Implementation of a Streaming Parallel Model for the TRIPS Architecture

by

Mark Gebhart

Undergraduate Honors Thesis

Presented to the Faculty of the

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Bachelor of Sciences

The University of Texas at Austin

May 2006

Implementation of a Streaming Parallel Model for the TRIPS Architecture

Approved by
Supervising Committee:

_____

_____

Acknowledgments

I would like to thank the following people:

- My supervisor Dr. Stephen Keckler for his guidance over the last two years
- Saurabh Drolia, Sadia Sharif, Paul Gratz, and the TRIPS team members for their help throughout this project.

Mark Gebhart

The University of Texas at Austin
May 2006

Implementation of a Streaming Parallel Model for the TRIPS Architecture

Mark Gebhart, B.S.
The University of Texas at Austin, 2006

Supervisor: Stephen W. Keckler

One common method of improving performance is to use a collection of processors to execute a large scale program in parallel. The Streaming Virtual Machine (SVM) model of computation is one method of extracting concurrency from traditional programs. In an SVM program tasks are partitioned into kernels that consume and produce streams of data rather than traditional operations that consume and produce single data values. This streaming model is particularly advantageous on architectures such as TRIPS that contain a Direct Memory Access (DMA) controller that allows for the overlapping of communication and computation. The author developed several parallel benchmarks along with enhancements to the TRIPS simulators to evaluate the viability of the SVM model on the TRIPS system. Initial result show that the SVM model holds promise in several domains

and a prototype system of four processors achieves an average speedup of 3.70 times over uniprocessor sequential execution on a collection of computationally bound benchmarks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One common approach to reducing a program's execution time is to execute the program in parallel on a collection of processors. In order to accomplish this a programmer must have a well defined method of portioning work among this collection of processors. The Streaming Virtual Machine (SVM) model [12] is one method of extracting concurrency from traditional programs.

The SVM model employs three main principles to allow traditional programs to execute efficiently on a collection of processors. First, SVM programs express data parallelism across multiprocessors. This is accomplished by partitioning the dataset and assigning a portion of the total data to each processor. Each processor then performs the same action of its portion of the total data. Secondly, the SVM model has better locality behavior due to its streaming nature. In a streaming program, operations consume input streams of data and produce output streams of data. By operating at a higher level of granularity than single data items the model can reduce the overhead of data access. Further, processors only operate on data in their own local memory, avoiding the long latency associated with main memory. Figure 1.1 shows the conceptual operation of the SVM system. Finally, the SVM model overlaps communication and computation by using DMA controllers to perform data

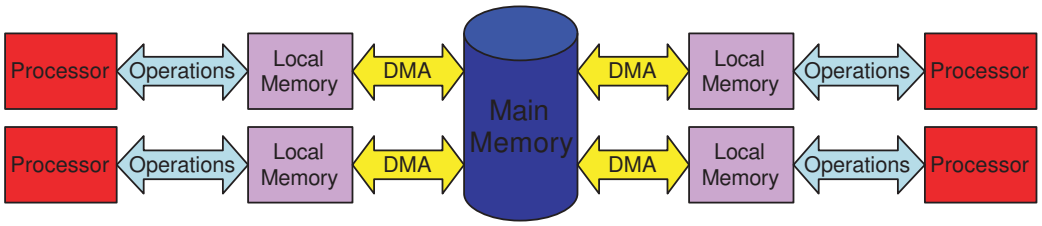

Figure 1.1: High Level Overview of SVM System

transfers while the processor continues to perform useful work.

The SVM model also seeks to reduce a programmer's burden when developing parallel applications. To accomplish this a two-staged compilation approach is used where a machine independent High Level Compiler (HLC) is used to detect and exploit parallelism from a program written in a high level language. Figure 1.2 shows the process of developing SVM applications. Along with the program the HLC is also given a machine model [11] that describes the system that the program will run on. This machine model contains details such as the number of processors, the size of local memory, and other machine specific details. The output of the HLC is standard C that makes calls to the SVM Application Programming Interface (API) to express parallelism. A Low Level Compiler (LLC), a machine's standard C compiler, then compiles the SVM application to native code for the target architecture. The Morphware Forum [6] developed the SVM standard with the target architectures of TRIPS, Monarch, RAW, and Smart Memories [17, 16, 9, 10]. This approach allows the machine independent HLC to handle the difficult task of extracting parallelism and then all participants can use the results.

We evaluate the SVM model on the TRIPS system, which is a collection of up to sixty-four TRIPS processors. The TRIPS system uses a shared memory approach where a global physical address spaces allows interprocessor communication. Each TRIPS processor contains local memory that behaves as a software controlled

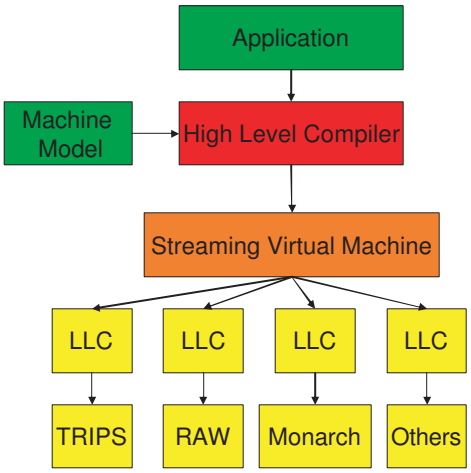

Figure 1.2: SVM Application Development

cache and a DMA controller that make it a good match for the SVM model of computation. In order to operate on data a TRIPS processor uses it DMA to pull data from main memory into its local memory and then once computation is complete returns the result to main memory using the DMA. The TRIPS processor is the first implementation of an EDGE ISA that seeks to scale future performance by explicitly encode instruction dependencies in the ISA [1]. A grid of replicated execution units allows for high IPC and allows the TRIPS processor to perform well on applications with high ILP.

This thesis describes the method and results of evaluating the viability of the SVM model on the TRIPS system. To quantify the performance of the SVM model we enhanced the multiprocessor functional simulator to generate an estimate of the cycle cost of an SVM program. We used a functional functional simulator because the large size of the SVM programs made running a cycle-accurate simulation impractical. To test the precision of the cycle estimation we ran a suite of benchmarks and found the estimates to be within 20.34% of the true values. These benchmarks have highlighted areas where future work could improve the model, such as a more

accurate model of the memory system and modeling of the branch predictor that will serve as future work.

To test the performance of the SVM model on TRIPS the author developed a suite of benchmarks from the scientific computing and digital signal processing domains with a collection of both computationally and memory bound algorithms. The modified simulator ran these benchmarks using a prototype system of 4 TRIPS processors and produced performance results. On average the SVM programs outperformed their sequential versions by a factor of 2.95. Further, the average speedup for the computationally bound algorithms was 3.70. These results show that the SVM model holds promise especially for computationally intensive applications and when the actual hardware completes fabrication later this year we can obtain a more accurate measure of overall system performance.

Chapter 2

Background

2.1 SVM Overview

The Streaming Virtual Machine (SVM) is a model for parallel execution that the Morphware Forum developed as part of the PCA project [2]. An SVM program divides the work into kernels that perform a specific task on input streams of data and produce output streams of data. The size of these streams is arbitrary but in our benchmarks is typically on the order of thousands of elements per stream. Under the SVM model one of the processor serves the role of Master and is responsible for initializing the shared structures at startup time along with the initial assignment of work to other processors. The SVM models addresses three main issues:

Programmer Burden: One of the obstacle for parallel programs is that the programmer is responsible for determining the parallelism within a program and writing the program to extract this parallelism. In order to move this responsibility away from the programmer the SVM model introduces a High-Level Compiler (HLC) which is responsible for taking a program that is written in a high level language such as C and translating it to a parallel SVM program. Figure 2.1 shows the

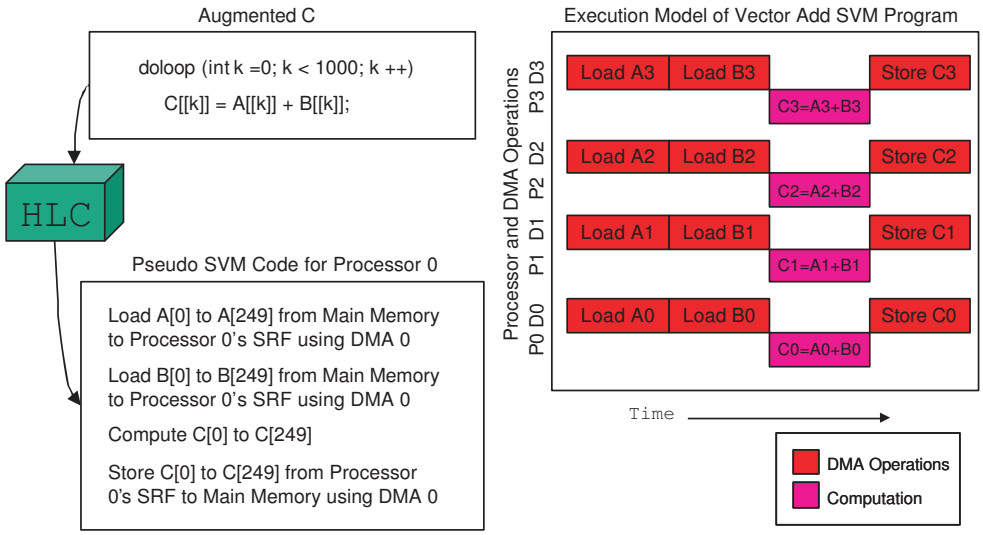

Figure 2.1: Vector Add Compilation

process of SVM application development on the TRIPS system for a simple vector add program. First, a programmer writes the vector add program in Augmented C, the standard C programming language with additional language constructs to allow the HLC to extract parallelism. These additional constructs include a doloop structure instead of a for loop and double array indices to indicate to the HLC there is no aliasing present in the arrays. Then this vector add program is provided to the HLC along with the TRIPS machine model that contains around 200 system specific parameters such as the number of processors, the size of local memory, etc. The output of the HLC compiler is standard C with calls to the SVM API to express the parallelism. For the vector add example, we see that first a portion of the A and B arrays are loaded into the processor's memory and then the processor performs the element-wise addition. Finally, the program copies the result back to main memory using a DMA controller. The final image in Figure 2.1 shows how the various operations map to the TRIPS processors and DMAs. While this example

Function	Description	Blocking
svm_kernelInit	Initializes a kernel object for use	No
svm_kernelAddDependence	Enforces the order of kernel execution	No
svm_kernelRun	Executes a kernel once all dependences have executed	**Yes**

Table 2.1: SVM Kernel API

Function	Description	Blocking
svm_blockInit	Initializes a memory object for use	No
svm_moveB2B	Transfers data from one contiguous area to another	No
svm_stridedScatterB2B	Transfers contiguous data to a destination using strided stores	No
svm_stridedGatherB2B	Transfers data in strides to a contiguous destination	No

Table 2.2: SVM Block API

shows no overlap between the processor and DMA operations, software pipelining can be used to overlap the processor and DMA operations to improve performance.

Division of Work: The SVM API has two major objects for controlling execution. First, svm_kernels control actions that need to be executed such as computation or DMA transfers. The API for SVM Kernels is shown in Table 2.1. The most important functions of the API is the ability to add dependences between kernels and to execute kernels. The svm_kernelAddDependence call signals that one kernel must not start executing until its dependent kernels complete. The svm_kernelRun call is used to start the execution of a kernel. This is a blocking call that first waits for all of a kernel's dependencies to finish executing and then executes the kernel. Using these kernel calls, a SVM program explicitly encodes the parallelism in a program for the hardware to execute. Table 2.2 shows the API for the other main SVM object, a svm_block. A svm_block represents a piece of memory that stores

svm_blockInit(A, size, location)
svm_blockInit(B, size, location)
svm_blockInit(C, size, location)
svm_moveB2B(main_memory, A)
svm_moveB2B(main_memory, B)
svm_addDependence(compute_kernel,move_kernel)
svm_kernelRun(compute_kernel, A, B, C)
svm_kernelAddDependence(move_kernel, compute_kernel)
svm_moveB2B(C, main_memory)

Table 2.3: Vector Add SVM Program for 1 Processor

application data. This data either resides in a processor's local memory or in the global main memory. Through the SVM API a programmer can control transfers from one block to another such as when data is transferred from main memory to a processor's local memory. These block move operations contain an svm_kernel object that allow dependencies to be expressed and control execution. The most powerful block transfer operations are the stridedScatter and stridedGather move operations. These operations are useful when extracting sub-blocks from a matrix which is common when operating on large matrices using a blocked algorithm. The memory move operations are nonblocking so the processor can continue to perform useful work while the DMA is transferring data. An example of how the SVM API is used to encode the previously discussed vector add program for one processor is shown in Figure 2.3. Each processor would run a version of this same code with the only difference being the portion of the data that it operates on. This allows the SVM program to exploit data parallelism across the processors. First, the programmer defines three blocks of memory to store the A, B, and C arrays. Next, the A and B arrays are moved into local memory. Then, the computation can begin as soon as the move operations are complete. Finally, the DMA moves the result back to main memory.

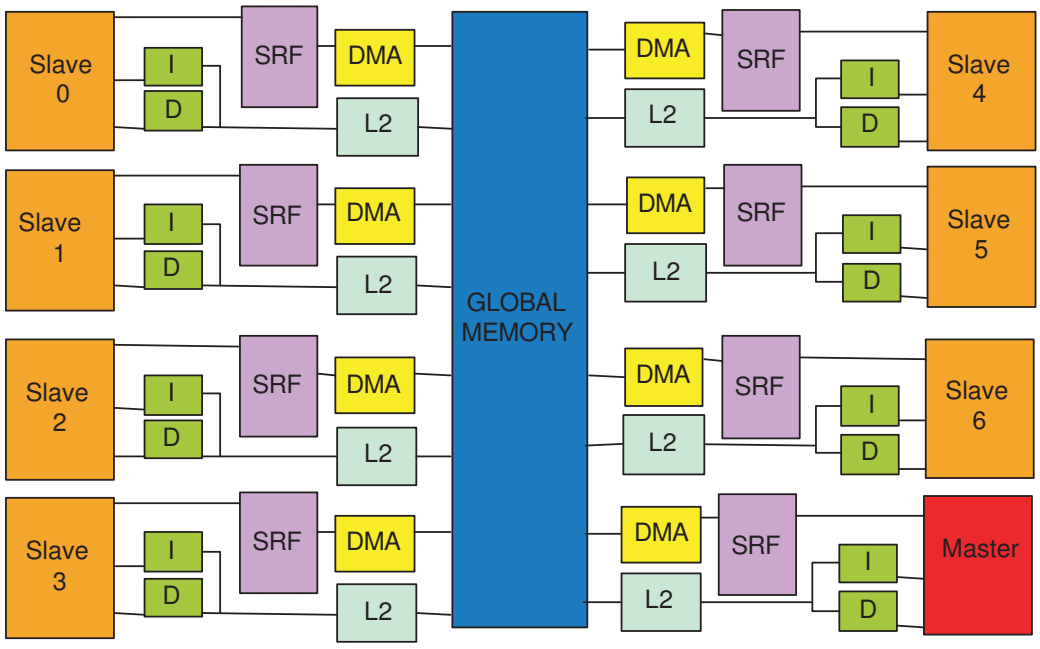

Figure 2.2: TRIPS System

Main Memory Access Latency: To mitigate the latency of a main memory access, processors using the SVM model first use their DMAs to bring data into their local memory and then perform operations on the data there. On the TRIPS processor a memory request to the local memory takes approximately 10 - 15 cycles while a request to main memory takes several hundred cycles. As long as the cost to execute a work kernel is greater than the cost of the DMA transfer that moves the data to and from local memory from the processors point of view there is no cost for moving the data. This is because the processor can perform useful work while the DMA is transferring data. This is true in general for algorithms with a computation complexity greater than linear, the cost of data transfer.

2.2 TRIPS System Overview

The TRIPS SVM system can be configured with up to sixty-four processors. All of these processors are homogeneous and the choice of which processor to perform the master duties is arbitrary. The master is responsible for initializing shared structures such as a mutex, message queue, and processor id mappings and initial work assignment. A block diagram of an 8 processor system, corresponding to a single TRIPS board is shown in Figure 2.2. Each processor has its own private L1 instruction and data cache and the L2 is distributed with a portion of the L2 on each processor. A static mapping from addresses to L2 assignments ensures that a data item can only reside in 1 of the L2 caches. For data, such as control structures that is shared among processors we mark that shared segment as L1 un-cacheable to avoid coherency issues. A later optimization would be to allow L1 caching and ensure that the L1 is flushed on writes to the shared segment or when a processor is polling on a shared variable.

DMA Engine Each TRIPS processor contains an on-chip DMA engine that allows for the overlapping of computation and communication. As the processor-memory gap continues to widen it becomes more important to mitigate the long delays associated with main memory access; the TRIPS DMA engine allows the processor to perform useful work while the DMA fetches data. This DMA is capable of performing contiguous, strided gather, or strided scatter operations. The strided operations are useful to extract sub-blocks of a matrix [4]. Because the DMA operates in terms of physical addresses, the virtual addresses must be converted to physical addresses with a system call before the DMA is programmed. The cost of doing the system call will be high so the actual hardware will only make the translation once per segment and calculate all subsequent physical addresses with an offset to the known base physical address. In addition to being able to transfer

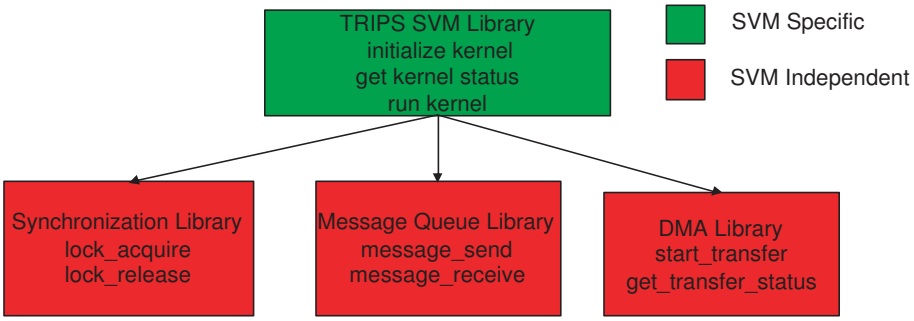

Figure 2.3: SVM Library Implementation

memory from global memory to a processor's local memory the DMA has the ability to transfer data from one processor's local memory to another processor's local memory which is useful for pipelined applications where the processors operate on data in succession.

Streaming Register File (SRF) Each TRIPS processor contains 8 64KB memory tiles which compose a portion of the global L2 cache. Any number of these memory tiles can be dynamically reconfigured to serve as a SRF. The SRF behaves like a software controlled cache where transfers into and out of the SRF are explicitly controlled by the application. For the study, 4 tiles per processor are used as a SRF and the other 4 tiles remain in use as L2. This provides each processor with 256 KB of local memory.

2.3 SVM Library Implementation

A TRIPS specific SVM library is used to implement the SVM API. Rather than a single monolithic library we developed four disjoint libraries that facilitated debugging and code reuse as shown in Figure 2.3.

Synchronization Library: Contains the basic synchronization objects such as a mutex that multiprocessor programs can use to coordinate their actions. The author developed this library based on a technical report that proposed synchronization primitives for the TRIPS architecture [13]. The synchronization objects are built on top of an atomic lock instruction that is part of the TRIPS ISA. Additionally, special block flags indicate to the hardware that blocks that try to access a shared lock need to be the only block running on the processor at the time the lock instruction is issued.

The following libraries were originally developed by Saurabh Drolia [3].

Message Queue Library: Allows processors to send messages to each other using the shared memory segment. This library is used by the SVM system to allow the master processor to assign work functions to the slave processors.

DMA Library: Presents an interface to applications to allow them to program the DMA controller. This library is used by the SVM system to initiate transfers of data streams.

TRIPS SVM Library: This library is the TRIPS specific implementation of the SVM API. It is built using primitives from the other three libraries.

The modular library design has proven effective for debugging and has allowed the majority of the code in the three support libraries to be reused for a current project to implement the Message Passing Interface (MPI) parallel model [8].

Chapter 3

Simulator Enhancements

When simulating a uniprocessor program for the TRIPS architecture a programmer currently has two choices. Either they can run their program on tsim_proc, a cycle-accurate simulator that fully models all of the actions of the processor, or they can run their program on tsim_arch, a functional simulator and does not provide performance metrics such as cycle counts. On a modern workstation the cycle accurate simulator can simulate one thousand instructions per second while the functional simulator can execute one million instructions per second.

The simulation time to run a large SVM program on the cycle accurate simulator is prohibitively expensive. For example it would take approximately four months to run the matrix multiplication SVM program on the cycle accurate simulator. Therefore the system simulator was constructed by using a collection of functional simulator instances, with each instance modeling one processor in the SVM program. The system simulator also models the inter-processor communication via the shared memory system. The use of the functional simulators limited performance metrics to extremely course statistics. In order to more closely model the viability of the SVM model, we modified the system simulator and the uniprocessor simulator to produce an estimate of the cycle cost of an SVM program.

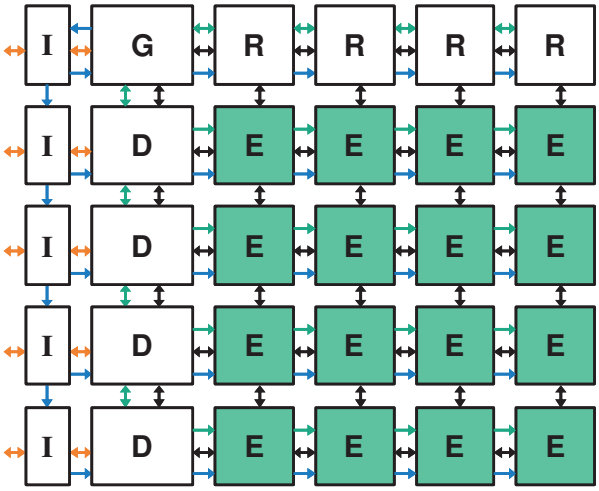

Figure 3.1: TRIPS Processor

3.1 TRIPS Execution Model

The SVM model views the TRIPS processor as a black box, but to explain the author's simulator enhancements the TRIPS execution model is presented. The TRIPS processor does not operate on an instruction granularity like conventional processors. Instead the atomic unit of execution is a block of up to 128 instructions. The functional simulator only reports the number of blocks executed per processor. Each block can contain up to 32 register reads, writes, and memory operations [19]. Each processor is capable of executing up to 8 blocks simultaneously. These instruction blocks are generated by the Scale compiler [14] and each instruction is statically assigned to one of 16 ALUs which are arranged in a two-dimensional mesh topology on the processor. Rather than returning intermediate results to the register file producing instructions pass their results directly to consuming instructions via an on-chip network that connects the execution units [7]. These instruction dependencies are explicitly encoded by the EDGE ISA. Figure 3.1 shows the tiles that make up 1 TRIPS processor. The tiles are as follows: I, instruction cache, D, data cache,

G, control, R, register file, and E, execution tile. Once the block starts executing each instruction fires dynamically as soon as its operands are available [15]. This combination of static placement and dynamic issue makes the TRIPS architecture a novel design.

3.2 Uniprocessor Simulator Enhancements

Along with being able to provide cycle estimates for SVM programs, the modified uniprocessor simulator can be used by other developers wishing to obtain approximate cycle times for long running sequential programs. This will be particularly useful when optimizing programs to obtain a estimate of the improvements of different optimizations without having to wait for the results of the cycle accurate simulator. To facilitate development, the simulator analysis code began extremely naive and was gradually enhanced with various factors to improve the model's precision. The various factors that were initially considered by the analysis code were:

- Static Instruction Cost: By examining the opcode a static cost for an instruction can be assigned. For example, the cost of executing a floating point instruction will be higher than the cost of an integer instruction.

- Static Routing Delay: When one instruction produces a value and passes this value through the on-chip network to a consuming instruction the cost of this transfer is 1 cycle per hop. We calculate the number of hops between the producer and consumer and factor this cost into the consuming instruction. Additionally, we consider the static cost of routing from the execution tiles to the register file for register reads and writes and the cost of routing to data tiles for memory operations.

Using this analysis we determine the critical path through the block and assign the block a cycle cost equal to the critical path plus a block fetch and commit

delay. This analysis provided a very coarse estimate of cycle cost but it become apparent after testing that a more precise model would be needed.

3.2.1 Advanced Modeling

Modeling Multi-Block Execution: In order to correctly model how the TRIPS processor would execute multiple blocks at the same time one must track the critical path through the entire program. The only way that instructions in a block depend on instructions from other blocks is if there is a producer/consumer relationship through the register file or memory. To account for this, we track for each register the cycle time when the last write occurred; whenever a read occurs, it must be delayed until after the last write to that register occurred. A similar approach is taken with regards to memory. While the simulator models multi-block execution, it executes the blocks one at a time and then looks backwards in time to see how blocks would have overlapped. This greatly simplified the implementation since modeling all eight blocks executing at the same time would have introduced numerous complications and slowed simulation time.

The following two enhancements were implemented by Paul Gratz.

Cache Modeling: An L1 data and instruction cache model that is used to determine if a load will hit or miss in the L1 cache. Since the difference in latency for an L1 hit versus a L2 hit is roughly 100%, this further improved the precision of the model. The instruction cache model is used to determine the fetch delay of the block. On an i-cache miss an extra delay of 20 to 30 cycles is added to the block fetch making this metric important to consider.

Load Store Dependence Prediction: When a load executes before all prior stores have executed, the TRIPS processor uses a dependence predictor to predict whether or not the load depends on the store. As long as the load does not depend

on the store, it is legal to execute the load, and thus all instructions that depend on the load can begin executing. However, if the load does depend on the store the load must be delayed until the store executes [18]. The two cases that are important to model are independent loads that are predicted to be dependent and dependent loads that are predicted to be independent. In the former case the load is delayed until after all prior stores have committed. This can be a severe penalty so it is important that the predictor be highly accurate. In the later case the load will receive invalid data causing the block to generate an exception. This exception flushes all ongoing work and re-executes the block ensuring that the load executes after the store. The inter-block analysis that examines producer and consumer relationships through memory operations correctly handle the remaining cases.

3.3 Simulator Simplifications

The current simulator makes several simplifications when it is calculating cycle estimates.

- Memory System: The current cycle accurate simulator assumes a perfect L2 cache. Our current modified simulator makes this same assumption. Since one of the main goals of the SVM model is to mitigate main memory latency this simplification masks any gains we would receive by using the streaming model. Future work could extend the simulator to model the L2 cache and thus model main memory latency.

- Network Contention: The analysis code that calculates the routing delays through the On-Chip Network (OCN) only considers the static cost based on the producer and consumer's location on the chip. However, in the real hardware only one value can be sent on a given link per cycle so link contention can add a dynamic delay to an instruction's routing cost.

- Branch Prediction: While currently we assume a perfect branch predictor we plan to add modeling of the branch predictor. On the EEMBC benchmarks we found the branch predictor to be accurate around 80% of the time. This presents an opportunity to increase the precision by modeling the branch predictor.

Since these simplifications are optimistic, the estimated cycle time that we generate represents the best case scenario and the actual cycle number can be expected to be higher than our estimated cycle count.

3.4 Simulation Precision

To evaluate the precision of the modified simulator, a suite of benchmarks from the standard EEMBC benchmark suite was used. All of the available benchmarks that the TRIPS compiler would compile successfully were used. Table 3.1 shows the results of comparing the actual cycle counts, obtained with the cycle accurate simulator, with the estimated number of cycles, obtained with our modified simulator. We see that there are two benchmarks where our estimate is pessimistic compared with the cycle accurate simulator. These cases are being investigated and most likely can be attributed to a bug in analysis code. On average our estimate proved to be within 20.34% of the true value.

The analysis code for the simulator was written primarily for correctness. Over the next several months as its use get more widespread it will be tuned for performance and additional analysis may be added to increase precision.

3.5 Simulation Speed

Because the point of this work was to create a fast but reasonably accurate simulator, the cost of the added analysis code was closely monitored. If the entire processor is

Benchmark	**Error**
automotive/a2time01	15.97%
automotive/aifftr01	10.21%
automotive/aifirf01	12.93%
automotive/aiifft01	7.51%
automotive/basefp01	3.25%
automotive/bitmnp01	-13.49%
automotive/cacheb01	87.04%
automotive/canrdr01	25.17%
automotive/idctrn01	11.87%
automotive/iirflt01	-1.96%
automotive/matrix01	14.86%
automotive/pntrch01	5.50%
automotive/puwmod01	26.70%
automotive/rspeed01	23.00%
automotive/tblook01	19.96%
automotive/ttsprk01	23.69%
networking/ospf	18.21%
networking/routelookup	24.65%
office/dither01	20.01%
telecom/autcor00	17.58%
telecom/conven00	4.38%
telecom/fbital00	24.63%
telecom/fft00	24.13%
telecom/viterb00	51.49%
Average	20.34%

Table 3.1: Cycle Estimation Error for EEMBC Benchmarks

modeled in the functional simulator we will have simply recoded the cycle accurate simulator. However, all of the analysis only introduced a slowdown of only 20%. This makes the modified simulator still roughly 800 times faster than the cycle accurate simulator and practical for running large SVM programs.

3.6 System Simulator Modifications

The system simulator initially operated on a block granularity where each processor would execute one block in a round robin fashion. However, to correctly model how applications would execute on hardware, the system simulator was enhanced to operate on a cycle granularity where each processor would execute a block, receive the estimate cycle cost of that block from the uniprocessor simulator, and then delay the execution of the next block based on this estimate. The system simulator also makes an estimate of the cycle cost of executing a DMA transfer based on the number of bytes transferred and ensures that the processor does not view the DMA transfer as finished until the specified number of cycles has passed. The system simulator was also enhanced to produce several metrics such as the number of times a blocks was executed, the average cycle cost of each block, and the DMA utilization.

Chapter 4

Application Development

4.1 Benchmark Suite Overview

In order to test the viability of the SVM model on the TRIPS systems, the author developed a suite of benchmarks. This suite is composed of benchmarks from the scientific computing and digital signal processing domains where inherent parallelism is common.

4.1.1 Computationally Bound

On applications that are computationally bound, the processors operate for extended periods of time without interactions with the SVM library so the overhead of establishing the SVM model can be easily amortized.

- Matrix Multiplication: The matrix multiplication algorithm, common in scientific computing, consisted of the multiplication of two 1040 by 1040 element matrices. The matrices were composed of double precision data so floating point operations were necessary. A blocked algorithm was used so that entire sub-blocks could fit into a processor's local memory. Further, the application was constructed so that the computation and communication could be

overlapped [5].

- Fast Convolution: Fast Convolution is an application from Digital Signal Processing where the algorithm performs repeated element wise multiplication on arrays of double precision data. Again the local memory was partitioned in a way such that the DMA could service part of the data while the processor operates on the other data.

- Finite Impulse Response Filter (FIR): The FIR benchmark is common is Digital Signal Processing and is used to filter signal data. It is a staged algorithm composed of a FFT, element wise multiplication, inverse FFT, and an element wise division.

4.1.2 Memory Bound

Memory bound applications have the potential to achieve substantial performance gains from the streaming nature of the SVM model.

- Matrix Corner Turn (Transpose):

  The matrix corner turn algorithm operates on a matrix of 250,000 elements. To perform the transpose the processor needs to execute a load, store, and the appropriate address calculations for each element. This is a very simple work function so the cost of transferring data into local memory is greater than the cost of executing the work kernel. This leaves the processor underutilized.

4.1.3 Other Types of Applications

Not all applications can be as easily decomposed across multiple processors as the current set of benchmarks. Applications with more frequent interprocessor communication would spend more time executing library code and synchronizing their actions. These applications still could benefit from the SVM model and future work

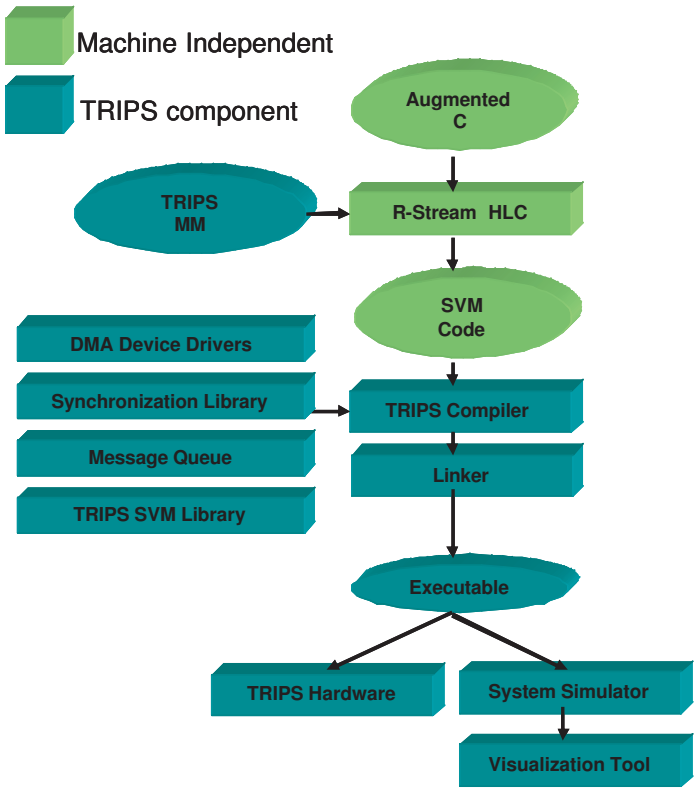

Figure 4.1: TRIPS SVM Toolchain

will expand the suite of benchmarks to examine the effectiveness of the SVM model for these applications that are not trivially parallelizable.

4.2 TRIPS SVM Toolchain

We developed these applications using the TRIPS SVM Toolchain shown in Figure 4.1. Initially, the author wrote them in Augmented C, a modified version of C with additional constructs to facilitate the HLC's discovery of parallelism. These Augmented C programs were compiled with the HLC along with the TRIPS machine model, that describes all of the system parameters of the TRIPS SVM system. Next

the TRIPS C compiler was used to compile the SVM code along with the support libraries discussed in Section 2.3. The system simulator, discussed in Chapter 3 was then used to run the resulting executable.

4.3 Analysis of HLC Generated Code

The author conducted extensive analysis of the output of the HLC to determine the HLC's ability to generate code that would be efficient on the TRIPS system. It became clear that several modifications would need to be made by hand in order to generate the most efficient SVM code for the TRIPS system. Since the HLC is still a prototype and there is ongoing work to improve the code quality this should not be seen as a sign that the two-phased compilation approach does not hold promise. All of the hand changes were made for performance reasons and not correctness. However, there were some cases where the HLC was not able to correctly identify how to parallelize code sections. In these cases, in order to produce working benchmarks, the code was hand generated. The SVM API does not depend on the HLC and the analysis undertaking was of the effectiveness of the SVM model rather than the HLC therefore hand coding for evaluation purposes was applicable. This hand coding highlighted several areas where the HLC could be improved and these observations were communicated to the HLC team who have included some of the features in the current generation of the HLC.

Address Generation: The original HLC generated code used function calls to access elements in a svm_block. This provides a high level of abstraction when dealing with data objects but has serious performance implications. Our analysis showed that this presented a significant cost on the TRIPS system and we proposed replacing the function calls with macros. While this provided a speedup, the low level compiler had difficulty unrolling loops with macro accesses. The macros

translated into pointer operations so when the loop was unrolled a separate address calculation was made for each pointer reference. When the macros were replaced with array accesses the compiler was able to reduce the number of address calculations and increase the unroll factor of the innermost loop providing a significant speedup. The three different approaches are shown in Figure 4.2. This optimization was communicated to the HLC team who implemented the change in subsequent versions.

Overlapping Communication and Computation: To fully exploit the SVM model one must ensure that communication and computation are overlapped. This ensures that the processor does not have to sit idle while data is transfered into its SRF. On some algorithms, such as matrix corner turn, where the work function is very simple this may not be possible. However, the complexity of many algorithms is n^2 or higher while the cost to transfer data into the SRF space is linear. One way to overlap communication and computation is to partition the local memory so that both the processor and DMA can operate on different portion of memory. For example in a vector add program the SRF space would be partitioned into four input blocks and two output blocks as shown in Figure 4.3. While the processor is adding two of the input blocks to produce an output block the DMA can be storing the results of the other output block and refilling the other two input blocks. Once the processor finishes operating on its data the processor and DMA switch memory spaces so that the processor operates on the new data that the DMA copied in and the DMA operatates on the results of the processor's computation. This technique proved very effective at improving the processor utilization.

Distributed Control: In the initial HLC generated code, computation was divided into rounds and the processors synchronized at the end of each round after which the master would assign another round of work. The programs were modified

so that all of the control was pushed to the individual processor so the master only has to intervene at the very beginning and at the very end of program. As this model scales to a larger number of processors it is important that it is decentralized.

```
// Using Function Calls
void svm_blockRead(svm_block block, int index, double * data) {
  (*data) =  *((double *)(block->address + index * (block->elementSize)));
}
void svm_blockWrite(svm_block block, int index, double * data) {
  *((double *)(block->address + index * (block->elementSize))) = *data
}

double A, B, C;
for (i=0; i < 250; i++) {
  svm_blockRead(blockA, i, &A);
  svm_blockRead(blockB, i, &B);
  C = A + B;
  svm_blockWrite(blockC, i, &C);
}
```

```
// Using Macros
#define svm_blockReadMacro(block, index, data, type)
  *((type*) data) = *((type*) (index * block->elementSize + block->address

#define svm_blockWriteMacro (block, index, data, type)
  *((type*) (index * b->elementSize + b->address))= *((type*) data)


double A, B, C;
for (i=0; i < 250; i++) {
  svm_blockReadMacro(blockA,i,&A,double);
  svm_blockReadMacro(blockB,i,&B,double);
  double C = A + B;
  svm_blockWriteMacro(blockC,i,&C,double);
}
```

```
// Using Array Indexing

double * A = blockA->address;
double * B = blockB->address;
double * C = blockC->address;

for (i=0; i < 250; i++) {
  C[i] = A[i] + B[i];
}
```

Figure 4.2: Different Addressing Options for Vector Add Program

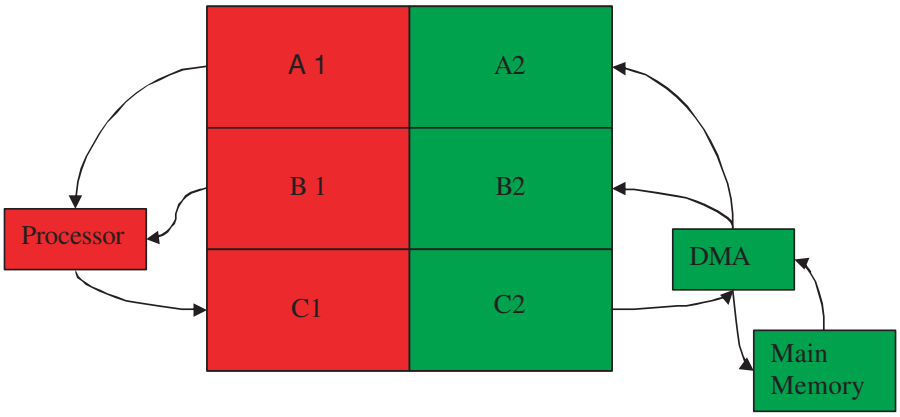

Figure 4.3: Processor's Local Memory for Vector Add Program

Chapter 5

Results

In order to test the effectiveness of the SVM model on the TRIPS system, we simulated the sample SVM programs with the modified simulator on a prototype system composed of 4 processors. Further, we developed sequential versions of all of the benchmarks in native C and used the modified sequential simulator to run them.

5.1 Speedup over Sequential

We show the results of running the applications in Table 5.1. We see that the speedups for the three computationally bound applications range from 3.6 to 3.7 which is within 10% of the optimal speedup of 4.0 that could be expected from 4 processors. However on the memory bound application our initial results indicate that the SVM program is actually outperformed by the sequential program. This result arises because our simulation model currently makes no attempt to model main memory access latency so all the longest delay a memory instruction can be assigned is an L2 hit. The corner turn application consists of 250,000 pairs of load and store operations which on the SVM system would hit in the local memory with

Application	SVM Program (millions of cycles)	Sequential Program (millions of cycles)	Speedup over Sequential
Matrix Multiplication	687.0	2,554.8	3.718
Finite Impulse Response Filter	32.7	121.9	3.730
Fast Convolution	5.7	20.8	3.646
Matrix Corner Turn	5.7	5.1	0.922

Table 5.1: Initial Results

an approximate cycle cost of 10-15 and on the sequential applications these memory operations would face a main memory latency of several hundred cycles. Therefore, we expect the SVM version of corner turn to outperform the sequential version and additional work could extend the simulator to account for main memory access latency.

5.2 Execution Breakdown

To evaluate the overhead of the SVM model we examine the amount of time the processor spends performing different operations. Table 5.2 shows a breakdown of the execution time of the master processor. We see that for the three computationally bound programs only 1-2% of the execution time is spent setting up the program or executing SVM library code. This high percent of useful work explains why we were able to achieve near ideal speedups. On the matrix corner turn benchmark the program spends nearly 40% of its execution time running overhead code for the SVM model. This corresponds with poor performance of the SVM version. To visualize program execution, we developed a tool that creates a graphical view of processor and DMA actions. Figure 5.1 show the visualization of the matrix multiplication

Benchmark	% useful work	%work assignment	%DMA control	%message queues, init, sync
Matrix Multiplication	**99.799**	0.042	0.129	0.065
Finite Impulse Response Filter	**98.207**	0.004	0.012	1.777
Fast Convolution	**98.917**	0.002	0.003	1.078
Matrix Corner Turn	**63.312**	13.204	16.512	6.972

Table 5.2: Execution Breakdown

SVM program. A dark horizontal line means that work is being performed at that time step. The four leftmost solid vertical lines show that the four processors are performing work throughout the entire execution. The four rightmost dashed vertical lines show that the four DMAs are only used intermittently throughout the execution. While it appears that the DMA are used a high percentage of the time, this is an artifact of the scale of the picture and Section 5.3 shows that the DMA utilization for the matrix multiplication program is less than 1%.

5.3 DMA Utilization

Another useful metric is what percent of the time the DMA was transferring data. Table 5.3 shows the DMA utilization of the four benchmarks. We see that the computationally bound programs make little use of their DMAs while the corner turn program uses its DMA over fifty percent of the time. The current model of the cost of a DMA operation is rather optimistic so the latency of a DMA operation in the actual system is likely higher than we predict. Therefore the DMA utilization in the actual system are likely to be higher than shown here and matrix corner turn for example will likely use its DMA almost all of the time. The DMA utilization for the computationally bound kernels might suggest that a DMA controller is not

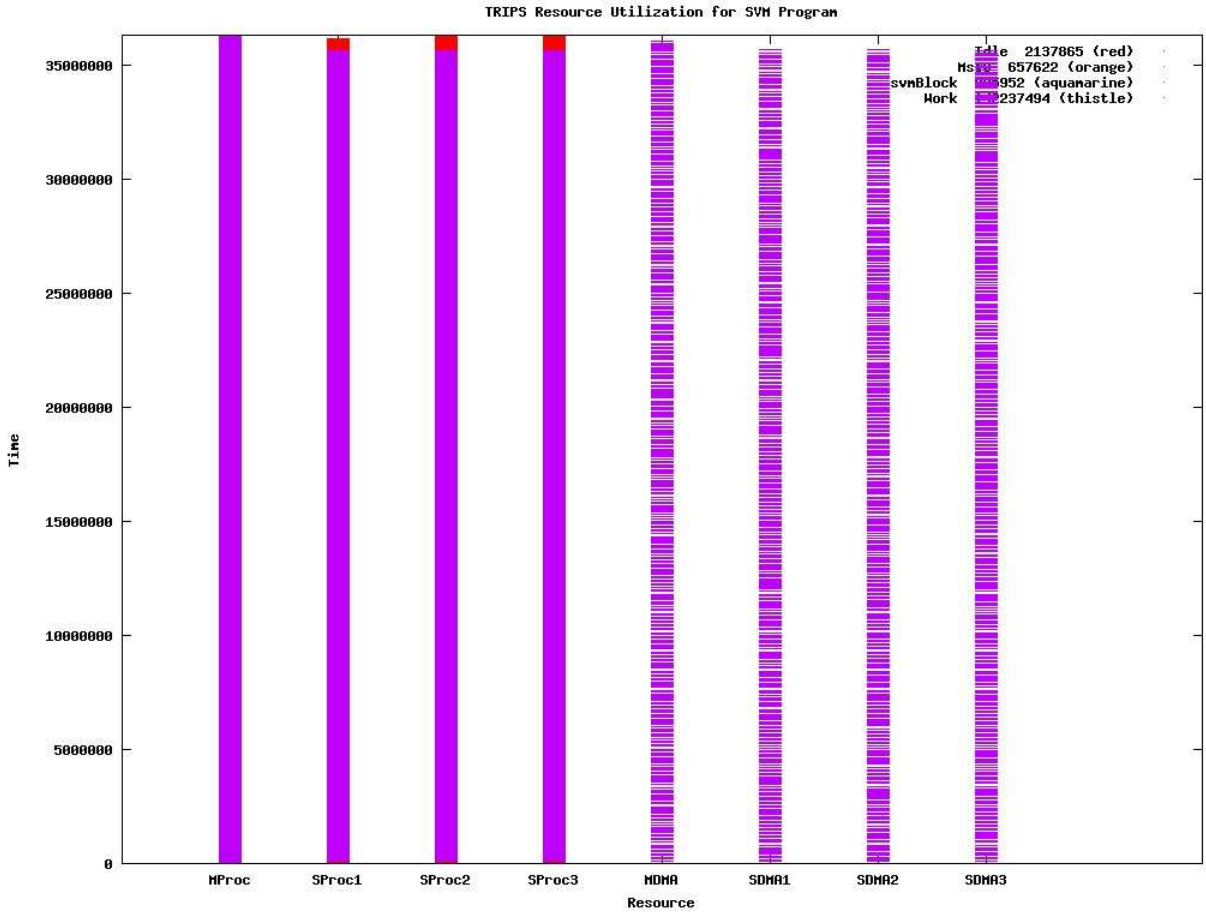

Figure 5.1: Visualization of Matrix Multiplication

Benchmark	DMA Utilization (%)
Matrix Multiplication	0.139
Finite Impulse Response Filter	0.046
Fast Convolution	0.025
Matrix Corner Turn	57.028

Table 5.3: DMA Utilization

necessary since it was underutilized. However, the utilization will increase with a more accurate model of the latency and the use of the DMA could have a larger impact on different classes of applications such as those with more frequent inter-processor communication.

5.4 Library Optimizations

Using the modified simulator to determine where execution time was spent, the author made several optimizations to the TRIPS SVM library. Initial analysis showed that the computationally bound kernels spent less than 2% of their time executing library code, so extensive effort was not spent completely optimizing the library. However, the memory bound application spent a far greater amount of time, around 40%, executing library code so optimizations provided more benefit for them.

5.4.1 Compilation with Full Optimizations

The initial SVM library did not use the volatile keyword to mark shared variables. In C the volatile keyword is used to indicate to the compiler that no optimizations should be performed on a variable. Without this keyword the TRIPS C compiler was register allocating shared control variables that processors polled on, creating race conditions. The initial solution was to compile the SVM library with no opti-

Application	Original Speedup	-O4 Compilation	Overall Improvement to Execution Time (%)
Matrix Multiplication	3.718	3.756	1.00
Finite Impulse Response Filter	3.730	3.680	-1.34
Fast Convolution	3.646	3.679	.89
Matrix Corner Turn	0.899	1.096	15.45

Table 5.4: -O4 Compilation

mizations but this had performance implications. The author augmented the library with the correct use of the volatile keyword and then compiled the library with full optimizations Table 5.4 shows the results of using the -O4 optimizations. The higher level of optimization actually hurt the performance of the FIR benchmark because of some issues regarding how the compiler backend handles the volatile keyword are still being resolved.

5.4.2 Linear Search

Analysis of execution time found several functions that were performing linear scans of an array of processor ids in order to translate from a SVM virtual identifier to a physical processor identifier. This code was replaced with an array lookup to avoid the linear scan. This will also serve to reduce pressure on the shared segment of memory where processor identifiers are stored. Table 5.5 shows the results of performing this optimization.

5.4.3 Library Race Conditions

Along with optimizations the author spent a significant amount of time debugging race conditions in the library code. Since the library code was developed on the original functional simulator, where each block cost the same amount of time to

Application	**Original Speedup**	**-O4 Compilation**	**Overall Improvement to Execution Time (%)**
Matrix Multiplication	3.718	3.758	1.05
Finite Impulse Response Filter	3.730	3.721	-0.22
Fast Convolution	3.646	3.698	1.42
Matrix Corner Turn	0.899	1.199	23.12

Table 5.5: ID Lookup Optimization

execute, when the simulator was modified, to execute blocks in an order based on the estimated cycle time, several race conditions appeared. Additional checks were inserted to prevent these scenarios from occurring.

5.4.4 Future Optimizations

There is still room for further optimization to the SVM libraries. Currently whenever the DMA is invoked to transfer data a system call is made to translate the virtual address to a physical address. This is a very expensive operation and a more efficient method for performing this translation would be to only translate the base addresses for each segment and then generate all physical addresses as an offset from the known base address. Since we do not attempt to model the cost of system calls, had this optimization been made we would not have been able to measure its effectiveness and therefore leave it for future work.

Chapter 6

Conclusions

This thesis evaluates the performance of the SVM model of parallel computation on the TRIPS system. The SVM model provides a method of dividing large computations amongst a collection of processors while trying to minimize the burden on a programmer. While we found the current High Level Compiler to be weak in certain areas, optimizations have been discovered which will hopefully improve future version of the High Level Compiler. Along with work distribution the SVM model uses a streaming approach which takes advantage of the TRIPS processor's DMA to mitigate the high cost of main memory access.

Related work, such as the StreamIt programming language, has exploited concurrency through high level language support for streaming. StreamIt aims to improve programmer productivity by directly exposing high-level streaming constructs that the programmer can use to write a concise program that the StreamIt compiler can execute effectively, using a streaming model of computation [20]. All of our current benchmarks have focused on applications where processors always return their results directly to main memory. In some situations it may be preferable for processors to pass their results directly to another processor, bypassing main memory. Applications that have several distinct stages could benefit from this

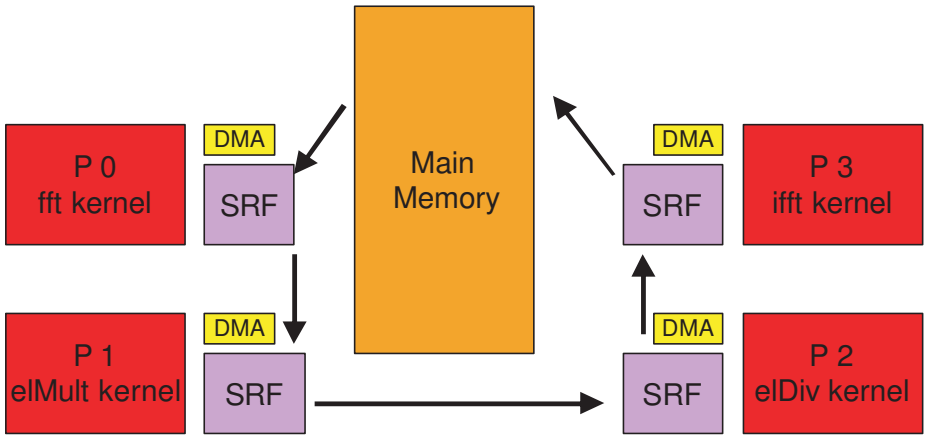

Figure 6.1: Possible Execution of FIR Algorithm

model, as long as the work is distributed approximately equally across the stages. Figure 6.1 shows an alternative implementation of the FIR algorithm, where each processor performs one of the four stages of the FIR algorithm and then uses its DMA to transfer its results directly from its local memory to another processor's local memory. Then the DMA can move the final result to main memory. This approach would also improve the i-cache performance since each processor is only executing a subset of the total code. Future work compare this approach of work distribution to the approach where each processor pulls and pushes data directly between main memory and local memory.

To quantify the performance of the SVM model, enhancements were made to the functional simulator to determine the approximate cycle cost of a TRIPS program. While several simplifications were made, this estimate proved to be within 20.34% of the true value on the EEMBC benchmark suite. Further the analysis code only introduced an overhead of 20%, meaning that it was still 800 times faster than the cycle accurate simulator. Future work will enhance the simulators memory model and add a model of the branch predictor.

The suite of SVM benchmarks shows that with 4 processors the SVM program executes 2.95 times faster than a native C sequential version of the same application. On the three computationally intensive benchmarks the SVM version is 3.70 times faster than the sequential version. This shows that for a large class of applications the overhead of using the SVM model can be mitigated and significant speedups can be obtained. Future work on the SVM project will extend the suite of benchmarks and quantify the performance for applications that express less inherent parallelism.

Bibliography

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, pages 44–55, July 2004.

[2] DARPA. *Polymorphous Computing Architecture Program, http://www.darpa.mil/ipto/programs/pca*, January 2006.

[3] S. Drolia. Support for Stream-based Parallel Programs:The TRIPS Architecture. Master's thesis, University of Texas at Austin, December 2005.

[4] S. Drolia and S. Keckler. TRIPS DMA Controller Specification. Technical Report Internal, Department of Computer Sciences, The University of Texas at Austin, 2005.

[5] M. Gebhart and S. W. Keckler. Matrix Multiplication on TRIPS SVM System. Technical report, The University of Texas at Austin, December 2005.

[6] Georgia Institute of Technology and Space and Naval Warfare Systems. Introduction to Morphware. Technical report, Polymorphous Computing Architecture, 2004.

[7] C. Kim, P. Gratz, and R. McDonald. On-chip Network Specification. Internal Technical Report, Department of Computer Sciences, The University of Texas at Austin, July 2004.

[8] M. Krishnan and S. W. Keckler. Implementation of MPI on TRIPS Specification Document. Technical report, The University of Texas at Austin, April 2006.

[9] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time Scheduling of Instruction-level Parallelism on a RAW Machine. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 46–57, New York, NY, USA, 1998. ACM Press.

[10] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA 2000: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, 2000.

[11] P. Mattson. PCA Machine Model. Technical report, Reservoir Labs, December 2004.

[12] P. Mattson, B. Thies, L. Hammond, and M. Vahey. Streaming Virtual Machine Specification. Technical report, Polymorphous Computing Architecture, March 2005.

[13] R. McDonald. Proposed Thread Synchronization Support. Internal Technical Report, Department of Computer Sciences, The University of Texas at Austin, June 2004.

[14] K. S. McKinly, J. Burrill, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The Scale Compiler. Technical report, University of Massachusetts, University of Texas, 2005.

[15] R. Nagarajan, S. K. Kushwaha, D. Burger, K. McKinley, C. Lin, and S. W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *International Conference on Compilation Techniques (PACT)*, September 2004.

[16] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch Parallel Processor Hardware Design. *Computer*, 23(4):18–28, 30, 1990.

[17] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceddings of the 30th Annual International Symposium on Microarchitecture*, pages 422–433, May 2003.

[18] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High-ILP Processors. *IEEE Micro*, 24(6):118–127, 2004.

[19] A. Smith, J. Burrill, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. TRIPS Application Binary Interface (ABI) Manual. Technical Report TR-05-22, Department of Computer Sciences, The University of Texas at Austin, March 2005.

[20] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Apr 2002.

Vita

Mark Gebhart was born on August 3, 1983 in Richardson, Texas. He graduated from Berkner High School in May of 2002 and enrolled at the University of Texas at Austin in the Fall of 2002. During his summers he has completed two internships with Lockheed Martin and an internship with the Department of Defense. He began working in the Computer Architecture and Technology Laboratory (CART) under Dr. Stephen W. Keckler in August of 2004. In May of 2006 he graduated with a Bachelor of Sciences in Computer Science and plans to begin work on a masters degree in August 2006 at UT-Austin.

Permanent Address: 2217 Windsor Dr., Richardson, TX 75082

This undergraduate honors thesis was typeset with LaTeX 2_ε[1] by Mark Gebhart.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this undergraduate honors thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.