Simulation of Streaming Applications on Multicore Systems

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

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

Simulation of Streaming Applications on Multicore Systems[1]

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

Abstract

This paper considers "streaming" applications that are implemented on multicore systems. The paper explores the use of the simulation component (X-Sim) of the Auto-Pipe development system. Under Auto-Pipe, users consider streaming application development in terms of sets of tasks that can be viewed as nodes in an acyclic graph. These nodes can be mapped to computational resources (e.g., General Purpose Processors (GPPs), Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), etc.), and the application can be simulated and subsequently deployed to a real system. In this paper Auto-Pipe is introduced and then applied to a streaming application, VERITAS. Here, gamma ray observation data from an array of telescopes is processed by a multicore system. This paper shows how the processing algorithm can be developed as a set of parallel pipelined tasks. It evaluates the effects of different task-to-core mapping strategies on overall performance. Simulated performance results are obtained for systems containing up to 16 cores. Actual deployed performance results for 1,2,3 and 4-core systems are gathered and compared with simulation results.

1. Introduction

Multicore systems present opportunities for large speedups and have become the standard offering for the dominant general purpose processor vendors (e.g., Intel and AMD). Commonly, the multiple processors are used to support the execution of distinct processes that are either independent or require infrequent inter-process communications and synchronization. These processes are scheduled by the OS. If the processor cores work together, as one would like in a parallel processing implementation of an application, then synchronization and inter-thread communication is necessary and mechanisms such as mutual ex-

clusion locks and/or semaphores must be employed.

There are various problems associated with using such approaches for exploiting the parallelism that may be found in a given application algorithm. One concern relates to the communications and synchronization overhead associated with the use of "heavy" processes. This problem is solved, or at least reduced, through the development of more effective hardware and OS techniques (i.e., light-weight threads). Perhaps a more difficult issue concerns the programming of parallel applications using a thread-based paradigm. Developing and debugging parallel applications that require extensive communications and synchronization is generally difficult and error-prone.

This paper presents, for a selected application domain, an approach to exploiting parallelism and pipelining in a manner that simplifies parallel program development, simulation, and deployment. The application domain of interest is what is broadly referred to as "streaming" applications; that is applications that are driven by streams of data that must be processed in a designated sequence. Such applications are typical of real-time data collection and processing systems, whether the data derives from a scientific experiment or from a financial market feed. Numerous applications are also found in the processing of real-time speech and video derived data.

Such applications can often be expressed in terms of parallel pipelines of tasks where individual stages may themselves consist of parallel execution components. In this paper, we briefly present a system called Auto-Pipe (detailed descriptions can be found in [5, 6, 13]) that has been developed to ease the expression, simulation, and deployment of streaming applications on a variety of computational resources. At the highest level, applications are expressed in terms of directed acyclic graphs where nodes in the graph execute standard compiled (say C++ or VHDL) code that executes on selected computational resources. While Auto-Pipe permits use of a diverse set of resources (e.g., FPGAs), in this paper we restrict ourselves to multicore systems.

One key feature of the system is that interconnections between the graph nodes are modeled (and to an extent im-

plemented) as queues. Viewing the overall application in terms of a graph where interconnections are queues simplifies expression of the overall application as well as individual tasks, and eliminates the need for explicit synchronization, thus easing the program development process. Auto-Pipe, through its X language, hides low level implementation and communication details from the application designer by providing a library of implementations and a simple queue-based interface.

Approaches such as OpenMP [3], for example, require the use of explicit synchronization primitives and, depending on the application, programming such applications can be difficult. Message-passsing systems such as MPI [8] are not streaming languages and thus are not tailored to the sort of applications associated with directed acyclic graphs. Some streaming languages and systems include StreamIt [11] and StreamC [4]. These systems, while generally easier to use than OpenMP and MPI, have the drawbacks that they are each specific to a single target architecture. StreamIt, for example, and its associated simulator btl [10] are targeted specifically to the RAW processor [14] developed at MIT. Similarly StreamC and its associated simulator ISim [1] are targeted to the Imagine processor [1] developed at Stanford.

Auto-Pipe, on the other hand, is an extensible framework for supporting streaming application development on a variety of platforms (e.g., "heterogeneous" systems). It does this, in part, by decoupling the coordination language from the block implementation language. Application tasks, or blocks, are implemented in a language appropriate for the computational execution platform (e.g., C/C++ for processors, VHDL or Verilog for FPGAs). Streaming interactions between blocks are coded in the X coordination language.

This paper briefly introduces Auto-Pipe, illustrates how a simple application is developed, and then proceeds to consider a scientific application, VERITAS, that involves real-time collection and processing of gamma ray observation data. The focus is on illustrating how different mappings (allocations) of the computational graph nodes to processor core resources can be explored with Auto-Pipe and how relative performance can be obtained. Simulation results are compared with deployed execution results on 1,2,3 and 4-core systems, while additional simulation results are gathered to predict performance on larger 8 and 16-core systems.

2. Auto-Pipe Toolset

The Auto-Pipe toolset is targeted primarily towards streaming computation applications. These applications typically involve a large amount of data flowing sequentially through a pipeline of computational stages, with each stage performing an incremental computation. Algorithms

for streaming applications can often be represented using an acyclic dataflow graph. Figure 1 shows a dataflow graph for a simple example streaming application that serves as an illustrative example.
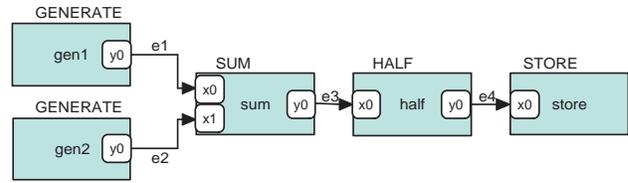

Figure 1. Algorithm for the `test1` **example**

The algorithm represented by this dataflow graph consists of five *tasks*, shown in the graph as *blocks*. A *task* may represent any computational process, from a simple addition to arbitrarily complex computations. A *block* is the representation of a computational task in a dataflow graph. Blocks are connected to each other by *edges* which are dataflow graph representations of communication channels between tasks. *Edges* can be thought of as infinite queues that are present between blocks. The dataflow graphs considered are acyclic in that there is no path by which data can re-enter a block it has passed through. Although this is a restriction, many high performance scientific computations can either be represented by such graphs, or their data intensive computation component can be reduced to such a representation.
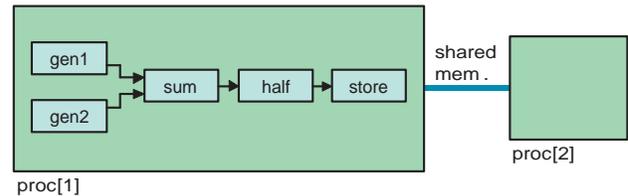

Figure 2. A one-core mapping of `test1`

Consider now deployment of the given example application to a sample processing architecture. Figure 2 shows a two-processor multicore system where the two symmetric cores communicate via a shared memory. Also shown here is a sample *mapping* of the algorithm to the target architecture. *Mapping* an algorithm to an architecture is the process of assigning each block to a Computational Resource (CR), and each application edge to an Interconnect Resource (IR). In this initial mapping, the entire algorithm has been mapped to a single core.

The general framework for Auto-Pipe supports multiple computation resources (e.g., GPPs, FPGAs, GPUs, etc.), as well as a variety of interconnect resources (e.g., shared memory, TCP/IP, PCI-X, file-based, etc.). This allows the
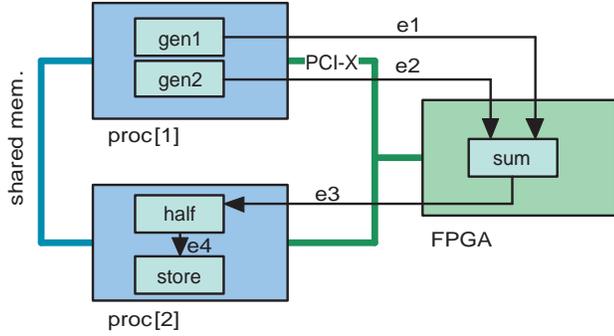
Figure 3. A heterogeneous mapping

integration of multiple parallel computing domains in a single heterogeneous system. Currently, Auto-Pipe supports native execution and basic simulation on processor cores, simulation of HDL (hardware description language) implementations in ModelSim [9], and hardware deployment on FPGAs [2]. In Figure 3, the sample `test1` application has been mapped and deployed to a heterogeneous architecture composed of general purpose processors and an FPGA connected by a PCI-X bus.
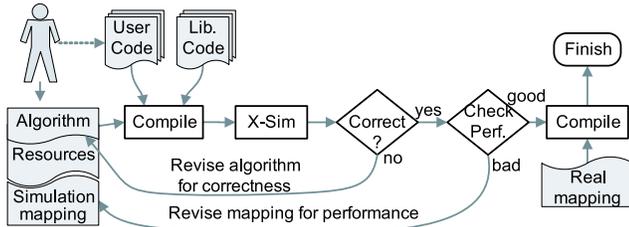

Figure 4. Auto-Pipe design flow

Figure 4 depicts a typical Auto-Pipe design flow to optimize an application's mapping. In this flow, the user provides an X Language [12] description of the algorithm, the resources available, and a mapping of the algorithm components onto simulation resources. The user also may provide implementations ("user code") of any blocks not already written. These components are input to the X Language compiler, which produces binaries that may be run by X-Sim. The results of the execution can then be analyzed to determine correctness; any errors here necessitate a change to the algorithm or user's code. If the application runs correctly, then its performance is analyzed. Performance is improved by modifying the mapping until it meets the users needs. At this point a mapping onto real resources is created from the simulation mapping, compiled, and deployed onto the target system.

The Auto-Pipe infrastructure makes the evaluation of a variety of target architectures and mappings a straight forward task for application developers. In this paper, we focus on the simulation and performance profiling of streaming applications targeted to multicore systems.

3. X-Sim Simulator

X-Sim [6, 7] is a *federated*, trace-based simulation system; that is a simulation system that uses different simulators for simulating different parts of the whole system being investigated. For example, X-Sim uses native execution on processors to simulate execution on GPPs, but uses Model-Sim to simulate execution on FPGAs. The term *trace-based* implies that X-Sim makes use of data traces (marked 'D' in figures in this paper) and timing traces (marked 'T') to keep track of intermediate data and simulation dataflow.
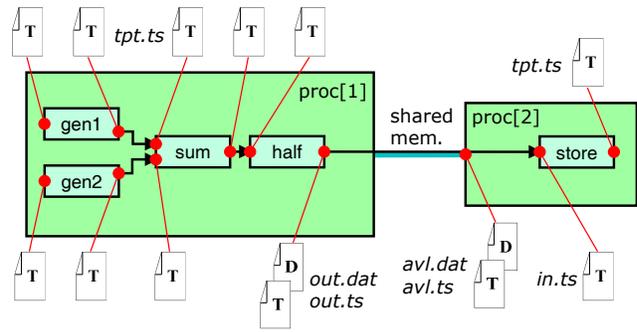

Figure 5. A 2-core mapping of `test1`

X-Sim analyzes the dependencies between different resources in a streaming application mapping and then runs the simulations for each part of the system in the correct dependency order. For example, consider the 2-core mapping of `test1` in Figure 5. In this mapping, X-Sim runs in three stages. In the first stage, X-Sim simulates `proc[1]` to produce the data (D) and timing (T) traces collected at the points shown in the left half of Figure 5.

The data trace file (D) records all the data leaving the `half` block. The corresponding nearby timing trace file (T) records the times at which this data leaves `proc[1]`. Other timing trace files record the times at which user-inserted *testpoints* were triggered. Such testpoints allow the user to capture the processor clock (using an `rdtsc` system call) at arbitrary points in the implementation of a block mapped to a core. For example, the timing trace files on the very left of the figure record the times at which the `gen1` and `gen2` blocks started generating each number.

In the next simulation step, a communication delay model is applied to output timestamps from `proc[1]` to generate availability timestamps for `proc[2]`. These timestamps indicate the times at which data was available to the block(s) mapped to `proc[2]`. X-Sim has the ability to apply arbitrary communication delay models, including constant delay models, effective bandwidth models, and

distribution based models.

In the final simulation step for this mapping, the simulation for `proc[2]` is run using the availability data and timing traces generated in the previous steps. In this step, input timing traces (in the middle) are generated to indicate the time that data was input into the core, while testpoint timing traces (on the right) indicate the time that processing of a data element finished.

Note that this resource-by-resource (e.g. `proc[1]` then `proc[2]`) simulation of the application explicitly prevents cyclic mappings from being simulated. For example, consider a mapping of the example application where the `half` block is mapped to `proc[2]` and all other blocks are mapped to `proc[1]`. This mapping can not be simulated because there is a cyclical data dependency between the processors. Future versions of X-Sim will support the ability to run multiple platform simulators in parallel, and thus support the simulation of cyclical mappings.

At the end of simulation, a comprehensive history of all data and timing traces is available for analysis. For example, the distribution of total times spent in various blocks in the `test1` application (Figure 6 shows times for the 1-core mapping) indicates that roughly half of the entire time in this application is spent in the `store` block. A speedup of about $2\times$ can be expected between a 1-core mapping (Figure 2) and a 2-core mapping (Figure 5) where the `store` block has been moved to a separate core.
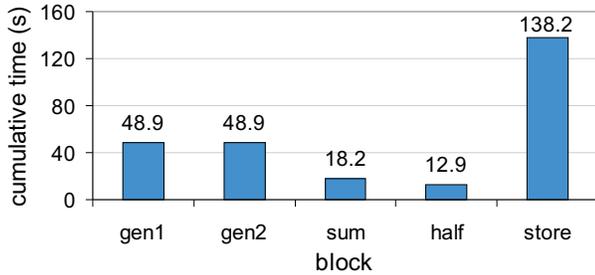

Figure 6. Time spent in `test1` **blocks**

Simulations of 1-core and 2-core mappings of `test1` show roughly this result, as can be seen in Figure 7. Throughout the paper, a constant, zero-cost communication model was used to represent the shared memory interconnect resource. This figure also shows the execution times gathered from actual deployments of the `test1` application. The 1-core native execution simulation and deployment are identical. In the 2-core mapping, the two differences in the deployment run from the simulation are that: 1) the communication mechanism is physical shared memory rather than a zero-delay model, and 2) two cores are used to execute the application in parallel. Recall that in the simulation, each core execution is performed in a separate
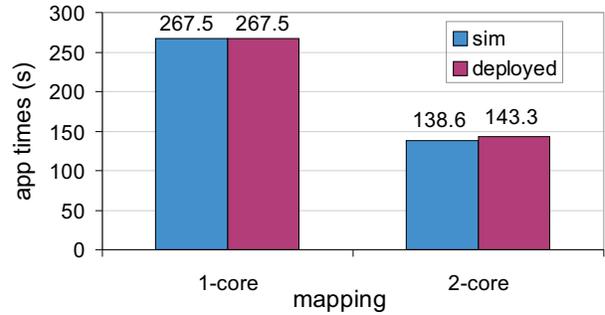

Figure 7. Application run times for `test1`

simulation step while recording the application execution state in trace files. From the graph in Figure 7, we can see that the 2-core deployment result closely matches X-Sim's simulated prediction for the simple `test1` example application.

4. VERITAS Example Application

We now focus our attention on results gathered from simulation and deployment runs of an astrophysics application. The VERITAS experiment [15] is an astrophysical gamma ray detection experiment with high data throughput requirements. Due to its streaming and highly parallel nature, it easily fits into Auto-Pipe's programming model.

Gamma rays are produced by extraterrestrial sources such as pulsars, supernovae, neutron star collisions, and super-massive black holes in galactic nuclei. When these rays strike the atmosphere, they result in photon showers called Cherenkov radiation.
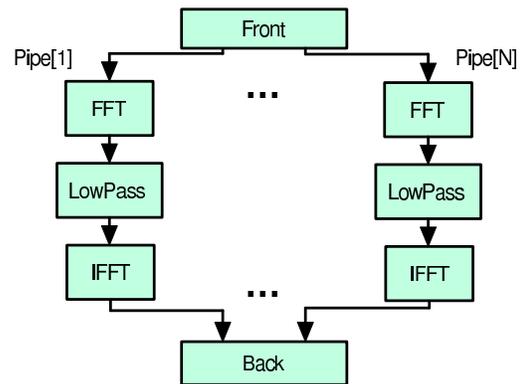

Figure 8. Simplified VERITAS dataflow

In the VERITAS experimental setup, this radiation is captured by arrays of hundreds of photomultiplier tubes. Analog to digital converters then generate a large amount
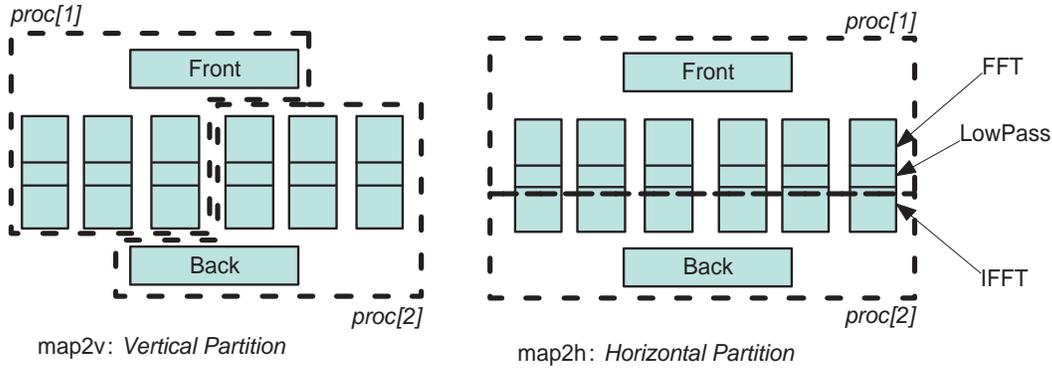
Figure 9. Vertical and Horizontal 2-Core Mappings

of raw pixel data, which subsequently undergoes computationally intensive digital signal processing. This processing consists of steps to clean up the signal data, to analyze it, and finally to determine the origin (in space) of the gamma ray source. Figure 8 shows a highly simplified view of the VERITAS application.

The `Front` section reads raw pixel data from a database and distributes it to N parallel pipes where the bulk of the computationally intensive digital signal processing is performed. Data from the pipes is merged into the `Back` section, which combines the processed pixel data.

Processing a sample set of 5000 gamma ray *events* on a single processor core required 127.2 seconds. Note that the actual physical system used to test out deployments was an AMD Athlon 64 X2 4400+ system with four cores, 1MB L2 cache and 8GB of system memory. Since scientists would like operate on inputs derived from millions of events, there is a need to speed up the application. Profiling the application shows that about 95% of the processing time is spent in the pipes of Figure 8. Within each pipe, the `LowPass` block takes up about 13% of the processing time, with the rest of the processing roughly evenly divided between the `FFT` and `IFFT` blocks. In this section, we will show how X-Sim can be used to explore alternative multicore mappings of the VERITAS application.

First consider the problem of mapping the VERITAS application to the same two-processor multicore system earlier used in the `test1` example. Consider a small system consisting of six pipes. Figure 9 shows two approaches to mapping VERITAS to two cores, one vertical (map2v) and one horizontal (map2h). In the vertical mapping, the `Front` section and the three `Pipe` blocks on the left are mapped to one processor, while the `Back` section and the three `Pipe` blocks on the right are mapped to the other processor. In the horizontal mapping, everything feeding the `IFFT` blocks is mapped to one processor (i.e., everything up to and including the `LowPass` blocks), while the `IFFT` and downstream

blocks are mapped to the other processor. Note that the vertical mapping is a relatively balanced mapping, whereas the horizontal mapping is somewhat imbalanced with a higher processing load placed on `proc[1]`.
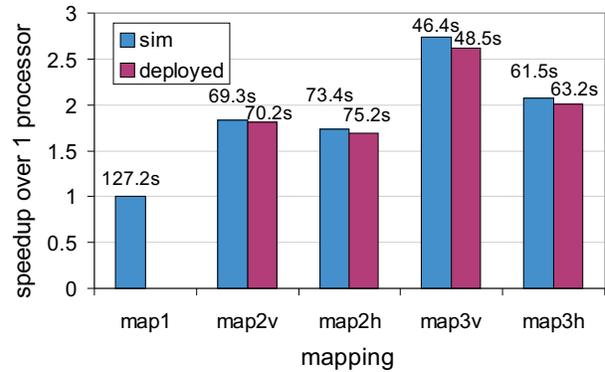

Figure 10. VERITAS speedups and runtimes on 2 and 3 cores

Figure 10 shows the results in terms of the application speedup relative to the single processor case. The bars labeled `map2v` and `map2h` show the total application run time measured from X-Sim simulations as well as deployments of the two mappings. Also shown in this figure is the running time for the 1-core mapping as well as 3-core mappings that are described later. There are two main things to note here. The first is that the simulation times are within 5% of the corresponding times measured on the multicore deployed application. Also of note is that the vertical and horizontal mappings are relatively close to each other, with times of 70.2 s and 75.2 s. The vertical 2-core mapping is slightly faster than the horizontal 2-core mapping, since it is a more balanced partitioning of the processing costs. Speedups (1.8× for `map2v` and 1.7× for `map2h` for the
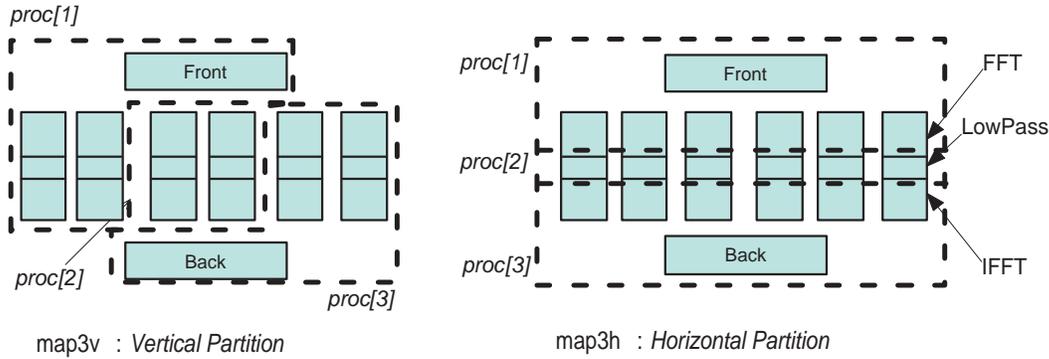
Figure 11. Vertical and Horizontal 3-Core Mappings

deployment runs) over the 1-core mapping are shown on the y-axis.

The next mapping problem was to run the VERITAS application on a 3-core system. Once again, two mappings were evaluated, one vertical and one horizontal. These mappings are shown in Figure 11. In the vertical mapping, two `Pipe` blocks are mapped to each processor, while the `Front` block is additionally mapped to `proc[1]` and the `Back` block is additionally mapped to `proc[3]`. In the horizontal 3-core mapping, the `LowPass` blocks are the only things mapped to `proc[2]`. Everything upstream of the `LowPass` blocks is mapped to `proc[1]`, while everything downstream is mapped to `proc[3]`.

The vertical mapping is a relatively balanced partitioning. The horizontal mapping, however, is not very balanced. This is because the part (`LowPass`) of the `Pipe` blocks mapped to the second processor is not as significant a portion of total processing as the `FFT` and `IFFT` blocks. Results from the 3-core mappings are shown on the bars labeled `map3v` and `map3h` in Figure 10. Once again the times from running the X-Sim simulations of the two mappings are within 5% of the times gathered from running the deployed mappings.

The vertical 3-core mapping shows a $2.6\times$ speedup over the 1-core mapping, while the horizontal 3-core mapping only shows a $2.0\times$ speedup. This difference can be attributed to the difference in balancing the processing load between the two mappings.

In the final experiment, X-Sim simulation results from mapping the VERITAS application to 1, 2, 3, 4, 8, and 16-core systems are considered. Earlier results indicated that a vertical partitioning of the VERITAS blocks resulted in relatively balanced distribution of the processing load. Thus, we only consider that mapping approach and allocate the `Pipe` blocks evenly among the processor resources. For example, the 4-core mapping has four `Pipe` blocks mapped to each processor. For the 3-core mapping, a 15-`Pipe` version of the application was used with five `Pipe` blocks mapped

to each processor.

The results for simulation runs for each of the multicore mappings are shown in Figure 12. Also shown in this figure are the deployed application run times for the 1 through 4-core mappings. Additionally, the leftmost bar in each grouping shows the ideal speedups (equal to the number of cores) that would be obtained for a perfectly balanced partitioning.
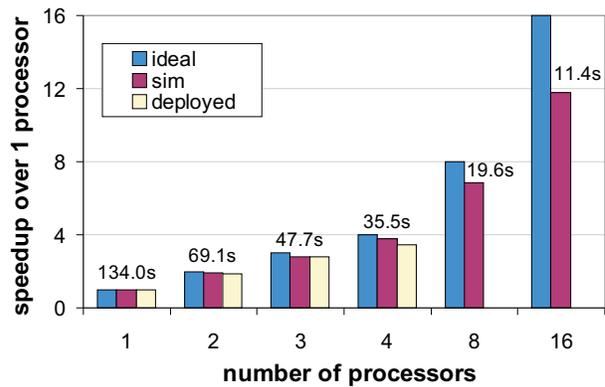

Figure 12. VERITAS performance scaling with number of processors

The graph shows that X-Sim predicts times that roughly follow the ideal speedups, with the difference from the ideal increasing with the number of processors. The cause for this is that the `Front` and `Back` sections are always mapped to the first and last processors, and only the `Pipe` section processing is evenly distributed among the different processors. Say the processing on `Front` takes f seconds, while the processing on a `Pipe` takes p seconds. For the 8-core mapping, `proc[1]` has the `Front` section and 2 `Pipes` mapped to it (and takes $f + 2p$ seconds). For the 16-core mapping, `proc[1]` has the `Front`

section and 1 `Pipe` mapped to it (and takes $f + p$ seconds). The speedup (for `proc[1]`) from 8-core to 16-core would thus be $(f + 2p)/(f + p)$. Going a step back, the speedup from 4-core to 8-core can similarly be calculated to be $(f + 4p)/(f + 2p)$. Thus, the speedup from 8-core to 16-core is less than the speedup from 4-core to 8-core. Another way of looking at this is that, as more processors are available, on the first and last processor, `Front` and `Back` computation times become a higher percentage of the overall processing time for the first and last cores.

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

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

5. Conclusions

This paper considers the use of multicore systems in streaming applications. The development tool Auto-Pipe and its simulation component X-Sim are presented and applied to a streaming application, VERITAS, where gamma ray data is sent from a telescope array through various converters, and then to the multicore system for further processing. The paper illustrates how the Auto-Pipe system can be used to evaluate candidate mappings of the streaming algorithm to available multicore systems. Various mappings are considered for systems ranging from 1 to 4 cores, and relative performance is obtained from both simulated and measured (deployed) execution of the algorithm. Simulation results using the optimal mapping is obtained for multicore systems containing 8 and 16 processors. From this, speedup as a function of number of processors is obtained. A number of results are presented including non-ideal scaling with the number of cores due to imbalanced mapping.

The Auto-Pipe system continues to be enhanced and is currently being used in heterogeneous system design activities involving both multicores and FPGAs.

References

[1] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine stream architecture. In *Proc. 31st Annual Int'l Symposium on Computer Architecture*, pages 14–25, 2004.

[2] R. D. Chamberlain, E. J. Tyson, S. Gayen, M. A. Franklin, J. Buhler, P. Crowley, and J. Buckley. Application development on hybrid systems. In *Proc. of Supercomputing*, Nov. 2007.

[3] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[4] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proc. of Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 33–42, Sept. 2006.

[5] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[6] S. Gayen. X-Sim and X-Eval: Tools for simulation and analysis of heterogeneous pipelined architectures. Master's thesis, Washington University in St. Louis, Department of Computer Science and Engineering, 2008.

[7] S. Gayen, E. J. Tyson, M. A. Franklin, and R. D. Chamberlain. A federated simulation environment for hybrid systems. In *Proc. of 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, pages 198–207, June 2007.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 2nd edition, 1999.

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

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

[11] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: a language for streaming applications. In *Proc. Int'l Conf. on Compiler Construction*, Apr. 2002.

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

[13] E. J. Tyson. Auto-Pipe and the X language: A toolset and language for the simulation, analysis, and synthesis of heterogeneous pipelined architectures. Master's thesis, Washington University in St. Louis, Department of Computer Science and Engineering, 2006.

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

[15] 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.