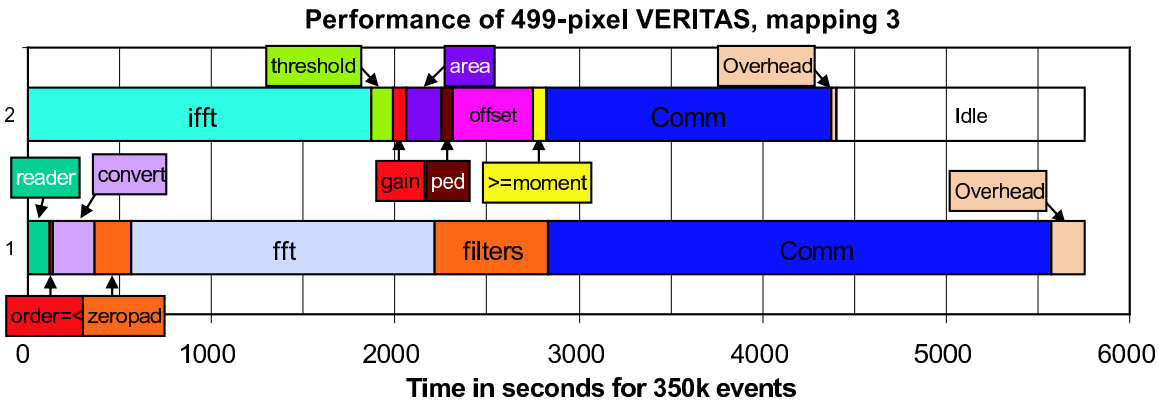



Figure 4.23: Results of mapping 3

Figure 4.23 presents the results of running the same application and dataset on mapping 3. The total runtime was 5755 seconds, 96.8% of the time to run the trivially mapped version.

This mapping is similar to mapping 2, with two processor cores, but with a "horizontal" partition between stages of the algorithm pipeline, rather than a "vertical" partition between equal numbers of nearly identical pipelines.

Most of the individual block classes perform slightly better in this mapping, possibly due to better locality of execution, however the communication time is much greater. The larger of the two communication times was 2741.8 (47.6%) seconds in the first core. The much larger communication time (compared to mapping 2) is probably due to the much greater amount of data transferred over the interconnect. In this mapping, 998 arrays of 128 FLOAT64s (about 1 MB) per event are sent from filters to IFFT in each event, compared to 250 arrays of 24 UNSIGNED8s and 250 FLOAT64s per event (8 kB).

Figure 4.24 presents the results of running the same application and dataset on mapping 4. The total runtime was 3299 seconds, 55.5% of the time to run the trivially mapped version, and 69.1% of the time to run mapping 2.

This mapping demonstrates the scalability of "vertical" algorithm partitionings. The per-channel signal processing pipelines are partitioned in such a manner. In mapping 2, the per-channel processing operations took an aggregate total of 5484 seconds divided among

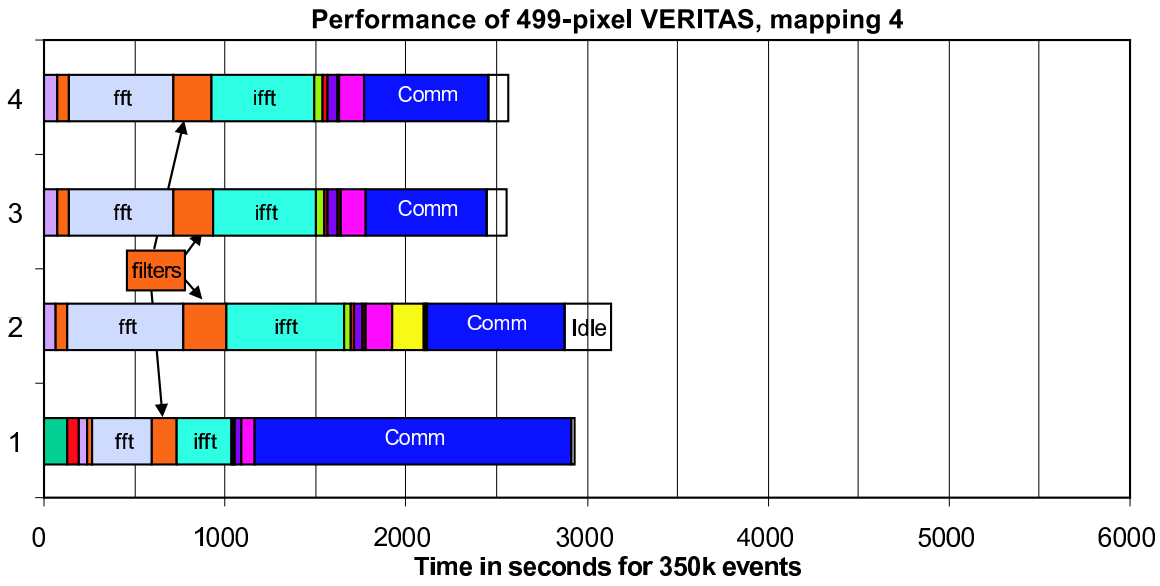



Figure 4.24: Results of mapping 4

two cores, while the same operations in mapping 4 took 6440 seconds (17.4% more) divided among four cores. Ideally, this could yield a 41.2% reduction in runtime over this portion of the algorithm, although a 32.5% reduction was actually seen due to imperfect partitioning in both mappings.

As expected by the increased communication between cores, communication took longer in mapping 4 than mapping 2. The core containing the front blocks took 1743 cumulative seconds (16% more than mapping 2), while the longest other core took 754 seconds (42% more than the second core in mapping 2).

### 4.3.3 Further performance analysis

The above results of mapping 4 demonstrated a speedup of only 1.8 over the single-processor case. At first, this may seem low when compared to a speedup of 4 under ideal scaling. To understand the nonideal scaling of the problem, consider the performance degradation of mapping 4 broken into the following categories:

- *Communication overhead*. The time spent transmitting or receiving data from another processor.
- *Memory contention overhead*. The increased time spent executing each block due to contention for the memory controller between multiple cores in a processor, and for the memory bus between multiple processors in the system. Blocks will vary in their sensitivity to this overhead depending on their utilization of memory.
- *Partition imbalance*. The idle time spent waiting to receive data from a busy upstream process (starvation) or waiting to send data to a busy downstream device (blocking on a filled queue). This also includes time wasted by an upstream device having finished its work before the downstream devices.
- *Removed parallelism*. Some operations that proceed in parallel within a processor, for instance input stream caching from disk, can reveal previously hidden delays if the main computational processing time goes down.
- *Other overhead*. The additional processing time introduced by the generated X code, for instance due to reduced spatial locality.

The accuracy of measuring each of these types of performance degradation varies greatly.

Communication overhead is easily measured by the time it takes to perform the transmit or receive operation (although note that communication affects partition imbalance as well). In mapping 4, the communication overhead accounted for 1743 out of 3299 seconds in the first core, 754 seconds in the second, 666 seconds in the third, and 688 seconds in the fourth. In the 13196 seconds of distributed execution, 3851 seconds (29.5%) were spent performing communication operations.

Memory contention overhead is difficult to directly measure without hardware support. However, the overhead can be estimated by executing four copies of the application in parallel and measuring the relative slowdown. Four copies, each executing one quarter of a 120000-event run, were simultaneously executed on the same four-core system used in mapping 4 (twice, to mitigate disk I/O time). In this configuration, the copies each took

626–628 seconds to execute, with a sum of 2504 seconds. Compared to 2191 seconds to run all events in one process, this is a 14.3% increase in distributed execution time. Assuming that the disk I/O effects have been reduced by caching in the OS, this increase in time is probably mostly due to contention for memory resources between the cores in the system.

The partition imbalance is more difficult to properly measure without a detailed trace of the data production and consumption times [3]. The measured "idle" time can be used, however, as an effective first-order estimate of the wasted time introduced by an imbalanced partition. In mapping 4, a total of 493 seconds, distributed over the 4 cores, were measured as time spent idle. Additionally, about 1150 seconds of distributed time was spent by core 1, 3, and 4, waiting for core 2 to complete its operations. By perfectly redistributing the operations and assuming no variation in block execution time, these 1650 seconds of wasted time (12.5% of the total) could be entirely removed.

Table 4.2 summarizes the three components discussed above. The sum of these is 7388 seconds, or 56.0% of the total execution time. From this number alone, one estimates a speedup over mapping 1 (which ran in 5945 seconds) of

$$\frac{5945\text{ s}}{\frac{1}{4\text{ processors}}\frac{5945\text{ s}}{1-.56}} = 1.76$$

This comes close to the realized speedup of 1.8.

It is important to note that using the sum of the components is only valid for a rough estimate, as the delays may be strongly interrelated, creating both constructive and destructive interference between their overheads. Additionally, this estimate ignores the final two overheads discussed above, which are more difficult to measure, and are assumed to affect the system performance to a lesser extent.

[3] This is one advantage of using the X-Sim tools, under development

Table 4.2: Estimated contributions of non-ideal scaling effects in mapping 4

| Category | Distributed time | Percent of total |
|---|---|---|
| Communication | 3851 | 29.5% |
| Memory contention | 1887* | 14.3% |
| Partition imbalance | 1650 | 12.5% |
| Sum** | 7388 | 56.0% |

*: Extrapolated from separate simulation. **: See text for important note.

# Chapter 5

# Summary

This thesis has presented the overall design of the Auto-Pipe toolset, the X Language, and X-Com, an X Language compiler. This final chapter summarizes the conclusions drawn from this work, the notable contributions that have been made, and presents an outline of future opportunities for research and development with Auto-Pipe and the X Language.

## 5.1 Conclusions

Developing parallel, performance-sensitive applications distributed across complex, possibly heterogeneous sets of computation resources is a difficult task. This requires a great deal of time and effort during the initial programming, the debugging, and the optimization stages of development.

Auto-Pipe is a toolset that reduces the effort required in these stages while improving the developer's understanding of the system performance. An important part of Auto-Pipe is the X Language, in which all applications are described at a high, structural level. The X Language compiler, X-Com, has been written to perform the core functionality of the Auto-Pipe system. The remaining simulator (X-Sim), optimizer (X-Opt), and deployment (X-Dep) components are currently being developed.

Even though the full Auto-Pipe system is not yet complete, X-Com shows promise as an easy-to-use and effective means of trying many different partitionings of an application

onto different sets of computational resources, connected with different types of interconnects. It also provides a useful infrastructure for combining different forms of execution such as native execution and hardware simulation; this aspect will be further enhanced when automated with X-Sim.

## 5.2 Contributions and Implementation Status

Section 1.3 introduced the design of Auto-Pipe, a flexible set of design tools that improve the development of heterogeneous, parallel applications. A set of architypical flow diagrams were introduced (Figures 1.8, 1.9, 1.10) which illustrate the flexibility of X-Com and X-Sim to enhance the development process from initial testing through completion. In the future the system will employ the power of an optimizer to improve the performance of the application. While the initial goals are for X-Opt to optimize the performance of pipelines through currently available analytic tools, the design framework, using the combination of X-Sim and X-Opt, permits a variety of optimization metrics, limited only by the supported set of simulators and models. Section 5.3 examines some possible future directions for this capability.

The contributions represented in this thesis include:

- An initial design (Section 1.3) of the structure and organization of the Auto-Pipe tool set.
- The X Language (Chapter 2), a hybrid dataflow coordination language with support for complex heterogeneous systems of computation and interconnect resources.
- X-Com (Chapter 3), a compiler that supports the X Language specification and currently performs code generation for C and HDL targets.
- Three applications (Chapter 4) which have been developed in the X Language. They demonstrate the effectiveness of the Auto-Pipe development flows to analyze the performance of a distributed application and improve its performance.

At the time of the publication of this thesis, the X-Com compiler core (excluding code generations) supports nearly all of the X Language specification presented in Chapter 2. The only omission is handling of the `STRUCT` composite data type. The C and VHDL generators support most of the remaining language structures. Missing from both C and VHDL generator implementations are proper handling of the `VARRAY` data type. The VHDL generator does not permit `ARRAY` data types to be provided as configuration. Irregularities exist in the handling of edge sets which are merged and then immediately split again; a temporary work-around is currently in use where a pass-through block is placed between the outputs and inputs, and the generated code is indistinguishible from a working implementation.

The binding operation is not yet automated. Eventually, a tool will perform the binding of code generated for resources by X-Com to locally available devices. Currently, the binding is performed by hand through the design of a Makefile script for each application (generally a one-time operation per resource allocation).

X-Sim is under development. Interconnect generators for the interface between simulators are being developed for X-Sim. The simulator bindings and simulator automation steps have not yet been designed. X-Sim is planned to be completed in the early summer of 2006.

X-Dep is under development. Currently, a manually created deployment Makefile exists, however work still needs to be done to automatically generate the deployment scripts for simulation and real device execution. X-Dep will be developed and completed during the summer of 2006.

X-Opt has only been designed at a conceptual level, and the nontrivial optimization algorithms have yet to be designed.

Currently, the available code generators in X-Com generate general-purpose C and VHDL coordination code. The C generator works under POSIX-compliant systems, and has been tested under 32-bit and 64-bit Linux 2.6, and 32-bit and 64-bit Cygwin. The HDL generator has been written structurally, but has not yet been tested under synthesis to a real device.

Interconnects are available for TCP and Unix sockets between C resources, but not interconnects are currently available for HDL resources (a rudimentary file I/O interconnect does currently exist for testing). Both TCP and Unix sockets use blocking I/O operations to send, and thus create a noticeable overhead when block processing times are small.

## 5.3 Future Work

While X-Com is nearly complete and the Auto-Pipe tools have been well defined, there is still work to be done in implementing the remaining parts, improving the performance of the generated code, and developing more targets for computation and interconnect resources and simulators. Many of the improvements will draw from current design techniques and recent research, but even yet-unfinished research may have a great impact on the Auto-Pipe tools and X Language design. This section outlines potential future work that may be done in the Auto-Pipe system, both planned and hypothetical.

The usability of the X Language should be analyzed and improved:

- Productivity could be recorded in a properly designed experiment, consisting of a reasonable sample size and control group of programmers developing applications using the X Language. Performance (time to develop, time to debug, resulting application performance) could then be compared between the two groups to quantitatively measure the effectiveness of the X Language compared to traditional methodologies.
- In general, expansion of the X-Com user base will result in more feedback, thus improving the "user-friendliness" of the compiler and X Language and help to remove any possible bugs.

The current C code generator, which is expected to be the most often used generator, may benefit from improvements to its API performance and scheduling techniques:

- The memory allocation calls need to be individually profiled in a variety of cases; currently, the standard `malloc()` and `free()` calls are used, however a special-purpose low-overhead allocater for small data elements might provide better performance.
- More scheduling models should be supported. Currently, the `go()` functions are scheduled in a round-robin manner, and all `push()` functions are called through direct function calls.
  - Dynamic and profile-based static scheduling of block calls could result in improved performance in certain applications.
  - Scheduling techniques optimized for pipelines could also improve performance in certain applications. An example of such techniques applied to heterogeneous systems may be found in [22].
- The performance of various schedules for I/O on external edges (those edges that connect across interconnect resources) should be examined. Currently, there may be a significant amount of time spent performing communications between resources that could be hidden by better software pipelining of inter-resource I/O.

Further generation targets may be developed for use with X-Com (and thus X-Sim):

- Computation resources, such as:
  - Specific FPGA resources (using brand-specific implementations of queues and chip resources).
  - Network processors (NPs), including potentially multiple models for implementation, such as treating the NP as a single resource with limited capacity (like an FPGA), or treating each "micro-engine" as an individual resource, connected using an on-chip interconnect fabric.
- Interconnect resources, such as:

  - Customized Internet Protocol communications, similar to TCP (currently available) but with relaxed timing requirements, reflecting the flexibility of one-way, already queued data links.
  - Special-purpose, high-performance clustering interconnects, such as InfiniBand, Myrinet, Fibre Channel, and others as they are developed.
  - DMA transfers between system memory and typical PCI-connected (or PCI-X, PCI-Express, etc.) FPGA development boards.
  - Custom protocols for on-board and inter-board interconnects between FPGAs.

The optimizer, X-Opt, needs to be developed:

- A framework for X-Opt needs to be created which will take as input the X algorithm description and the results of X-Sim, and produce a new resource allocation.
- Multiple optimization algorithms should be developed to improve performance under a number of different metrics, for instance:
  - Total amortized throughput per unit time
  - Average and maximum latency per unit time
  - Total amortized throughput per unit time per resource cost (a function of the used resources)
  - Total amortized throughput per unit time per resource utilization (e.g. logic elements on an FPGA)
- Depending on simulator accuracy (of both X-Sim and the supported simulator set), "embedded" applications with hard real-time constraints could be analyzed and optimized using X-Opt.

# 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] S. S. Bhattacharyya. *Software synthesis from dataflow graphs*. Kluwer Academic Publishers.

[4] Ian Buck et al. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of the 31st international conference on computer graphics and interactive techniques (SIGGRAPH 2004)*, August 2004.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[6] Mentor Graphics Corp. Modelsim. `http://www.model.com`.

[7] Seema Datar. Pipeline task scheduling with application to network processors. Master's thesis, Washington University in St. Louis, August 2004.

[8] 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.

[9] Jonathon Driscoll. Computer aided optimization for the VERITAS project, June 2000. Undergraduate thesis, Washington University in St. Louis.

[10] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[11] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, page 49. IEEE Computer Society, 2000.

[12] GStreamer: open source multimedia framework. `http://gstreamer.freedesktop.org/`.

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

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

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

[16] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, March 2006.

[17] G. Kanbach, D. L. Bertsch, C. E. Fichtel, R. C. Hartman, S. D. Hunter, D. A. Kniffen, B. W. Hughlock, A. Favale, R. Hofstadter, and E. B. Hughes. The project EGRET (Energetic Gamma-Ray Experiment Telescope) on NASA's Gamma-Ray Observatory (GRO). *Space Science Reviews*, 49:69–84, 1988.

[18] Praveen Krishnamurthy. Evaluating performance for hybrid architectures. Doctoral dissertation proposal defense, Washington University Department of Computer Science and Engineering, 2005.

[19] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29th annual ACM symposium on Theory of Computation*, 1997.

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

[21] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using ptolemy. *Journal on VLSI Signal Processing*, 9(1):7–21, January 1995.

[22] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Proceedings of the 13th International Heterogeneous Computing Workshop (HCW 2004)*, 2004.

[23] SCIRun: A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), `http://software.sci.utah.edu/scirun.html`, 2002.

[24] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. Technical Report CSD-TR-97-4, 29, 1997.

[25] T. Sterling, J. Kuehn, M. Thistle, and T. Anastasis. Studies on optimal task granularity and random mapping. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 349–365. IEEE Computer Society Press, 1995.

[26] The GLib team. GLib reference manual. `http://developer.gnome.org/doc/API/glib`.

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

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

[29] J. Tsay, C. Hylands, and E. A. Lee. A code generation framework for java component-based designs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, November 2000.

[30] Eric Tyson. X language specification v1.0. Technical Report WUCSE-2005-47, Washington University Dept. Computer Science and Engineering, 2005.

[31] The World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0. http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation, February 2004.

[32] 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.

[33] *Xilinx System Generator for DSP*. Xilinx, Inc., 2006.

# Vita

Eric J. Tyson

| | |
|---|---|
| **Date of Birth** | November 1, 1982 |
| **Place of Birth** | Menomonie, Wisconsin |
| **Degrees** | B.S. Summa Cum Laude, Computer Science and Computer Engineering, August 2006<br>M.S. Computer Engineering, August 2006 |
| **Professional Societies** | Institute of Electrical and Electronics Engineers (IEEE) |
| **Publications** | 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 *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*. IEEE Computer Society Press, April 2006. |

August 2006