Report Number: WUCSE-2005-26

2005-06-01

Auto-Pipe: A Pipeline Design and Evaluation System

Mark A. Franklin, John Maschmeyer, Eric Tyson, James Buckley, and Patrick Crowley

Auto-Pipe is a tool that aids in the design, evaluation, and implementation of pipelined applications that are distributed across 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. In this paper, the Auto-Pipe design flow is introduced and two sample applications, developed for compatibility with the Auto-Pipe system, are presented. The sample applications are the Triple-DES encryption standard and a subset of the signal-processing pipeline for VERITAS, a high-energy gamma-ray astrophysics experiment. These applications... **Read complete abstract on page 2.**

Auto-Pipe: A Pipeline Design and Evaluation System

Mark A. Franklin, John Maschmeyer, Eric Tyson, James Buckley, and Patrick Crowley

Complete Abstract:

Auto-Pipe is a tool that aids in the design, evaluation, and implementation of pipelined applications that are distributed across 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. In this paper, the Auto-Pipe design flow is introduced and two sample applications, developed for compatibility with the Auto-Pipe system, are presented. The sample applications are the Triple-DES encryption standard and a subset of the signal-processing pipeline for VERITAS, a high-energy gamma-ray astrophysics experiment. These applications are analyzed and one phase of the Auto-Pipe design flow is illustrated. The results demonstrate the performance implications of different task-to-stage and stage-to-platform (e.g., processor, FPGA) assignments.

2005-26

Auto-Pipe: A Pipeline Design and Evaluation System

Authors: Franklin, Mark A.; Maschmeyer, John; Tyson, Eric; Buckley, James; Crowley, Patrick

June 1, 2005

Abstract: Auto-Pipe is a tool that aids in the design, evaluation, and implementation of pipelined applications that are distributed across 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. In this paper, the Auto-Pipe design flow is introduced and two sample applications, developed for compatibility with the Auto-Pipe system, are presented. The sample applications are the Triple-DES encryption standard and a subset of the signal-processing pipeline for VERITAS, a high-energy gamma-ray astrophysics experiment. These applications are analyzed and one phase of the Auto-Pipe design flow is illustrated. The results demonstrate the performance implications of different task-to-stage and stage-to-platform (e.g., processor, FPGA) assignments.

Auto-Pipe: A Pipeline Design and Evaluation System

Mark A. Franklin
John Maschmeyer
Eric Tyson
James Buckley
Patrick Crowley

Mark A. Franklin, John Maschmeyer, Eric Tyson, James Buckley, and Patrick Crowley. “Auto-Pipe: A Pipeline Design and Evaluation System.” Technical Report WUCSE-2005-26, Dept. of Computer Science and Engineering, Washington University, St. Louis, June 2005.

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

A copy of this report is available at `http://cse.seas.wustl.edu`

Auto-Pipe: A Pipeline Design and Evaluation System *

Mark A. Franklin[1,†], John Maschmeyer[1], Eric Tyson[1],
James Buckley[2] and Patrick Crowley[1]
[1]*Department of Computer Science & Engineering* [2]*Department of Physics*
Washington University in St.Louis

Abstract

Auto-Pipe is a tool that aids in the design, evaluation and implementation of pipelined applications that are distributed across 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. In this paper, the *Auto-Pipe* design flow is introduced and two sample applications, developed for compatibility with the *Auto-Pipe* system, are presented. The sample applications are the Triple-DES encryption standard and a subset of the signal-processing pipeline for VERITAS, a high-energy gamma-ray astrophysics experiment. These applications are analyzed and one phase of the *Auto-Pipe* design flow is illustrated. The results demonstrate the performance implications of different task-to-stage and stage-to-platform (e.g., processor, FPGA) assignments.

1 Introduction

There are numerous approaches to achieving computational speedup by performing tasks in parallel. 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 that is being developed to automate the pipeline design process. In its full, eventual form, it will include the ability to:

- Specify, in a high-level application language, a set of sequential, relatively coarse-grained tasks, to be mapped onto a computational pipeline.
- Specify a computational pipeline in terms of a number of stages where each stage may be implemented on one of a variety of platforms (e.g., processors, FPGAs, Network Processors).

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

†Communicating author: Mark Franklin, jbf@cse.wustl.edu; Washington University, Dept. of Computer Science & Engineering, Campus Box 1045, St. Louis, MO 63130.

- Simulate the pipeline under alternative task-to-pipeline stage assignments and alternative stage implementation platforms. Obtain performance data from these simulations to guide overall design.
- Given the first two items above, and a given specific design (i.e., a mapping of tasks to platforms): a) Produce the interface infrastructure necessary, and b) Integrate the program code and FPGA design specifications for a pipeline implementation of the desired application.

In this paper, the overall design of *Auto-Pipe* is presented and a subset of its operational capabilities is 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 CMPs (Chip Multi-Processors) have made utilization of pipelined designs more attractive from both implementation and cost perspectives. The application domains of interest are given below.

- **Networking and Communications:** In the networking environment, routers and related components must examine and process packets on communications data in real-time and perform a host of operations (e.g., packet routing, classification, encryption, etc.) on the data. In this domain, computational pipelines are often implemented using network processors [5]. *Auto-Pipe* will enable specification and implementation of pipelined architectures targeted to this domain.

In this context, one example problem we have examined relates to real-time packet encryption using the DES [1] algorithm. DES is a common and widely studied algorithm. The algorithm involves transforming unsecured information into coded information with the transformation being controlled by an algorithm and a key. The standard DES algorithm operates on 64-bit data blocks using a 56-bit key. For encryption of longer data blocks, multiple iterations are required with encryption of a single block requiring 16 stages. Triple-DES, considered here, is a variant of DES requiring three sequential DES stages. This increases the effective key size to 168 bits, thus making it more secure than DES.

While the inner loop of Triple-DES can be broken into many small stages, we demonstrate *Auto-Pipe* using a simple 3-stage pipeline, one stage for each DES block (Section 3.1).

• **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. Due to technology advances, magnetic bit densities have grown faster than comparable semiconductor bit densities. The result has been a significant decrease in the cost of disk-based magnetic storage. This decrease in cost, coupled with the increasing demands from government and industry to store and process ever increasing amounts of information, has led to a performance bottleneck. That is, there are a growing number of applications where processing cannot keep up with the growth of the datasets that provide the driving inputs to the computation.

New system architectures oriented towards alleviating this problem are now being developed. One approach is to move pipelined computational capabilities (e.g., processors, FPGAs, etc.) 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, 22] and its use in a computational genetics application may be found in [16]. A commercial unit that aims at implementing this sort of architecture is the SGI Altix system [20] where FPGA "brick" nodes are being designed and integrated into a shared memory multiprocessor [18].

• **Scientific Data Collection:** The final application domain is in the area of scientific data collection. The combination of low cost electronics (e.g., sensors) and the availability of low cost communications links, processors, and data storage has led to an explosion in the amount of data being collected in the course of various scientific experiments. Many of these projects involve gathering information

in real-time from a group of sensors. 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. Thus, a pipeline of processing steps is required. We focus here on an experiment derived from the **VERITAS** project [21] and investigate one particular but 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 specific case of VERITAS, we implement the signal processing block responsible for reconstructing characteristics of the Cherenkov pulses resgistered by an array of sensors and flash analog-to-digital convertors (Section 3.2).

The next section presents the overall design of *Auto-Pipe*. 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 that include both processors and FPGAs. 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

For algorithms that can be implemented in a pipelined manner, a large set of interacting design choices is available. *Auto-Pipe* addresses four key high-level choices:

- Determine the set of sequential tasks that implement the application algorithm.
- Determine the number of stages in the pipeline.
- Determine the type of implementation to be used for each stage (e.g. processor, FPGA, etc.)

- Determine the mapping of tasks to stages. Multiple tasks may be mapped to a single stage.

Determining the best design choices is a difficult problem [1] since the design space is large. The design goal for *Auto-Pipe* is to permit the designer to explore this space incrementally and in detail. *Auto-Pipe* addresses pipelined algorithms with a coarse-grained data-flow approach. As such, each algorithm is described as a set of connected atomic processing tasks. Each task has a well-defined interface of inputs, outputs, and static configuration information with the interfaces being agnostic to the platform on which the task is implemented.

2.2 Design Flow

Figure 1 depicts the *Auto-Pipe* design flow. The flow is divided into four main phases that proceed from functional representation and correctness checking, to actual pipeline implementation. Since *Auto-Pipe* is still under development, the examples in this paper focus on Phase 2 of the flow.

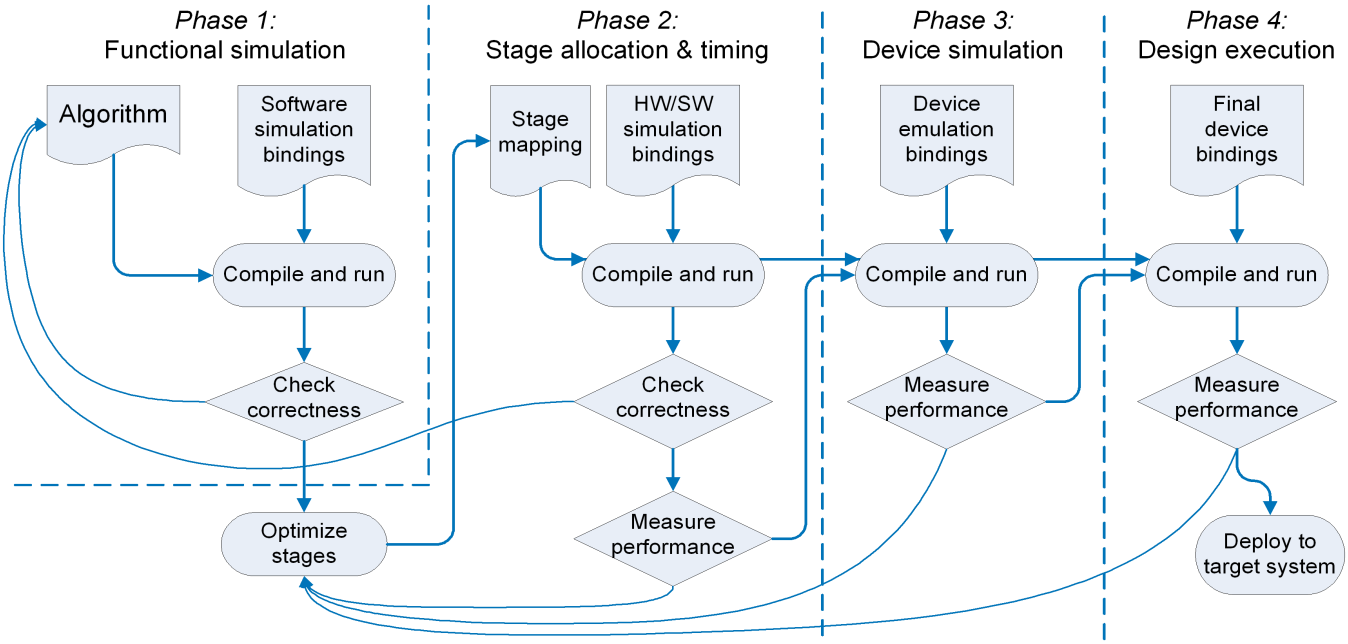

Figure 1: The *Auto-Pipe* design flow.

In the first phase, *functional simulation*, the general algorithm is developed without concern for

[1] A subset of the task to stage mapping problem is considered by Datar and Franklin [6, 11].

timing, execution platform issues, or overall performance. The algorithms of interest are composed of a set of basic tasks that generally communicate in a pipelined manner. *Auto-Pipe*, when completed, will provide a block oriented input language that will permit the user to represent the algorithm in terms of connected and interacting sequences of tasks that are then compiled and executed. The goal of this phase is to provide a high level representation mechanism followed by execution of the overall algorithm permitting verification of functional correctness.

The next phase focuses on *stage allocation & timing*. In this phase, three aspects of design are considered. The first two correspond to components of task allocation. First, the tasks defined in Phase 1 are assigned to pipeline stages. While initially this is done manually, task assignment aids are also available [6, 11]. Second, the platform on which the stages are to be executed is specified. 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. Given these first two elements, *Auto-Pipe* uses a common MPI (Message Passing Interface) interface between the pipeline stages, creates the appropriate software executables, and then executes the simulation. 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-to-stage, and stage-to-platform (e.g., processor or FPGA) assignments can be considered to improve performance. This is the *Auto-Pipe* phase that is illustrated in the examples that follow.

Following the general hardware/software simulation is the *device simulation* phase. This phase particularizes the hardware and software device models of the previous phase to simulate or emulate the real devices on which the application may be eventually deployed. For example, say that a given pipeline contains two stages with the first targeted for implementation on a specific general purpose processor, and the second on a particular FPGA. In this phase, the C code implementation of the

first stage is compiled for execution on a simulation model of the processor, and the VHDL model is parameterized to the selected FPGA component (e.g., back annotation from the synthesized logic). The two are then simulated together and performance statistics generated are used to further optimize the task and device mappings.

In the final phase, *design execution*, the compiled objects run on the actual devices. This phase is used to further tune the pipeline 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 an FPGA (Xilinx Virtex II) and dual Opteron processors (AMD). A later version of this system will include a network processor.

2.3 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, and hardware/software codesign toolsets. 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* [15]. 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++[17], or Java[4]. These projects share the similar goal of easing the development of streaming algorithms in hardware, however they concentrate on fine-grained and implicit dataflow programming, as opposed to the coarse-grained programming of *Auto-Pipe*. These projects are complementary with *Auto-Pipe*, and their use as an implementation language for *Auto-Pipe* processing blocks is potentially useful.

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 three distinct ways:

1. *Auto-Pipe* aids the analysis and performance tuning of pipelined architectures. New processing infrastructures can be discovered and tested by modeling the performance of hypothetical computational resources.
2. *Auto-Pipe* strongly supports the ability to effortlessly move processing components onto different devices. The same compiler can use the same functional description and produce code for different systems of devices. Such devices will eventually include any mix of FPGAs, network processors, clusters, and desktops.
3. *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, we have such interfaces in C and C++ for software development and VHDL for hardware development.

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 (e.g., encrypting a 256-bit block requires four iterations).

Encryption of a single block requires 16 stages (see Figure 2) with the key being used to generate 16 48-bit subkeys, one key for each stage. In each stage, the 64-bit input block is split into two 32-bit halves. The right half is expanded from 32 bits to 48 bits using a fixed table and this result is XORed with the stage's subkey. Using another set of tables (S-boxes), the 48-bit value is then divided into eight 6-bit segments and transformed into a single 32-bit result. This value is then permuted using

another table and the result XORed with the left half of the original input. This final value is used as the new right half for the next stage, and the original right half is used as the new left half.

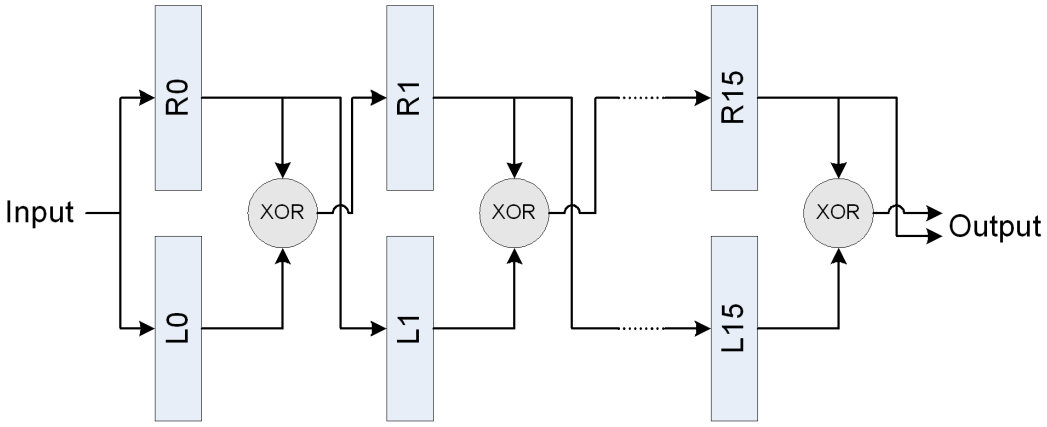

Figure 2: Triple-DES algorithm block diagram. L blocks $\equiv$ left 32 bits; R blocks $\equiv$ right 32 bits.

Triple-DES, a variant of 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 (VHE) gamma-rays and cosmic-ray particles strike the atmosphere and create a shower of particles that produce Cherenkov light. These VHE gamma-rays, reaching energies as high as 10TeV have been observed from supernova remnants and pulsars.

A number of new-generation projects including HESS [14], MAGIC [19], and the VERITAS [21] 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

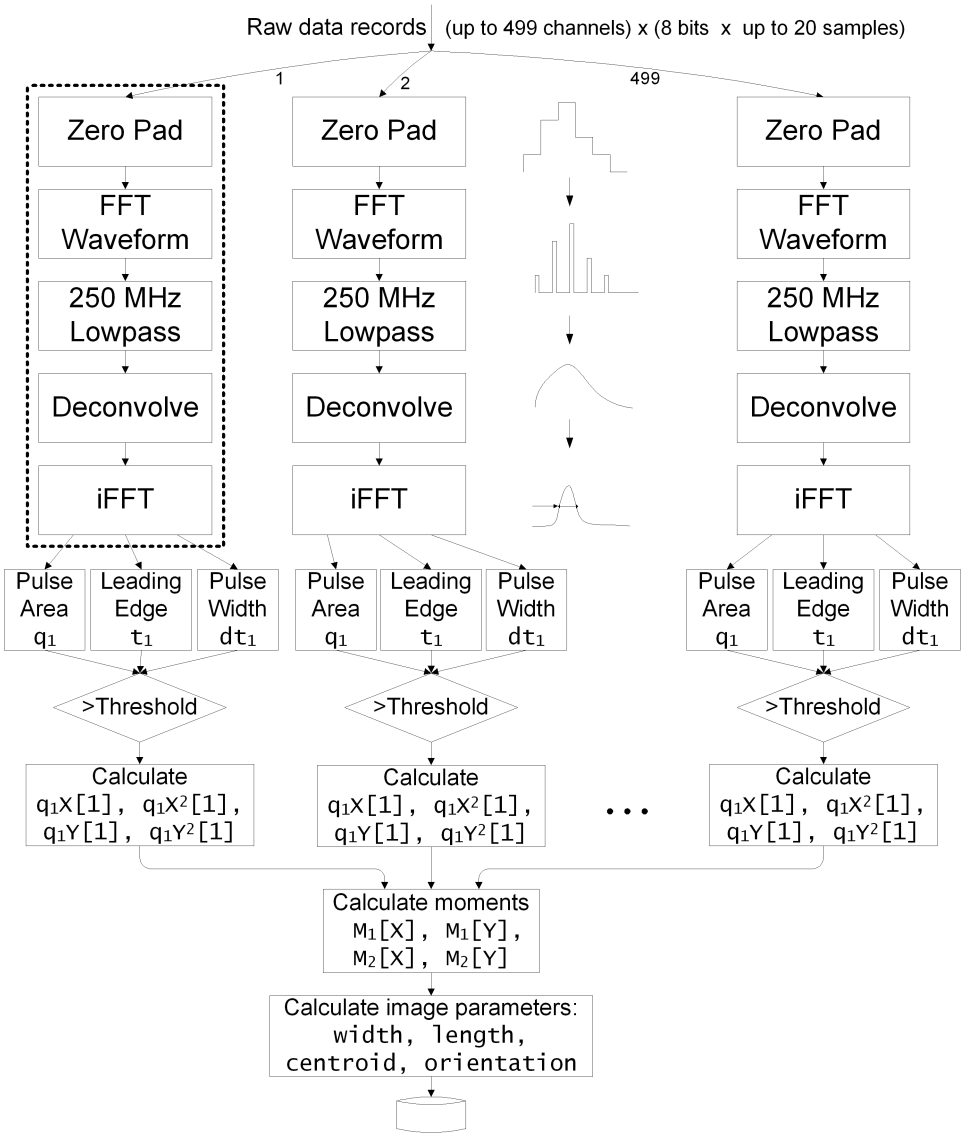

Figure 3: The Overall VERITAS Pipeline for Cherenkov Light Processing

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/electronics channel. The signal must be deconvolved, its signal-to-noise ratio improved, and the timescales that characterize the Cherenkov pulses extracted. Furthermore, an image processing analysis must be performed across all channels in a telescope to detect and characterize Cherenkov events based on the

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 3.

In this pipeline, records containing signal waveforms are retrieved from a subset of telescope pixels. These are distributed among up to 499 (the total number of pixels) signal processing pipelines. In each pipeline, the waveform is zero padded to a higher resolution and undergoes a 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 [7]). These steps accomplish a sinc interpolation of the input which smoothes the waveform, and subsequently improves the photon pulses signal-to-noise ratio thus helping to reduce 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 pixel set is analyzed and gamma-ray images determined.

Our initial efforts aim at employing *Auto-Pipe* in evaluating and implementing one of the real-time processing pipelines required; for example the left most, dot enclosed pipeline shown in Figure 3.

4 Implementation and Performance Evaluation

This section examines *Auto-Pipe* Phase 2 implementations of the two prototype applications. As shown in Sections 3.1 and 3.2 these applications are easily expressed as a simple linear pipeline of tasks. In order to use *Auto-Pipe*, the basic tasks were first coded in a combination C and VHDL. Table 2 lists the application tasks and the available implementation platforms. Once functional correctness was established, performance was measured on an 3.4GHz Pentium 4 single-processor PC. Message passing between processors used the MPICH implementation of MPI. ModelSim and Synplify Pro were used to obtain approximate performance for VHDL implementations on a selected FPGA (a Xilinx Virtex II). From the simulations, data was obtained for each task executing on one or both

platforms.

Auto-Pipe provides for generation of standard MPI [9] interfaces between sequential tasks assigned to processors, and for MPI like interfaces for processor to FPGA, and FPGA to processor interfaces. The overhead associated with moving messages between stages depend on a number of factors. If the stages are assigned to a single processor, then we assume that message passing consists of a simple function call and takes negligible time. If message passing crosses platforms (e.g., processor $\rightarrow$ FPGA; FPGA $\rightarrow$ processor; processor $\rightarrow$ processor) then the overhead consists of two primary components: the processor overhead and the transfer overhead. The processor overhead is the amount of additional processor time required to generate the message and send (or receive) it. This step must be carried out before the processor is ready to work on the next input from the pipeline. It must thus be added directly to execution time associated with the task. The transfer overhead is the time required for the data to arrive at the next pipeline stage and may overlap stage processing time. This value will increase the latency of pipeline, but will not affect the throughput. Naturally, the actual value for the transfer overhead will depend on the interconnect used. Note that in situations where stages have been assigned to the same processor platform, the stages communicate using MPI calls that are optimized to the single processor environment. In this case, the overheads are sufficiently low that they are not considered in the analysis. The measured performance of the MPI processor messaging overhead is given in Table 1. The execution times and throughputs (1/execution time) for each task taken separately, both with and without messaging overhead, are shown in Table 2.

In the following sections, we investigate a "manual approach" to the iterative optimization component of *Auto-Pipe*. In particular, the performance implications of different stage-to-platform allocations is discussed. This corresponds to repeated iterations of Phase 2, *stage allocation & timing*. In effect, we have done by hand what the *Auto-Pipe* tool will automatically do in its first three phases to

explore the design space of a pipelined algorithm.

Message	Base Processor (μs)	Base Processor Overhead: Overhead/KB: (μs)
Send	4.53	1.68
Receive	0.84	0.012

Table 1: Messaging Overhead

Function Block	Execution Platform	Execution Time (ET): (μs)	Throughput (TP): (KOps/s)	ET with Overhead:(μs)	TP with Overhead (KOps/s)
Zeropad	Proc	0.937	1067.24	9.694	103.15
FFT	Proc	52.163	19.17	60.920	16.41
FFT	FPGA	6.880	145.35	6.880	145.35
Vector Mult.	Proc	5.812	172.06	14.569	68.64
IFFT	Proc	55.487	18.02	64.244	15.57
IFFT	FPGA	6.880	145.35	6.880	145.35
DES	Proc	45.060	22.19	50.446	19.82
DES	FPGA	0.106	9400.00	0.106	9400.00

Table 2: Stage performance with & without messaging overhead

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 the pipeline 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 can be seen on the bottom two rows of Table 2. Using these values, along with overhead estimates for the message passing mechanisms used to transfer results from one stage to the next, estimates

of the performance of alternative task(s)-to-stage and stage-to-platform assignments can be evaluated directly.

	Configuration	Longest Stage (μs)	Throughput (KOps/s)
A	P	135.180	7.5
B	P-P-P	50.446	19.8
C	P-F-P	50.446	19.8
D	F-F-F	0.106	9400.0

Table 3: Performance of various Triple-DES pipeline implementations

The results of four assignments are shown in Table 3. For the simplest case, A, all three tasks are placed on a single processor (P). While message passing can be optimized into simple function calls creating negligible message passing overhead, all three stages must be carried out sequentially. Thus, throughput is limited to approximately 7.5 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) using standard MPI interface calls. 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, including processor overheads, or approximately 19.8 KOps/s. The speedup of 2.68 is slightly less than 3, which is consistent with the expected speedup for a three-stage pipeline.

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. Here the stages implemented on processors will still contain the message passing overhead, while the FPGA stage does not. This is due to our assumption that message setup and reception can be pipelined at the same clock frequency as the main functional block. 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 *Auto-Pipe* can move a block from one target implementation to another.

The final case, D, implements the entire pipeline on FPGAs (F-F-F). Unlike the processor cases, there is no throughput impact due to message passing overheads. As a result, message passing only increases the latency. Thus, the throughput is identical for pipelines implemented on a single or multiple FPGAs (assuming, as is the case here, that the FPGA is large enough to hold all the stages). The throughput of the entire system will be based on the clock frequency attained for the functional block. If this block is internally pipelined, a very large throughput can be attained. In the case of the DES block, the FPGA implementation completes a single operation every 17 cycles and can be clocked at about 160 MHz. This results in a maximum throughput of about 9,400 KOps/s, or a speedup of nearly 500 over the pipelined processor software implementation and over three orders of magnitude over the single processor implementation.

4.2 The Astrophysics Data Pipeline

As shown in Figure 3, 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 and has been discussed in Section 3.2. The performance for each of these operations on various execution platforms can be seen in the top rows of Table 2. In the VERITAS pipeline, each input contains twenty 8-bit values. The output is an upsampled signal containing 256 samples.

	Configuration	Longest Stage (μs)	Throughput (KOps/s)
A	P	120.21	8.32
B	P-P-P-P-P	64.24	15.57
C	P-F-P-P-F	14.57	68.64

Table 4: Performance of various Veritas pipeline implementations

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, there are no messaging overheads and the latency of a single operation is simply the sum of the individual block latencies. This results in 120.2 μs per operation or a throughput of 8.32 KOps/s.

The second configuration, B, places each stage onto a separate processor. In this case, processor messaging overheads must be taken into account, and the throughput of the entire pipeline is limited by the throughput of its slowest stage. The IFFT stage is the limiting stage, requiring 64.24 μs per operation. The throughput of the entire pipeline is only 15.57 KOps/s, a speedup of only 1.87 over the single processor implementation. This is a result of the unbalanced workload in each stage. In fact, the zeropad stage could be combined with the FFT stage (creating a single-stage latency of 61.86 μs) and the two vector multiply stages could be combined together (creating a single-stage latency of 20.38 μs) with no loss in performance. In other words, the five-stage pipeline could be reduced to three-stages while maintaining the same throughput.

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 1032 cycles and can be synthesized at a rate of 173 MHz on a Xilinx Virtex II. This results in an operation latency of only 6.88 μs. By replacing the processor implementations with FPGAs, and by again noting that the messaging overhead can be pipelined in with the function on an FPGA, the pipeline throughput can be dramatically improved. The pipeline is now limited by the vector multiply stages, giving it a throughput of approximately 68.64 KOps/s. This is a speedup of 4.4 over the multiple processor pipelined version and 8.25 over the single processor implementation. Thus, nearly an order of magnitude in performance can be gained by pipelining 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 *Auto-Pipe* tool is not yet complete, this paper represents a first presentation of its overall structure and usage.

Auto-Pipe was developed to focus on the needs of specific applications in communications and scientific data collection. In this paper, we have presented prototype examples representing the Triple-DES encryption standard and the VERITAS high-energy astrophysics experiment. The performance of each application was analyzed and selected portions of the *Auto-Pipe* design flow were illustrated. In the Triple-DES application, for example, four alternative pipeline designs were considered and the best one, a pipeline mixture of processors and FPGAs, had a throughput over two orders of magnitude higher than a pure processor pipeline.

We are now in the development stage of a language to describe the interconnection of basic processing task blocks and the presence of implementations of each block in a hierarchy of platforms. This language will be integrated into the *Auto-Pipe* design flow as the method to express algorithms and device bindings to the compiler. The compiler will then be able to create the entire distributed application (i.e. software programs and hardware descriptions) from this high-level language input. Thus, while in the paper the tasks were manually integrated into the pipeline structures using MPI-like interfaces,

later versions of *Auto-Pipe* will perform these and other composition operations. Thus, in the final system, user-supplied task implementations will access a uniform interface to receive and transmit data between the task block's inputs and outputs. A library of software-software and hardware-software communications routines will be available to *Auto-Pipe*. The proper routines will be automatically chosen by the tool to connect interfaces across different platform devices, or among tasks on the same device. To implement the "optimize stages" step depicted in Figure 1 and illustrated manually in Section 3, the Phoenix[6] 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 application is the complete VERITAS signal analysis pipeline. Figure 3 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. Future application of *Auto-Pipe* will focus on developing appropriate computational pipelines for rapid processing of this dataset. This will be useful in exploring various astrophysical problems including the identification of rare astrological events.

Beyond the VERITAS application, other high-performance computing algorithms will also be tested using the *Auto-Pipe* system, particularly in the field of computational biology. The HMMer

bioinformatics algorithm [8] is one such algorithm that is currently being analyzed for performance improvements using customized hardware. An additional problem now being investigated involves mass spectroscopy. The issue revolves around 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 Architecture and Parallel I/Os (SNAPI'03)*, pages 65–72, April 2004.

[4] Michael Chu, Nicholas Weaver, Kolja Sulimma, Andr DeHon, and John Wawrzynek. Object oriented circuit-generators in java. In *Proc. International Symposium on Field-Programmable Gate Arrays for Custom Computing Machines (FCCM'98)*, Washington, DC, USA, 1998. IEEE Computer Society.

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

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

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

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

[9] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[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*, pages 280–287, October 2004.

[11] Mark Franklin and Seema Datar. *Network Processor Design: Issues and Practices*, volume 3, chapter 11, pages 219–244. 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)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[14] W. Hofmann. Status of the high energy stereoscopic system (H.E.S.S.) project. In *27th International Cosmic Ray Conference*, 2001.

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

[16] 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)*, pages 365–375, October 2004.

[17] Oskar Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, page 67, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Steven Miller. Requirements for scalable application specific processing in commercial HPEC. In *8th High Perf. Embedded Computing Workshop (HPEC'04)*, September 2004.

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

[20] Silicon Graphics, Inc. SGI Altix 3000. `http://www.sgi.com/servers/altix`.

[21] 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.

[22] 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 Processing*, April 2004.