Auto-Pipe and the *X* Language: A Pipeline Design Tool and Description Language

Mark A. Franklin[1], Eric J. Tyson[1],
James Buckley[2], Patrick Crowley[1], and John Maschmeyer[1]

[1]Department of Computer Science and Engineering [2]Department of Physics
Washington University in St.Louis
St. Louis, MO 63130 USA
[1]{jbf, etyson, pcrowley}@wustl.edu [2]buckley@wuphys.wustl.edu

Abstract

Auto-Pipe *is a tool that aids in the design, evaluation and implementation of applications that can be executed on computational pipelines (and other topologies) using a set of heterogeneous devices including multiple processors and FPGAs. It has been developed to meet the needs arising in the domains of communications, computation on large datasets, and real time streaming data applications.*

This paper introduces the Auto-Pipe *design flow and the* X *design language, and presents sample applications. The applications include the Triple-DES encryption standard, a subset of the signal-processing pipeline for VERITAS, a high-energy gamma-ray astrophysics experiment. These applications are discussed and their description in* X *is presented. From* X*, simulations of alternative system designs and stage-to-device assignments are obtained and analyzed. The complete system will permit production of executable code and bit maps that may be downloaded onto real devices. Future work required to complete the* Auto-Pipe *design tool is discussed.*

1. Introduction

For application sets where the input data arrives as a sequential stream and data must be processed in real-time, the use of pipelining is an effective design style for exploiting parallelism. *Auto-Pipe* is a design tool aimed at automating this process. While pipeline architectures are the main focus of the work, more generalized topologies can also be implemented. *Auto-Pipe* includes the following components:

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

- A high-level application language, *X* [21], for specifying a set of relatively coarse-grained tasks to be mapped onto devices (i.e., computational nodes) connected in a general topology. In the initial version, tasks are written using either C or VHDL. Associated with the *X* language is a compiler that integrates the tasks and topology specifications with the components and features given below.
- A set of both generic devices and particular devices (e.g., Xilinx Virtex II) on which tasks execute. The devices may include a combination of single processors, chip multiprocessors, or FPGAs.
- A mapping of tasks to generic or particular devices.
- A simulation infrastructure that permits simulation and performance analysis of the system with alternative task and device mappings.
- A set of loadable modules consisting of compiled computational tasks, loadable FPGA bitmaps, and interface modules. Together with the appropriate hardware, this will permit implementation of an application.

In this paper, the overall design of *Auto-Pipe* and the *X* language is presented and a subset of their operational capabilities are described along with their use in example applications. *Auto-Pipe* was motivated by pipeline design issues that arise in the following application domains: networking and communications; large, mass storage based computation; and real-time scientific experimentally derived data. Additionally, technology developments in the areas of NPs (Network Processors) and chip multiprocessors have made utilization of pipelined designs more attractive from both implementation and cost perspectives. While pipelined applications are of principal interest, *Auto-Pipe* has been designed to handle more general topologies. The application domains of interest are:

Networking and Communications. In this environment, routers and related components must perform real-

1-4244-0054-6/06/$20.00

time examination and processing of data packets, where processing includes such operations as packet routing, classification and encryption. While currently computational pipelines are often implemented using network processors [6], *Auto-Pipe* will permit the exploration and implementation of more general topologies and target platforms. The example problem considered in this paper relates to real-time packet encryption using the DES [1], a common and widely studied algorithm.

Storage Based Supercomputing. The sizes of databases and associated mass storage devices have grown dramatically over the past ten years with systems containing tens of terabytes of data now becoming common. During this time period magnetic bit densities have grown faster than comparable semiconductor bit densities. One result is that a performance bottleneck now exists between processing performance (along with disk-processor interconnect bandwidth) and storage capacities. That is, there are a growing number of applications where processing cannot keep up with the growth of the datasets that provide the driving application inputs.

One approach dealing with this problem is to move pipelined computational capabilities closer to where the data is stored, and stream the data directly from the disk heads to a processing pipeline which, in turn, feeds one or more primary system processors. If the data is partitioned appropriately over a multiple RAID system, then multiple computational pipelines can operate in parallel across the data, providing for even higher performance. This general approach has been presented in [3, 10, 17, 23].

Scientific Data Collection. The third application domain is in the area of scientific data collection. The combination of low cost electronics (e.g., sensors) and low cost communications links, processors, and data storage has led to an explosion in the amount of data being collected in various scientific experiments. The information generally originates in the analog domain, is transformed to the digital domain, goes through a sequence of processing steps (e.g., filtering), and is finally stored away for further processing at a later time. Pipelines are natural architectures for processing this data.

We have focused on an experiment derived from the VERITAS project [22] and on a very common step in many areas of high-energy physics and astrophysics: the processing of time-domain data streams in continuously digitized signals from high speed sensors. In the case of VERITAS, we implement the signal processing tasks responsible for reconstructing characteristics of the Cherenkov pulses registered by an array of sensors and flash analog-to-digital converters (Section 3.2).

The next section presents the overall design of *Auto-Pipe* and the *X* language. Section 3 presents two example applications, DES encryption and VERITAS signal preprocessing. Section 4 presents the mapping of the applications' computational tasks to several pipeline implementations. Performance data is presented and we indicate how that data may be used in obtaining a "good" pipeline design. Section 5 contains a summary of the paper and projected future work in this area.

2. The *Auto-Pipe* System

2.1. System Overview

Many applications can be decomposed into a set of tasks, some of which can be executed sequentially in a pipelined fashion, and some in parallel. With such applications, a large set of interacting design choices are available that relate to determining the best task graph topology, mapping tasks onto generic devices (e.g., processors, FPGAs), and specifying the implementation platform or particular device to be used (e.g., the specific FPGA model).

Auto-Pipe and the *X* language allow the user to specify an application in terms of a set of user-defined coarse-grained tasks or blocks. These blocks communicate using well-defined interfaces for inputs, outputs, and a user-determined static topology. At the user level, interfaces are agnostic to the device on which the task is implemented. The language supports a hierarchical description of blocks where both multiple instances of a block may be invoked, and blocks may be aggregated into higher level groupings. *Auto-Pipe* allows arbitrary feed-forward pipelines, which also permits parallel structures for improved performance.

Clearly, determining the best design choices in terms of topology and task-to-device mapping is a difficult problem. However, *Auto-Pipe* permits investigation of alternative algorithm partitionings and mappings. All interface-to-interface communication is robustly handled by the compiler with ordered queuing and flow control automatically generated, thus significantly reducing programmer effort.

2.2. Design Flow

Figure 1 depicts the overall *Auto-Pipe* design flow with the flow being divided both horizontally and vertically. Horizontally, four main phases are present and range from functional representation and correctness checking, to actual pipeline implementation. Vertically, **A** represents the *X* language specification level; **B** corresponds to the activity of the *X* language compiler and other compilers and simulators (e.g., GCC, Modelsim), and **C** contains performance measurement, optimization and output tools. We focus here on phases 1 and 2, and on levels **A** and **B** that are currently implemented.

In Phase 1, *functional simulation*, levels **A** and **B**, the general algorithm is developed and specified in the *X* lan-

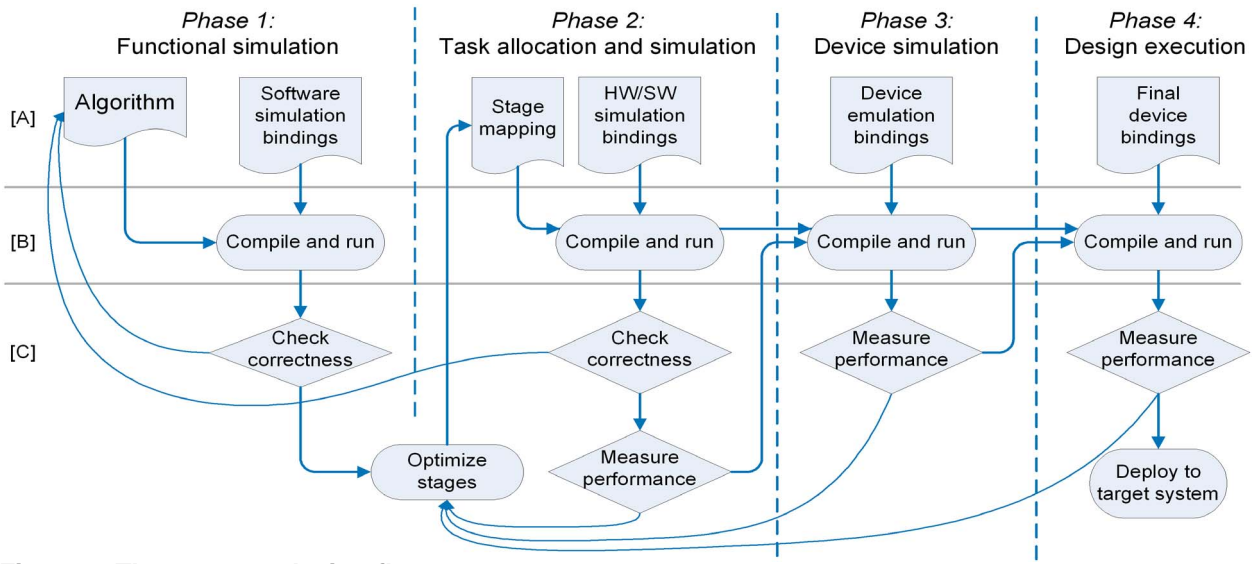


Figure 1. The *Auto-Pipe* design flow. **A**: *X* language,**B**: compilers,**C**: performance measurement, optimization, etc.

guage without concern for timing, execution device/platform issues, and certain aspects of overall performance. The goal at this point is to provide a high level representation mechanism (**A**) that captures aspects of the pipeline and parallel structures the algorithm designer believes are useful, and then executes (**B**) the overall algorithm providing verification of functional correctness. The compiler provides for all of the proper interfaces necessary for proper simulation of the specified task graph.

The next phase focuses on *task allocation & simulation*. In **A** of this phase, tasks/blocks are assigned to generic devices where these devices are either processors or VHDL targeted devices (e.g., FPGAs or ASICs). More than a single task can be assigned to a given device and the hierarchical structure of the language can be used to aid in the specification. When topologies are not linear pipelines, the assignment will initially be done manually. However, for linear pipelines, tools are available to aid in the process[7, 11].

If the platform is a processor, then the appropriate software libraries and compilers are designated. If the platform is an FPGA or an ASIC, then the VHDL code libraries and simulation engines are specified and in **B** the system is again simulated. *Auto-Pipe* uses a message passing TCP interface between the processor based devices, and a file I/O approach when interfacing between processors and VHDL models (simulated in ModelSim). From this, using generic processor and FPGA parameters, the performance statistics associated with each task in the algorithm are gathered. Based on these results, alternative task and device type assignments can be explored and the process repeated.

Phase 3 replicates Phase 2, with the generic devices now identified with particular components. The simulations and performance measurements are further refined and more specific optimizations involving topology, task assignments, and particular device selections are made.

In the final phase, *design execution*, the compiled objects run on the actual devices. This phase is used to further tune the system by testing the design under expected running conditions (e.g. nonzero bus and interconnect utilization). Initially we are targeting a general-purpose hardware system that contains a Xilinx Virtex II FPGA development board and dual AMD Opteron processors.

2.3. The *X* Language

2.3.1. Overview

X is a dataflow-like programming language that can express the nodes and edges of a task processing graph, as well as the configuration, platform definition, and device bindings required to implement an application (a set of tasks) on a set of devices. The language was developed to aid the incremental development of applications distributed across a set of traditional and non-traditional computing platforms, and to aid developers in integrating designs into the *Auto-Pipe* design flow for design optimization. Unlike other projects with similar motivation (see Section 2.4), *X* does *not* intend to be the sole development language within an overall project. Instead, *X* was created to:

• Serve as a connection language, enforcing strict interfaces between the processing components of a system. In doing this, *X* is able to automate the tedious and error-prone process of creating and managing the communication channels between tasks.

• Express a binding or assignment of tasks to devices. A primary goal of the *X* language is to increase the ease with which a developer may experiment with placing tasks on

different devices.

• Be able to associate additional parameters with task blocks and devices. Other software tools may also embed additional information in *X* language objects. The *AutoPipe* toolset uses performance statistics gathered by executing compiled *X* applications in exploring and optimizing the mapping of blocks to platforms.

In the following subsections, a representative subset of the syntactic and semantic features of the *X* language are described in the context of a single example (figure 5).

2.3.2. Descriptive features of language

X supports an assortment of common basic datatypes, and allows for their grouping into composite data types. The basic types supported are `unsigned8`, `unsigned16`, `unsigned32`, and `unsigned64` for 8, 16, 32, and 64 bit unsigned integer data types; `signed8` through `signed64` for their signed counterparts; `float32`, `float64`, and `float128` for IEEE-754/854 single, double, and extended precision floating points, and the `string` type for convenient configuration. These types represent the basic data types common to traditional programming languages.

In addition to basic types, *X* supports both homogeneous and heterogeneous data types comprising multiple data types. The `array` type is an ordered, statically sized array of one kind of data. For instance, `array<unsigned32>[4]` describes an array of four 32-bit unsigned values. `varray` is an array with variable length and one kind of data. Finally, the `struct` type can be used to describe heterogeneous data types with named contents, for example `struct<float32 x,float32 y, unsigned8 count>`. Each composite data type supports a hierarchy of datatypes, such as `array<array<unsigned8>[8]>[4]` which describes a 4 by 8 matrix of `unsigned8`s. Not all device execution platforms will need support for all data types. For instance, while all basic types are supported on a generic processor software platform, VHDL platforms generally do not support the `string` type except as a configuration input.

The primary building block of the *X* language is the Block. Blocks are processing structures with static, well-defined interfaces. They may contain *X* language structures, or be empty *atomic* blocks. Regardless of contents, a Block is the object that may be directly tied to an implementation of the computation (i.e., a computation task specified by C and/or VHDL programs).

```
block Cblock {
    input unsigned8 quant, data;
    output array<unsigned8>[3] result;
    config string setup = "mult";
};
```

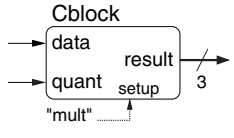


Figure 2. Block declaration example

Blocks may be given any number (including zero) of *input ports*, *output ports*, and configuration inputs. In Figure 2, `Cblock` is an atomic block. It has two inputs, `quant` and `data`, each with type `unsigned8`. The output `result` is an array of three `unsigned8`s. The configuration option `setup` is a `string`, and is given the default value `"mult"`. This means that unless otherwise specified in the *X* input, the value `"mult"` will be passed to the implementation as a static configuration parameter.

Blocks may also be composed of multiple blocks, as long as they do not instantiate themselves infinitely. Figure 3 shows a *composite block*, `Dgroup` that has one input and one output, and contains a simple pipeline of two `Dblock` (`D1` and `D2`). Connections, described below, associate the ports of `Dgroup` with the ports of `D1` and `D2`.

```
block Dgroup {
    input unsigned8 in;
    output unsigned8 out;
    Dblock D1, D2;
    in -> D1 -> D2 -> out;
};
```

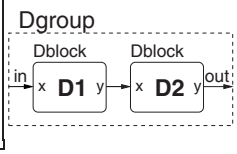


Figure 3. Composite block example

In this composite block example, the expression `in->D1->D2->out;` describes how to connect (with *edges*) the inputs, outputs, and internal blocks. The example depicts three edges, connecting `in` and `D1`, `D1` and `D2`, and `D2` and `out` and is a shortcut for the more verbose notation: `in -> D1.x; D1.y -> D2.x; D2.y -> out;`

When there is only one input or output port, this single *default port* may be inferred as it was for the ports in the above example. Also, for readability and convenience of the common pipelining operation, *X* allows the program to chain multiple default connections on a single line.

Figure 4 depicts two additional *X* language features; block arrays and the *split* operation. In the example, `Dgroup myDg[2];` declares an instance of two `Dgroup` blocks, named `myDg[1]` and `myDg[2]`. Block arrays are statically sized, and they may be configured through their `config` ports either individually or as a group.

The `=<` (*split*) operation in Figure 4 takes the array output of `myC` (introduced in Figure 2) and splits it into three scalar components. These components are then distributed to the ordered list of three input ports given in the right-hand side of the edge. Splits only perform the operation

```
Cblock myC;
Dblock myD;
Dgroup myDg[2];

myC.out =< { myDg[1].in,
             myDg[2].in,
             myD.in};
```

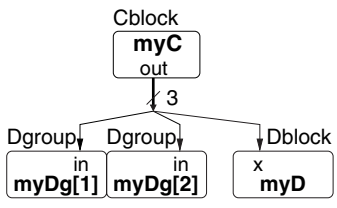
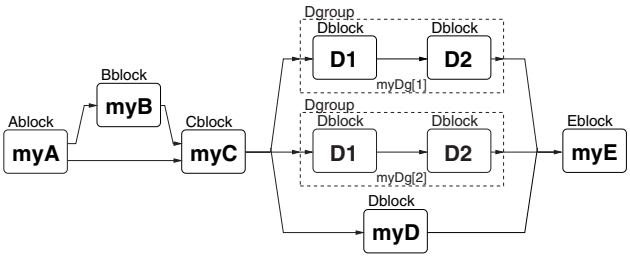


Figure 4. Block array and split example

of dividing an N-element array into its individual elements and routing them to a corresponding number of block inputs. If other functionality is required (e.g. position- or time-multiplexing, round-robin, etc.), the programmer must design a new block or use one of the available blocks that explicitly performs the appropriate operation.

Blocks without inputs or outputs are called *systems* and are considered to be self-contained entities that are instantiated at the highest level; *X* does not assume any data inputs or outputs on its own, so all such sources and sinks are contained within a *system*. In Figure 5 the entire set of blocks constitutes a *system*, using the blocks and syntax introduced earlier. The `use` keyword indicates to the *X* compiler that code is to be generated/synthesized for this system (Top). Multiple instances of the same system are permitted if desired, and they may be passed configuration inputs to distinguish their function.




```
block algo {
    Ablock myA;
    Bblock myB;
    Cblock myC(setup="mult");
    Dblock myD;
    Dgroup myDg[2];
    Eblock myE;

    myA.one -> myB -> myC.quant;
    myA.two -> myC.data;
    myC =< { myDg[1], myDg[2], myD };
    { myDg[1], myDg[2], myD } >= myE;
};
use algo Top;
```

Figure 5. Complete System Example

In addition to describing the processing tasks and systems which are available, developers may express particular device *platforms* on which tasks may be assigned. Platforms are declared in a hierarchy, wherein each platform may be a derivative of a broader class of platforms. For example, a hypothetical "`C-Opteron`" platform would derive from a "`C-x86`" platform, which would derive from a base "`C`". Similarly, an FPGA development board might be called "`VHDL-VirtexII-XYZDevBoard`" and derive from "`VHDL-VirtexII`" and further "`VHDL`". In this way the user can attach their implementing functions (e.g. C functions or VHDL entities) to the appropriate platform. Below is an example of attaching C and VHDL functions to the *X* blocks used in the system example. The `platform` statement indicates its name, parent platform, and any known block implementations and configuration options.

```
platform "C" {
    impl Bblock "func_b";
    impl Eblock "efunction";
    impl Dblock "slowDfunc";
};
platform "C-x86" : "C" {
    impl Ablock "do_a_x86";
    impl Cblock "fastCfunc";
    impl Dblock "fastDfunc";
    config string gatherStats = "FALSE"; // default configuration
};
platform "VHDL" {
    impl Dblock "DBLOCK";
};
platform "VHDL-ModelSim" {
    impl Cblock "CPROC_TEXTIO";
    config string statsOutputFile; // required configuration
};
```

Once the *X* programmer has developed a full system, the next step is to synthesize it in a functional test configuration, as described in the first phase of the *Auto-Pipe* design flow (Section 2.2). To do this, the `target` keyword is employed, for example:

```
target test = { Top };
```

This invokes synthesis of the entire system `Top` to the "test" device, which creates a set of source files that compile to a functional test or simulation of the system.

The subsequent phases of the *Auto-Pipe* design flow are performed by defining and refining new target devices, to which different blocks are allocated. For example, splitting `Top` across three processors could be performed by:

```
device proc[3] : "C-x86";
target proc[1] = { Top.myA, Top.myB, Top.myC };
target proc[2] = { Top.myDg[1], Top.myDg[2].D2, Top.myE };
target proc[3] = { Top.myDg[2].D1, Top.myD };
```

The `device` keyword indicates that an array of three "C-x86" devices exist, and the `target` statements are used to allocate sets of blocks to each device. If the `Dblock` operations were particularly slow, the designer might explore their options by simulating a hardware device as follows:

```
device proc : "C-x86";
device fpga : "ModelSim-VHDL" (statsOutput="mysim.stats");
target proc = { Top.myA, Top.myB, Top.myC, Top.myE };
target fpga = { Top.myD, Top.myDg };
```

With this device allocation, the five parallel blocks in the middle of the system are allocated to an FPGA, and the remainder of the blocks execute on a processor.

2.3.3. Implementation and Related Issues

One of the principal goals of the *X* language is that it provide a straightforward programming model to the developer.

The semantics of the edges between blocks supports this by allowing the programmer to assume a large queue exists between any two connected blocks. The *X* language compiler then ensures that the connections are implemented and provide this "large queue" abstraction for block interconnects.

Edges are also abstracted from the actual code that performs the communication. The programmer may simply place the blocks on either side of an edge onto different devices, and all device interfaces are generated automatically during compilation of the *X* application description. Further parametrization of the edge is possible if desired.

Performance capture is also an important feature of the compiled *X* code. At every stage of the *Auto-Pipe* design flow, the programmer will want to acquire performance statistics as detailed as possible, in order to optimize the mapping of blocks to devices. Every platform supported by *X* therefore supports the capturing of detailed performance information, which integrates into the *Auto-Pipe* toolset. Our current version of *X* language implementations consists of three platforms: a C simulation platform, a C multiprocessor platform, and a VHDL simulation platform.

The C implementations use standard *GNU C*, tested with versions 3.4.4 and 4.0.2 on Linux and Cygwin (Windows) platforms. We support 32- and 64-bit x86-compatible processors, and make use of the RDTSC system clock instruction for accurate performance statistics. Data types and queues use the *glib-2* core libraries. Future versions of *X* will use *glib*'s threading and memory management capabilities.

Execution of the compiled *X* language application representation requires that communication take place between processors, between FPGA devices, and between both processors and FPGAs. Currently, in the first phases of *Auto-Pipe* where the goals are achieving functional correctness and obtaining block level timing data, communication between C processors is performed by TCP using traditional BSD sockets. In the next version of *Auto-Pipe* and the *X* language, the user will be able to select (or provide) among various edge communications models (e.g., Myrinet, shared memory, etc.) that correspond more closely to the underlying hardware system being designed.

The VHDL simulation implementation creates VHDL 1993 syntax behavioral architectures, using FIFO queue structures between blocks and the standard *TEXTIO* package to communicate with C processes using the filesystem. Data types use the standard IEEE 1164 types, as well as floating point types from the proposed IEEE 1076.3 (VHDL200x) specification. Mentor Graphics' *ModelSim* performs the simulation.

2.4. Related Work

Auto-Pipe shares many features with other academic and commercial tools. It draws on developments in performance modeling, graphical and streaming programming languages, hardware/software codesign toolsets and earlier work on implementing dataflow languages. An early review of some of this work is given in [19] and some examples of related work are outlined below.

The programming interface employed by *Auto-Pipe* is similar to many other graphical system programming languages such as *LabVIEW* [16]. Additionally various projects allow for the simplified development of streaming applications using the familiar environment of traditionally sequential programming languages. Most involve a sub- or super-set of the functionality available in C [13], C++ [18], or Java [5]. These projects share the similar goal of easing the development of streaming algorithms in hardware and software [20], however they generally concentrate on fine-grained and implicit dataflow programming, as opposed to the coarse-grained programming of *Auto-Pipe*. In some cases, they are restricted to the limited topologies (e.g., pipelines) and in others they rely on C to VHDL compilation techniques. Currently, *Auto-Pipe* assumes that user selected tasks have already been programmed in VHDL. In the future, however, as C to VHDL compilation improves in the efficiency of the resulting FPGA designs, *Auto-Pipe* will be modified to use such languages (e.g., Handel-C [4], System-C [14]) in specifying tasks, compiling them to VHDL and then integrating the VHDL models into *Auto-Pipe*.

The codesign aspect of *Auto-Pipe* shares features with other systems design projects. *Bluespec* [2], for example, is a hardware design toolset for behavioral synthesis using the SystemVerilog HDL. Bluespec creates accurate C programs and testbenches at all levels of development, including behavioral, timing, and gate-level implementation.

The *Auto-Pipe* infrastructure differs from the above projects in several ways:

(a) *Auto-Pipe* aids the analysis and performance tuning of both pipelined architectures and architectures with more general topologies. New processing infrastructures can be discovered and tested by exploring alternative topologies. Also, connection semantics are not limited to certain synchronization assumptions (e.g. synchronous dataflow).

(b) *Auto-Pipe* supports the ability to easily try both different task to generic device assignments, and generic device to particular device platform assignments. Such devices will eventually include any mix of FPGAs, network processors, clusters, and desktops.

(c) *Auto-Pipe* does not constrain development to any single set of hardware and software languages. Instead, it is a code generation tool for any language for which an *Auto-Pipe* interface has been written. Initially, in-

terfaces for C software development and VHDL hardware development have been created.

3. Example Applications

3.1. DES Encryption

Encryption involves transforming unsecured information into coded information under control of a key. The Data Encryption Standard (DES) operates on 64-bit data using a 56-bit key. To encrypt data blocks longer than 64 bits, several iterations are required.

Triple-DES uses three sequential DES stages to increase the key size. Each stage performs a standard DES encryption using the first, second and third 56-bit keys (64-bits with parity) respectively, and results in a more effective key-length of 168 bits, (versus 56 bits in DES). While the inner pipeline of Triple-DES can be broken into many small stages, we have chosen to demonstrate *Auto-Pipe* using a simple 3-stage pipeline, one stage for each DES block.

3.2. Astrophysics Data Pipeline

In ground-based high-energy astrophysics observations, very high energy gamma-rays and cosmic-ray particles strike the atmosphere and produce Cherenkov light. Energetic gamma-rays have been observed from scientifically interesting sources including supernova remnants and pulsars.

A number of new-generation projects including HESS [15] and the VERITAS [22] project are based on the technique of stereoscopic imaging of Cherenkov light from gamma-ray induced electromagnetic showers. These systems employ large (10 - 17m diameter) mirrors to image the faint flashes of Cherenkov light onto large arrays of photomultiplier tube sensors, each capable of detecting single photon events at sampling rates surpassing 500 MHz. To improve the signal-to-noise ratio for detecting these images against the large diffuse night-sky background light, rigorous signal processing must be performed on the digitized waveforms registered by each sensor channel. The signal must be deconvolved, its signal-to-noise ratio improved, and the timescales that characterize the Cherenkov pulses extracted. Furthermore, an image analysis must be performed across all channels in a telescope to detect and characterize events based on the shape properties of the images produced by the gamma-ray showers. A detailed description of this process is given in Gammel [12] and an example processing pipeline is shown in Figure 6.

In this pipeline, records containing signal waveforms are retrieved from a subset of telescope channels. These are distributed among up to 499 (the total number of channels) signal processing pipelines. In each pipeline, the waveform is zero-padded to a higher resolution and undergoes a

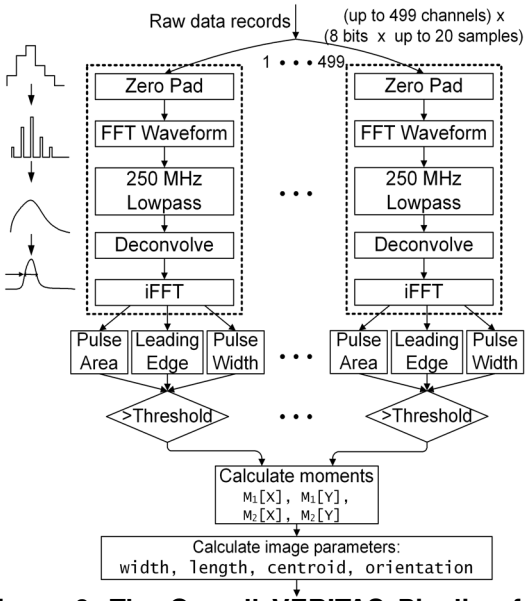


Figure 6. The Overall VERITAS Pipeline for Cherenkov Light Processing

Fast Fourier Transform (FFT). The frequency-domain signal is low-pass filtered and then deconvolved (a vector multiplication with the channel's inverse transfer function [8]). These steps accomplish an interpolation of the input which smooths the waveform and improves the photon pulses' signal-to-noise ratio by reducing signal "smear" from adjacent pulses. This is followed by an inverse FFT that returns the signal to the time domain. Following the processing of each channel, the entire channel set is analyzed and image parameters determined. Our initial efforts aim at employing the *X* language in evaluating and implementing one of the real-time processing pipelines required; for example the leftmost, dot enclosed pipeline shown in Figure 6.

4. Application Implementation & Evaluation

We have written *X* language code to generate the DES and VERITAS applications. Figure 7 contains representative high-level block from each. For brevity, the atomic blocks and top-level blocks containing the data sinks and sources are omitted.

Each code segment directly follows from the discussed application flow. Note, however, that the frequency domain signals (i.e., between the FFT and IFFT blocks) are divided into real and imaginary values. This allows us to parallelize the filtering on each. The implications of this on performance optimization are discussed later. The remainder of this section examines these *X* implementations and their resultant simulation performance with different device assignments. In order to use *Auto-Pipe*, the basic tasks were first coded in a combination C and VHDL.

```
constant array<FLOAT32>[128] FILT_LOWPASS = {...};
constant array<FLOAT32>[128] FILT_DECONV = {...};

block VeritasSigProc {
    input array<UNSIGNED8>[32] in;
    output array<FLOAT32>[256] out;
    ConvertU8F32 conv;
    ZeropadF32 zp(pad = 8);
    FFT256F32 fft;
    IFFT256F32 ifft;
    FilterF32 lpr(filter=FILT_LOWPASS),
              lpi(filter=FILT_LOWPASS),
              dcr(filter=FILT_DECONV),
              dci(filter=FILT_DECONV);

    in -> conv -> zp -> fft;
    fft.real -> lpr -> dcr -> ifft.real;
    fft.imag -> lpi -> dci -> ifft.imag;
};
```

```
typedef array<UNSIGNED8>[8] phrase;

block TripleDESEncrypt {
  input phrase in;
  output phrase out;
  des_encrypt des_e1(keyfile="key1");
  des_decrypt des_d1(keyfile="key2");
  des_encrypt des_e2(keyfile="key3");
  in -> des_e1 -> des_d1 -> des_e2 -> out;
};
```

Figure 7. DES and VERITAS Applications

Table 1 lists the application tasks and the available implementation devices. Once functional correctness was established, performance was measured on an 3.4GHz Pentium 4 single-processor PC which provided base performance metrics. For VHDL performance, we assumed a clock rate of 150MHz based on our experience synthesizing the major components on a Xilinx VirtexII FPGA.

In the following subsections, we investigate a "manual approach" to the optimization component of *Auto-Pipe*. In particular, the performance implications of different stage-to-device allocations is discussed. This corresponds to repeated iterations of Phase 2, *task allocation & simulation*.

4.1. The DES Pipeline

While the inner pipeline of Triple-DES could be broken into many small stages, we have chosen to demonstrate *Auto-Pipe* using a simple 3-stage pipeline, one stage for each DES block. Each of the three pipeline stages takes a single 64-bit input and generates a single 64-bit output. The 56-bit key is not treated as an input since it is only set once. We assume data can be supplied to, and results read from, the pipeline at a rate high enough to ensure the pipeline is a performance bottleneck. Both the encryption and decryption blocks can be implemented on a conventional microprocessor or on an FPGA. The time for each of the identical stages when implemented on either of these platforms is given in Table 1. Using these values, performance of alternative task(s)-to-stage and stage-to-platform assignments can be evaluated.

Function Block	Execution Platform	Execution Time (μs)	Throughput (KOps/s)
Zeropad	Proc	3.90	256.41
FFT	Proc	30.54	32.75
FFT	FPGA	0.87	1153.40
Low Pass	Proc	4.67	214.18
Deconvolve	Proc	5.63	177.78
IFFT	Proc	25.85	38.68
IFFT	FPGA	0.87	1153.40
DES	Proc	49.95	20.02
DES	FPGA	0.11	8823.53

Table 1. Stage performance

	Configuration	Longest Stage (μs)	Throughput (KOps/s)
A	P	146.63	6.82
B	P-P-P	49.95	20.02
C	P-F-P	49.91	20.01
D	F-F-F	0.11	8823.53

Table 2. Performance of various Triple-DES pipeline implementations

The results of four assignments are shown in Table 2. For the simplest case, A, all three tasks are placed on a single processor (P) and must be executed sequentially. Thus, throughput is limited to approximately 6.8 thousand Triple-DES operations per second (KOps/s).

In the next case, B, the Triple-DES stages are implemented on a pipeline consisting of three separate processors (P-P-P). The maximum throughput for the pipeline is now limited by the slowest stage of the pipeline. Since the stages are identical, the throughput is equal to the throughput of a single stage or approximately 20 KOps/s, about three times that of a single processor. This is consistent with our first level approximation which has no message passing overhead.

Case C implements the Triple-DES pipeline on a mixed platform (P-F-P). The first stage consists of a single processor, the second stage an FPGA, and a second processor is used for the third stage. In this case, the total system throughput is again limited by the throughput of an individual stage running on a processor. While this example does not show any performance benefit, it does demonstrate the ease with which a user of *X* can move a block from one target implementation to another.

	Configuration	Longest Stage (μs)	Throughput (KOps/s)
A	P	66.72	14.99
B	P-P-P-P-P	30.54	32.75
C	P-F-P-P-F	5.63	177.67
D	P-F= $\begin{matrix} P-P \\ P-P \end{matrix}$ =F	3.900	256.41

Table 3. Performance of various VERITAS pipeline implementations

The final case, D, implements the entire pipeline on FPGAs (F-F-F). Since the FPGAs are identical, the throughput of the system will be based on the FPGA clock frequency that can be achieved. If this block is internally pipelined on a single FPGA, a very high throughput can be attained. Message passing delays in this case will be negligible. In the case of the DES block, the FPGA implementation completes a single operation every 17 cycles and can be clocked at about 150 MHz. This results in a maximum throughput of about 8,823 KOps/s, or a speedup of over two orders of magnitude over the pipelined processor software implementation.

4.2. The Astrophysics Data Pipeline

As shown in Figure 6, initial VERITAS processing is implemented as a five-stage pipeline. This pipeline is a general pipeline for optimizing the signal reconstruction for a given sensor/electronic channel (Section 3.2). The performance for each of these operations on various execution platforms can be seen in the top seven rows of Table 1. In the VERITAS pipeline, each input contains twenty 8-bit values. The output is an up-sampled signal containing 256 samples.

As in the DES example, the VERITAS pipeline can be implemented in multiple configurations. In the first configuration, A, the entire pipeline is implemented on a single processor. In this case, one would expect very low messaging overheads and overall latency is simply the sum of the individual block latencies. This results in 66.72 μs per operation or a throughput of about 15 KOps/s. Note that this is a bit lower then the result obtained if one adds up the individual times found in Table 1. This is due to the efficiencies associated with executing all the tasks as a single process on a single processor.

The second configuration, B, places each stage onto a separate processor. The FFT stage is the limiting stage, requiring 30.54 μs per operation. The throughput of the entire pipeline is 32.75 KOps/s, a speedup of only about two over the single processor implementation. This is a result of the unbalanced workload in each stage. In fact, the deconvolve stage could be combined with the IFFT stage creating a shorter 4-stage pipeline with only a small loss in performance.

A significant performance improvement over the software implementations can be gained by replacing the FFT and IFFT stages with faster FPGA implementations. In fact, the FPGA implementation used can perform a 256-point FFT or IFFT in 132 cycles and can be synthesized at a rate of 150 MHz on a Xilinx Virtex II. This results in an operation latency of only 0.88 μs. By replacing the processor implementations with FPGAs, the pipeline throughput can be dramatically improved. The pipeline is now limited by the lowpass and deconvolve stages, giving it a throughput of approximately 177 KOps/s. This is a speedup of 5.4 over the multiple processor pipelined version and almost 12 over the single processor implementation. Further improvements can be made by splitting the pipeline after the FFT stage so that the real and imaginary parts are dealt with separately and in parallel. When this is done, the longest latency stage (zeropad) is 3.9 μs, resulting in a throughput of 256 KOps/s. This is 17 times faster than the single processor case. Thus, over an order of magnitude in performance can be gained by pipelining and parallelizing the operation and optimizing the computation of blocks through the use of FPGAs.

5. Summary and Conclusions

This paper has presented an introduction to *Auto-Pipe*, a design tool that addresses many difficulties faced by designers of pipelined algorithms. Applications that make use of pipelined algorithms face a very broad design space. The *Auto-Pipe* tools make development easier by providing a way to logically express such algorithms, obtain performance data, generate "glue code" to connect tasks with well-defined interfaces, and provide tools to optimize the allocation of tasks to pipeline stages where the stages themselves may be on a variety of platforms (e.g., FPGA, ASIC, processor, etc.). While the entire *Auto-Pipe* suite is not yet complete, this paper represents a first presentation of its overall structure and usage. Further developments will focus on providing a richer library of interconnection modules so that simulation of messaging overheads can be evaluated. Additionally, work is progressing on implementing hardware support for *Auto-Pipe* phases 3 and 4.

To implement the "optimize stages" step depicted in Figure 1 and illustrated manually in Section 3, the Phoenix[7] tool for optimizing the task-to-stage mapping of a network processor pipeline will be extended. An important issue that is being considered is just how to enable *Auto-Pipe* to include current network processors and other CMPs (Chip Multi-Processors) that are rapidly becoming available as target implementation platforms.

This paper presented relatively small applications that serve as sample problems. Future research will include applying the *Auto-Pipe* design flow tools and programming

methodology on other more complex problems that fit the general set of domains outlined in the introduction.

One such recently completed application is the complete VERITAS signal analysis pipeline. Figure 6 shows the entire pipeline, including the remaining portions (outside the dashed-line box). Current plans for VERITAS estimate an average data production of 8 megabytes of event data per second with 10 percent live time, or about 24 terabytes per year of operation. For offline data analysis, such a large database introduces many restrictions to the types of queries that can be performed by traditional software processing systems.

Beyond the VERITAS application, other high-performance computing algorithms will be tested using the *Auto-Pipe* system, particularly in the field of computational biology. The HMMer bioinformatics algorithm [9] is one such algorithm that is currently being analyzed for performance improvements using customized hardware. An additional problem now being investigated involves mass spectroscopy and the fast identification of substances in a mass spectrometer by comparing results in real-time with large datasets of peptides. The comparison is computationally complex, but permits a pipelined implementation. Streaming data from the disks holding the dataset, while ingesting data from the mass spectrometer and performing the appropriate computation will require special hardware. *Auto-Pipe* will be used in the system evaluation and design process.

References

[1] American National Standards Institute and International Organization for Standardization. *Announcing the Standard for DATA ENCRYPTION STANDARD (DES)*, volume 46-2 of *Federal Information Processing Standards publication.* American National Standards Institute, December 1993.

[2] L. Augustsson, J. Schwarz, and R. S. Nikhil. *Bluespec Language definition.* Sandburst Corp., 2002.

[3] R. Chamberlain, R. Cytron, M. Franklin, and R. Indeck. The Mercury system: Exploiting truly fast hardware for data search. In *Proc. Int'l Workshop on Storage Network Arch. & Parallel I/Os (SNAPI'03)*, April 2004.

[4] Stephen Chappell and Chris Sullivan. Handel-c for co-processing & co-design of field programmable system on chip. http://www.celoxica.com/.

[5] Michael Chu et al. Object oriented circuit-generators in java. In *Proc. International Symposium on Field-Programmable Gate Arrays for Custom Computing Machines (FCCM'98)*, Washington, DC, 1998. IEEE Computer Soc.

[6] P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk. Network processors: Emerging themes & issues. In *Network Processor Design - Issues & Practices II*. Morgan Kaufmann, September 2003.

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

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

[9] S. R. Eddy. HMMer: profile hidden Markov models for biological sequence analysis. http://hmmer.wustl.edu, 2001.

[10] M. Franklin, R. Chamberlain, M. Henrichs, B. Shands, and J. White. An architecture for fast processing of large unstructured data sets. In *Proc. 22nd Int'l Conf. on Computer Design*, October 2004.

[11] Mark Franklin and Seema Datar. *Network Processor Design: Issues & Practices III*, chapter 11. Elsevier/Morgan Kaufmann Pub., 2005.

[12] Stephen Gammell. *A Search for Very High Energy Gamma-Ray Emission from Active Galactic Nuclei using Multivariate Analysis Techniques*. PhD thesis, University College Dublin, October 2004.

[13] 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)*, Washington, DC, 2000. IEEE Computer Soc.

[14] T. Grotker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Kluwer Academic Pub., 2002.

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

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

[17] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proc. 15th Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP'04)*, October 2004.

[18] Oskar Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proc. 10th Ann. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, Washington, DC, 2002. IEEE Computer Soc.

[19] R. Stephens. A survey of stream processing. *Acta Informatica*, 34, 1997.

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

[21] Eric Tyson. X language specification draft. Technical Report WUCSE-2005-47, Washington University, 2005.

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

[23] Q. Zhang, R. Chamberlain, R. Indeck, B. West, and J. White. Massively parallel data mining using reconfigurable hardware: Approximate string matching. In *Proc. Workshop on Massively Parallel Proc.*, April 2004.