Scheduling of Processes for a Simultaneous Multithreading Processor

A Thesis
Submitted to the Faculty of

Rose-Hulman Institute of Technology

by

Theodore J. Gould

In Partial Fulfillment of the
Requirements for the Degree

of

Master of Science in Electrical Engineering

May 2000

Abstract

Simultaneous Multithreading (SMT) is an architecture which allows for multiple thread contexts to exist simultaneously inside the processor. This thesis analyzes the relationship between this architecture and the software running directly on top of it, specifically in scheduling. A model is built to simulate different scheduling algorithms for use within this relationship. Four algorithms are tested with the model on data sets created to simulate real world conditions for the processor. Two of the scheduling algorithms allow for commonly used thread contexts to remain within the processor at the end of a timeslice. The results show that allowing these thread contexts to remain in the processor can shorten wait and response times without sacrificing performance on a variety of workloads.

To Stephanie,
for being the light behind
my brightest days

Acknowledgements

It is traditional in American culture to believe that one person writes a paper. That person is called the author. While this is the person who is given the credit, it is rare that the person in question worked in a vacuum. Countless people have helped to create the paper from conception to realization. This paper is not an exception. I would like to thank my Thesis Committee, for their nights and weekends of reading and suggestions for improvement. My friends and family, for their continual support and understanding when I had to excuse myself to go and write. Overall, this paper's origins are too numerous to mention, but, I would like to thank all of them.

Table of Contents

List of Tables vii

List of Figures viii

Glossary of Terms x

Chapter 1: Introduction 1

Chapter 2: Background 3

- Pipelining Processors 3
- Superscalar and Out-of-Order Execution 5
- Memory Problems 6
- Enter SMT 6
  - Filling the Pipelines 7
  - Costs of SMT 8
- Competing Architectures 10

Chapter 3: Modeling and Data Gathering 12

- Simulation Methods 12
  - Process Representation 13
  - Process Execution Engine 14
  - OS Role 15
  - Scheduler 15
  - Data Set Generation 16
  - Data Gathering and Graphing 16
- Scheduling Algorithms 17
  - Traditional Processor 18
  - Load Top 19
  - Shift In 20
  - Persistent Data 21
  - Short Form 23
- Data Sets 24
  - Solid 24
  - Simultaneous 25
  - Spiked 25
  - Long 26

All Data Sets 26
Short Form 27
Statistics Gathering 28
CPU Utilization 28
Response Time 28
Wait Time 29
Throughput 29
Assumptions 30
Process Blocking 30
Simplicity of OS call 31
Round Robin "divides out" 31
Overview 31

Chapter 4: Data Analysis 33

Long 33
Simultaneous 36
Solid 37
Spiked 39
Overall 42

Chapter 5: Conclusion 43

Further Research 44

List of References 45

Appendix A: Scheme Source Code 49

Description of Code 49
core.ss 49
dataset_long1.ss 51
dataset_long2.ss 51
dataset_long4.ss 51
dataset_long8.ss 51
dataset_long16.ss 52
dataset_simultaneous.ss 52
dataset_solid.ss 52
dataset_spiked.ss 52
display.ss 52
load.ss 54
os.ss 54
proc-run.ss 54
record-def.ss 55
run.ss 56

schedule_one.ss 56
schedule_shift.ss 57
schedule_shift_persist.ss 58
schedule_top.ss 60
schedule_top_persist.ss 60
variables.ss 62

Appendix B: Matlab Source Code 63

Description of Code 63
getdata.m 63
getdataset.m 63
getgraph.m 64
getscheduler.m 65
makegraph.m 65

List of Tables

Table 3.1 Comparison of scheduling algorithms used in model. 23

Table 3.2 Comparison of the Data sets. 27

Table 4.1 Table of results comparing scheduling algorithms to data sets. 42

List of Figures

Figure 2.1 Issue slots for a traditional processor over several cycles. (Tullsen, 1998) 7

Figure 2.2 Issue slots of a processor with SMT. Four threads are executed. 8

Figure 3.1 Graph showing the number of frames for a process given the percent CPU usage. The traditional processor is represented by the straight line. The curved is the shift in algorithm. 20

Figure 3.2 Graph showing the number of frames that a process gets using the Shift In algorithm. Lines represent the number of thread contexts in the processor, with the linear relationship being for one. 21

Figure 4.1 Graph showing throughput while varying the number of processes in the data set on the long data set. 34

Figure 4.2 Graph showing throughput while varying the chance of an OS call on the long data set with 8 processes. 34

Figure 4.3 Graph showing average wait time while varying the number of processes in the data set on the long data set. 35

Figure 4.4 Graph showing the maximum wait time varying the number of processes in the data set on the long data set. 35

Figure 4.5 Graph showing average wait time while varying the memcall attribute in the simultaneous data set. 36

Figure 4.6 Graph showing maximum wait times while varying the memcall attribute on the simultaneous data set. 36

Figure 4.7 Graph showing throughput while varying the percentage of memory calls on the simultaneous data set. 37

Figure 4.8 Graph showing average wait time while varying context switch time on the solid data set. 38

Figure 4.9 Graph showing the average response time while varying context switch time on the solid data set. 38

Figure 4.10 Graph showing average wait time while varying the frame size on the solid data set. 38

Figure 4.11 Graph showing CPU usage while varying context switch time on the spiked data set. 40

Figure 4.12 Graph showing throughput while varying context switch time on the spiked data set. 40

Figure 4.13 Graph showing the average response time while varying the context switch time on the spiked data set. 41

Glossary of Terms

Central Processing Unit (CPU) - the part of the computer where the majority of processing is done. This typically controls all other parts of the computer.

CPU - *see Central Processing Unit*

ILP - *see Instruction Level Parallelism*

IA-64 - a 64-bit processor instruction set developed by Intel and Hewlett-Packard which has explicit ILP.

Instruction Level Parallelism (ILP) - a term to describe how much dependency the instructions of the program have on each other.

Operating System (OS) - the computer program that controls the basic functions of the hardware and schedules the processes for the user.

OS - *see Operating System*

process - a single list of instructions which can be executed. Sometimes this is thought of as a program, but a single program can contain smaller processes called threads.

Simultaneous Multi-Threading (SMT) - a process which allows multiple threads to be executed simultaneously inside a processor.

SMP - *see Symmetric Multi-Processing*

SMT - *see Simultaneous Multi-Threading*

Symmetric Multi-Processing (SMP) - describes a computer that has multiple processors with similar characteristics as to make controlling them easier.

Thread Level Parallelism (TLP) - the term used to describe the parallelism that can exist between multiple threads in a system.

TLP - *see Thread Level Parallelism*

Chapter 1: Introduction

In the last twenty-five years, the architecture with which the Central Processing Unit (CPU) has been designed significantly changed. This change, along with the advances in chip development, has lead to one of the largest increases of price/performance of any product. Although the developments in the processes that produce the chips are remarkable, to cover everything would take much more than one paper. As each significant change in processor design is developed, all of the relationships with that processor functions and the system around it have to be rethought. Although this is not necessary always, it is done to satisfy the insatiable craving for speed that the human race seems to have. The architecture change to implement Simultaneous Multi-Threading (SMT) is no different.

SMT moves computer architecture to a new phase in its history, a point where Thread Level Parallelism (TLP) is exploited. This changes the way one looks at the processor. No longer is it a slave to the Operating System (OS); rather it is an active participant in the threading process. Tasks can be executed while the OS is handling other processes's system calls. This is not entirely new as supercomputers have been dealing with many of these issues throughout their history. Never has this been dealt with on a single processor system. This is why the components of the overall computer system must be rethought.

Does it make sense for an ethernet card to be able to have its own private thread in the processor for encryption? Can the hard drive use a thread to defragment the filesystem in real time? Could threads turn into a resource that every process is given when it starts up similar to file handles? These tasks, while they seem very high level for what a processor should be handling, are possibilities that can be explored in an SMT world. Today, the first step is the most important. The first step is to get cooperation between the operating system and the SMT architecture. The next step is for further research (which is discussed in Chapter 5).

The total effect of making the OS more "SMT aware" will be a performance increase. To make an architecture successful it cannot only implement something that is successful on the micro level, it must also improve the performance of the overall system. Most users do not really care about the details in the processor architecture of the system; they only care about the throughput that their program is able to achieve. Total system performance is achieved through the integration of the fundamental architecture of the system and the OS that runs above it. Unlocking the performance gain achievable through this integration is the key point of this thesis.

This thesis starts by looking at the past. Chapter 2 looks at both the history of the microprocessor and of SMT. Chapter 3 then starts to talk about the model that was created for this research, how it works, and what it does and does not do. In Chapter 4, data is generated and analyzed for all of the data sets. Chapter 5 then brings everything together into a conclusion. It also discusses how further research could be carried out on this subject. Finally, the Appendix contains all of the source code that was used for this research.

Chapter 2: Background

SMT is a new way of looking at technologies that have found themselves in processors of the past. To truly understand the impact of SMT, one must look at the technologies that form its foundation. In this chapter, the processor technologies of pipelining and out of order execution are discussed. Together these make up the traditional processor used later in the simulation. There is also a short discussion on why memory delays are still an issue with these two processors, and a small outline of the memory hierarchy. Next, a large section describes the development of SMT on top of these technologies. This outlines the previous research on the subject and some of the results of that research. Lastly in this chapter there is some discussion on the IA-64 architecture, another modern architecture in development, for comparison against SMT.

Pipelining Processors

Patterson and Hennessy[1] equate the pipelining of processors to a automobile assembly line. They talk about how the instruction goes through different stages of

---

[1] Patterson, David A. and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, (San Mateo, Calif.: Morgan Kaufmann Publishers, 1994), 364

the processor, much like a car getting its bumper installed at one station and its windshield at the next. This analogy works very well in creating the idea of how the processor works, but it does not explain all the issues involved in pipelining a processor.

A processor is divided into several sections including registers, memory access, and mathematical units. These are not all of the units in a processor, but they will suffice for this discussion. When the processor is doing a memory access, it is not simultaneously using the mathematical units. This means that another instruction could use these units while the first is waiting on the memory access units. By standardizing the order that instructions can use the differing units in the processor, each unit can be servicing an instruction simultaneously in the processor. This institutes better use of hardware already in the processor.

The length of time that each of the individual units takes becomes the limiting factor on the clock speed of the processor. This increases the overall throughput of the processor.

> Under ideal conditions, the speedup from pipelining equals the number of pipe stages; a five-stage pipeline is five times faster. Usually, however, the stages are imperfectly balanced. In addition, pipelining involves some overhead.[2]

While the ideal pipeline increase the speed of the processor, hazards are introduced into the processor.

Several different types of hazards exist including data, control, and branch

---

[2] Ibid., 365.

hazards. Each hazard causes the processor to be unable to use its pipeline most efficiently. The processor must detect these hazards and apply avoidance techniques that have been developed to remedy the situation. While overall time is saved, the processor grows in size with the additional hardware to detect these hazards, and also slows down slightly from the ideal pipelined processor. Typical overall speedup is two and a half to three times on four to eight stage pipelines.[3]

Superscalar and Out-of-Order Execution

Superscalar processors take the ideas of pipelining and literally double them. Although they are not limited to two, the first superscalar processors had two pipelines that were identical to each other to execute instructions. This adds problems as machine code expects to be executed in the order that it was compiled. Sometimes the execution of a single instruction is blocked by one unit that is slower than other units in the pipeline. If the next instruction does not need that unit, then it could move on through the pipeline without waiting for the instruction in front of it. This is what is called out-of-order execution. Both superscalar and out-of-order execution have issues dealing with instructions that are dependent on one another.

With some sets of instructions, it is completely impossible to dispatch to a second pipeline. It is also impossible to speed up execution by moving the instructions out-of-order. When a set of instructions does not allow for optimization, the extra hardware is wasted, and the processor is reduced to a simple pipelined processor.

---

[3] Ibid., 437.

Memory Problems

"Because CPU speeds continue to increase faster than either DRAM access times or disk access times, memory will increasingly be a factor that limits performance."[4] For this reason, a memory hierarchy is created in modern computers. This hierarchy consists of a hard drive at the bottom, standard DRAM and cache in the middle, with register files on the top. This allows the computer to keep the data which is used the most in the fastest memory, thus hiding the fact that most of the memory is relatively slow. The memory hierarchy also enables programs to use as much memory as they would like, especially as hard disk prices continue to drop.

While the memory hierarchy does a good job of hiding the slower memory, it is not perfect. When the processor is forced to access main memory, or worse, the disk, it is forced to stall other instructions from entering the processor. Superscalar and out-of-order systems are able to deal with this to some extent, but for long requests even they are forced to stall. As cache miss rates approach almost one percent[5], a 500 MHz processor still has over 5 million misses a second. Anything that a processor can do to help with the memory speed problems can add up very quickly.

Enter SMT

The developments of SMT started in early 1994 as a group of researchers realized that modern processors were not getting as much performance out of the chips

---

[4] Ibid., 519.

[5] Ibid., 514.

as they could.

> We were beginning to see commercial microprocessors that could issue many instructions per cycle (wide superscalars), but which rarely did so due to dependencies and long memory latencies. In fact, processor utilization seemed to be declining as fast as instruction issue width was increasing.[6]

They then started to work towards a general solution to allow multiple threads to fill up the unused cycles in the processor. By using all of the instruction cycles that are available to the processor, the overall performance of the system is increased.

Filling the Pipelines

Through the hazards and dependencies that were discussed earlier in this chapter, the non-SMT, or traditional, processor is unable to fill all of its pipelines every cycle. In some cases, it can even be as bad as shown in Figure 2.1. This figure also introduces the terms horizontal and vertical waste. Horizontal waste occurs when there are not enough instructions that can be executed in parallel to fill all of the pipelines. Vertical waste occurs when instructions are dependent on a previous instruction, but that instruction is blocked in

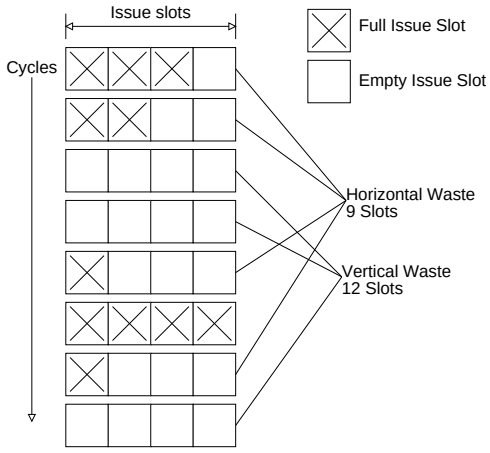

Figure 2.1 Issue slots for a traditional processor over several cycles. (Tullsen, 1998)

---

[6] Tullsen, Dean M., Susan J. Eggers and Henry M. Levy, "Retrospective: Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *25th Year of the International Symposia on Computer Architecture*, ed. Gurinder Sohi (New York: ACM Press, 1998), 115.

some way, usually waiting for memory. Both of these are possible issue slots where an instruction is not started, and thus a waste of the processor.

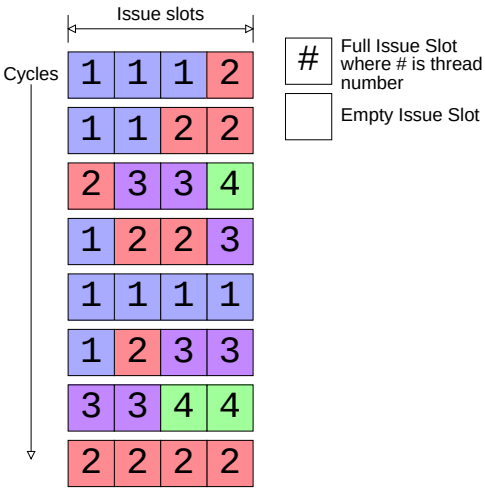

Figure 2.2 Issue slots of a processor with SMT. Four threads are executed.

SMT fixes this problem by keeping multiple thread contexts in the processor simultaneously. This can be seen in Figure 2.2. In this picture the additional issue slots are filled by the other threads that are in the system. Although it would be ideal that all of the slots would be filled, realistically there will always be slots that go unused. This is because the instructions from the different threads will not have dependencies on each other. Research has shown that with four threads vertical waste is reduced to less than 3%.[7] This is an incredible step forward compared to a 8-issue traditional processor where vertical waste is closer to 50%.[8] By allowing multiple threads to execute in these otherwise lost cycles, the overall performance of the processor increases to otherwise unprecedented levels.

Costs of SMT

Nothing good comes without a cost. Thankfully, the cost to implement SMT

---

[7] Tullsen, Dean M., Susan J. Eggers and Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *25th Year of the International Symposia on Computer Architecture*, ed. Gurinder Sohi (New York: ACM Press, 1998), 538.

[8] Ibid., 537-8.

in a processor that is already superscalar is minimal. One of the biggest complications in adding SMT to a processor is the ability to handle the issuing of multiple instructions per cycle. This is already handled in the superscalar design. The out-of-order execution units also allow for the multiple threads to be independent of each other. The TLB and branch prediction can be reused between all of the threads and still be effective.[9] This also means that adding SMT to a processor does not have a large impact on the design of the processor.

One component where SMT has a big impact that is not included in a standard superscalar processor is in the size of the register file. The SMT register file must be larger than a traditional processor's register file. If four threads are executing simultaneously, then the register file needs to be four times as large. This becomes a significant limitation as it is difficult to create a register file which can access large numbers of registers within the shortening cycles at which processors are running. Here is a place where techniques in register relocation[10] could be useful, but are not currently being used in SMT research today. These techniques could reduce the size of the register file needed, however it would still need to be larger than a standard register file, making the implementation difficult.

---

[9] Lo, Jack L., et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," http://www.digital.com/alphaoem/papers/smt-vs.pdf (20 July 1999).

[10] Waldspurger, Carl A. and William E. Weihl, "Register Relocation: Flexible Contexts for Multithreading," *Proceedings of the 20th Annual Symposium on Computer Architecture*, (New York: ACM Press, 1993).

Competing Architectures

With any problem, different groups of people are going to come up with different solutions. Two solutions that are competing in the marketplace with SMT are the IA-64 architecture and Multiprocessors (MP). Each solution tries to use the additional area that is available to today's microprocessor to achieve an overall performance increase for the user.

The IA-64 architecture tries to remove some of the dependencies that the instructions have on each other in the compiler. The compiler can optimize instruction ordering and can explicitly set instructions to execute in parallel inside the processor. The compiler can also decrease memory latencies by executing speculative commands to pull up data from memory ahead of time. The IA-64 also handles predication, which allows branches to be handled quickly within the processor. The IA-64 architecture, to a large extent, increases instruction level parallelism (ILP) for the processor.[11] Unfortunately, there are no formal results comparing the two architectures yet.

MPs take the space that is normally dedicated to a wide superscalar processor and fill it with several smaller processors. This also exploits TLP as different threads are sent to each processor independently of the other processors. This solution also takes care of many of the synchronization problems that occur with Symmetric Multiprocessor (SMP) systems. By having the two processors share the same cache and memory, they do not have to be continually synchronized. Jack Lo, et al., in their

---

[11] Intel Corporation, *IA-64 Application Developer's Architecture Guide, Rev. 1.0*, (Intel Corporation, 1999), ch. 9.

paper[12] compare several MP designs with SMT architectures. Their conclusion is that the SMT design is more flexible in handling both ILP and TLP where it is available. The MP designs were optimized for either ILP or TLP, but not both.

[12] Lo, Jack L., et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," http://www.digital.com/alphaoem/papers/smt-vs.pdf (20 July 1999).

Chapter 3: Modeling and Data Gathering

To analyze the effectiveness of different scheduling algorithms for SMT, a model was created. This model needed to simulate real workloads for the different scheduling algorithms and test their effectiveness on that workload. The first part of this chapter looks at the implementation details of the model that was created. Following that, the different scheduling algorithms tested in the experiment are discussed along with the data sets on which they run. These descriptions outline the idea and explain how the model was expected to perform in the simulation. Next, there is some discussion of the different statistics that will be gathered on each of the runs to allow for a comparison of the results. Lastly, discussion about the assumptions that are made about the model that has been created for this experiment are presented. For a detailed look at the implementations used in this experiment please see Appendix A and B for the source code.

Simulation Methods

To simulate the SMT processor several modules were used to represent the parts of the processor. Those modules are outlined in this section. All of the modules were written in Scheme using the petite version of the Chez Scheme 6.0a interpreter. The data gathering was done in Matlab version 5.3. All of the development was done

with RedHat Linux 6.0 for the x86 processor. All of the processing was done with Sun Solaris for the Ultra Sparc.

Process Representation

Each of the different processes are represented by a record in Scheme. The records themselves contain the following data fields: `oscall`, `oscalllength`, `cycles`, `id`, `output`, `engine`, `memcall`, and `starttime`. The `oscall` field gives the percentage of time that the process is making an operating system call. This is evaluated every time that the process is executed. When the process performs an OS call, it calls the `os-makecall` routine and then returns to its current position as an engine. This makes the process execution engine think that the process is done and it moves on, thus simulating a block for that process. The `oscallength` attribute determines how many cycles long the created OS call is. The `cycles` attribute records the number of cycles that it takes for the process to be finished executing. This number is used in the creation of the engine for the process. `Id` stores an unique identifier for each of the processes. This identifier is used by some of the scheduling algorithms to determine if a context switch has occurred. The `output` attribute is only used when the processes have finished executing. When the process is finished, it returns a list of all the times that it executed, and the number of cycles left to execute at that time. This data is used later in statistics gathering. The `engine` field is a place for the engine that the process uses to execute to be stored. This value is initially set to `null` and is populated by the `process->engine` function when the

process is moved onto the OS stack. The `memcall` variable stores a function that returns a value for how many cycles a process can use out of a given set of cycles. This emulates how processes spend large amounts of time waiting for data to be brought out of memory. This number also represents the inherent ILP of the process. If the `memcall` attribute is low, the process could either be very serial or memory dependent; the effect is the same in this model. The process cannot use all of the pipelines or, in this case, cycles it is given. SMT helps hide this by executing another process in the remaining cycles. Lastly, the `starttime` variable stores the time when the process will be moved into the OS queue to be executed by the processor. This is the basis of many of the statistics that are gathered about the scheduler's performance.

Process Execution Engine

The Process Execution Engine consists of two parts: the process queue and the engine function. The process queue function takes a list of processes, the amount of time that they need to execute, and moves through the queue executing the engines of the processes. In the initial call to this function, the number of cycles is four times the frame size. This represents a processor that has four pipelines that the processes can use for execution. It then looks to the first function in the list and gives it as much CPU cycles as its `memcall` attribute allows it to have. If the process returns that it is completely done, it is stored on the expired stack and its output is saved. Otherwise its new engine is saved and the rest of the cycles are given to another process in the queue. In the traditional processor case, it just exits. In the SMT

schedulers the next process is chosen by making a call to the `schedule-realtime` function, which is discussed in the scheduler subsection of this section. (The traditional scheduler also makes a call to `schedule-realtime`, which just returns the null list causing the function to quit).

The engine execution function is where the 'work' is done for the process. This is the function on which the engine is set up to work on. This function looks at the overall global time and decides if it has been sleeping or executing. If it has been sleeping, it logs its start of execution and calls itself recursively. Before it calls itself, it checks to see if it needs to make an OS call. If an OS call occurs, it returns the blocked engine to start execution again when the OS call is resolved.

OS Role

The role of the OS in this model is more than just providing services that the processes need. The OS calls are also representing large device accesses. For example, if a process was to access the network card, it would effectively be an OS call. The OS infrastructure is relatively simple in that all it does is add a process record to the stack where other OS processes already exist. This stack is in the global context such that the scheduling algorithms use it themselves to determine how the OS calls are dealt with by the scheduler.

Scheduler

The scheduler is a simple but key part of the entire model. The scheduler has two parts: `schedule` and `schedule-realtime`. The `schedule` function takes

in the list which represents the OS queue and the previously executed processor queue. This program then takes these two lists and returns them, once again, with the processes redistributed between them. Most of the scheduling algorithms also look at the global OS stack to move those processes into the processor. The `schedule-realtime` function is called every time a process has finished executing. For most cases this function just returns the next process on the processor stack, but the flexibility allows other permutations of scheduling algorithms to exist. This feature is exercised in the Persistent Data schedulers so that they can immediately handle the OS calls.

Data Set Generation

Overall, data set generation is a straightforward process. This section outlines the usage of the `make-data-set` function and some of its interesting components. The `make-data-set` function takes most of the same parameters that exist in the process records, except that it takes a number of processes, and generates their ids and start times. The start times are based on a spacing variable that is also passed. The `cycles`, `oscall`, `oscalllength`, and `memcall` variables are all plus or minus ten percent. This range allows each of the processes to be slightly different, much like a real OS. This still works with the specialized data sets that are discussed later in this chapter, as it does not significantly change the values that are passed.

Data Gathering and Graphing

To gather the data and display them in an intelligible manner, Matlab was

used. The `getdata.m` script runs the actual Scheme code, and move the data generated by `display.ss` into Matlab arrays. These arrays of data can be used to generate graphs of the trends that exist. To do this the `makegraph` function is used. This function takes a list of parameters, one of which should contain a list of values. It detects which of these is a list and varies the data along those values. Every point on the graph consists of the value generated by the SMT scheduling algorithm divided by the traditional processor's value for that configuration. This makes all of the graphs relative to the traditional processor.

Scheduling Algorithms

There are several scheduling algorithms that have different possibilities for being used in the SMT architecture. The simulation is designed to have a modular scheduling algorithm to allow multiple algorithms to be tested. This module can include anything from the traditional processor to the four different SMT schedulers that are being compared in this paper. The traditional processor is included to try and remove any bias of the model and the experimentation. All the final statistics are measured relative to the values generated by the traditional scheduler.

Research for SMT has dealt with looking inside the processor and compiler optimization. Previous research on multithreaded architectures has handled scheduling in more detail and is built on here. Rafael Saavedra-Barrera discusses[13] two methods

---

[13] Saavedra-Barrera, Rafael H., "Analysis of Multithreaded Architectures for Parallel Computing," *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, (New York: ACM Press, 1990).

of switching processes, which he calls "switch always", and "switch on load." Derivatives of these are discussed in this chapter as Load Top and Shift In, respectively. The persistent data algorithms steam from discussion on further research in Jack Lo's paper[14] under "Implications for Operating Systems Developers." He suggests looking at some threads as being more important in scheduling, and in this case the operating system is being determined an important thread.

There are four SMT based algorithms that are analyzed in this paper. The Load Top algorithm quickly moves through all of the processes in the OS by taking four at a time to process. This limits the amount of time each process is given, but each gets more tries at the processor. The Shift In algorithm takes the process on the top of the OS process queue and slowly shifts it into the processor. Meanwhile it shifts out the process which has gotten the most time, back onto the OS process queue. This moves slower through the queue, but it allows more time for each process. Finally, each of these algorithms is modified to have a persistent version of the algorithm. This means that one of the thread contexts is permanently filled with the OS context. This can allow for fast OS calls, and less context switching.

Traditional Processor

The traditional processor uses this model to create a traditional processing system. The `schedule` routine for this algorithm first looks at the OS stack; if there is something there, it starts to process that. If there is nothing on the OS stack, the

---

[14] Lo, Jack L., et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," http://www.digital.com/alphaoem/papers/smt-vs.pdf (20 July 1999).

scheduler grabs the next process off of the OS queue. The process that was originally in the processor is put on the end of the OS queue. This effectively creates a round-robin type scheduler in the OS. This is one of the reasons that all of the statistics are being analyzed relative to this algorithm. While it is known that the round-robin scheduler is not the most used in many real-world systems, this experiment is not concerned about issues like priority scheduling in the OS. The processes do not even contain a priority attribute. This experiment is more concerned with the relationship between the OS and the processor, not about the OS itself. By looking at the relationships between the different algorithms, this issue is neutralized.

Load Top

The Load Top algorithm looks at the top four processes in the OS queue and schedules them into the processor at the same time. Every time the scheduler is called, it has the potential to switch out all of the processes that are in the processor. This is a method similar to those which are used on SMP systems. The SMP systems have the advantage that they know all processors in the system are the same, and so all processes will receive the same amount of attention in a given time slice. That is not the case with SMT. This algorithm does have potential to perform well in testing, as it moves processes through the processor very quickly. Small workloads have the potential to get processed with shorter wait times than with other scheduling algorithms. However, one significant problem with this algorithm exists, it has the potential for starvation to occur. This occurs as the potential for a process to not get any processor time exists, even for extended periods of time. If the right number of

processes are in the OS queue a process could be assigned with the lowest priority for the processor consistently (this is easy to see with round-robin but in fact can happen with any other algorithm too, it is just harder to prove). This last slot may not get any actual processing time if all of the first three processes are able to use it. Even the second and third slots have potential for starvation with high values of `memcall`. Evidence of this will appear in the final statistics by having the maximum wait time value being extraordinarily high. This model does not have the potential for complete starvation as eventually there would only be one process, and the simulation will continue until they all are completed.

Shift In

With the Shift In algorithm, the process with the highest priority in the processor is the only one that is removed after the timing interrupt. By leaving the other three processes in the processor, they each have a chance to get the highest priority. This removes the starvation problem with the Load Top algorithm. With the Shift In algorithm, each process gets a larger part of a CPU frame to use for actual processing, even with its memory latencies and reduced ILP. The equation to determine the amount of a frame that a given process receives for its time in the processor can be seen in Equation 3.1 and is plotted in

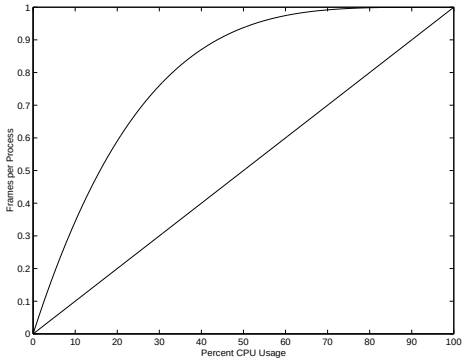

Figure 3.1 Graph showing the number of frames for a process given the percent CPU usage. The traditional processor is represented by the straight line. The curved is the shift in algorithm.

Figure 3.1. This figure shows the true benefits of SMT and this algorithm. At 100%

$$\sum_{i=1}^{n} x(1-x)^{i-1} \qquad (3.1)$$

CPU utilization, there is no difference between the line generated by the uniprocessor system. This is an ideal situation that will never happen. In the more practical 60% to 90% range, significant benefits can be seen. Even though at any one time the process is unable to use the whole processor, its total trip through the processor amounts closer to a complete frame of true processing time. At 80% CPU usage, the difference is almost indistinguishable.

Persistent Data

The Persistent Data method is applied to both the Load Top algorithm and the Shift In algorithm to create two new algorithms. The Persistent Data method allows some of the data to stay in the processor, in this case the operating system. By leaving this process continually in the processor no context switch is needed to move it into the processor. With the amount of OS calls that are made by the programs, the context switch time can be large. Unfortunately, this reduces the number of processes that can be in the SMT processor at one time. Some of the performance gained from simultaneous execution is lost. This loss for the Shift In algorithm can be seen in

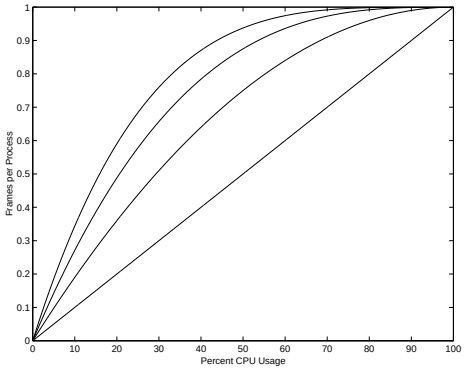

Figure 3.2 Graph showing the number of frames that a process gets using the Shift In algorithm. Lines represent the number of thread contexts in the processor, with the linear relationship being for one.

Figure 3.2. As the number of simultaneous processes drops, the curve gets much closer to the linear representation of the traditional processor. The gain is still significant though. Previous research[15] has shown that three thread contexts is the most economical gain, about 50%. Promising results for the persistent algorithms. These methods have the potential to decrease the overall execution and increase the overall throughput of the entire processor.

---

[15] Saavedra-Barrera, Rafael H., "Analysis of Multithreaded Architectures for Parallel Computing," 2nd Annual ACM Symposium on Parallel Algorithims and Architectures, (New York: ACM Press, 1990), 176.

Short Form

Table 3.1 contains a list of all of the scheduling algorithms contained above. It contains a direct comparison of the algorithms that are in this study.

Table 3.1 Comparison of scheduling algorithms used in model.

	Traditional Processor	**Load Top**	**Shift In**	**Persistent**
Description	Standard processor used for comparison of the other algorithms discussed	Loads the top four processes in the OS queue into the processor	Shifts the first processes on the OS queue into the processor	Allows some data to remain in the processor consistently
Advantages	- Simple - No new hardware required	- Quickly moves through all processes in OS queue - Small work loads will have small wait times	- Fair to all processes - Can effectively increase the amount of CPU time give to a processes in a time slice	- Lowers the number of context calls for the persistent thread
Disadvantages	- No TLP - Unable to hide large memory latencies with other processes.	- Potential for starvation - Long processes could have very long wait times - Lots of context switching	- Moves slowly through the OS queue, thus longer wait times	- Effectively lowers the number of threads that can be run in the processor

Data Sets

To test the various scheduling algorithms, different data sets were developed to simulate real-world conditions in which the processor could be put. Each of the data sets has approximately ten million total cycles between all of the processes in the data sets. Each of the data sets vary in how these cycles are laid out in time. The data sets are designed to create a set of extremes to push the scheduling algorithms to their limits.

The four data sets that are discussed each use their cycles in different ways. The Solid data set evens out the load over time. This allows the processor to have fewer processes, but new processes are constantly entering the system. The Simultaneous data set puts all of the processes up front. This puts a large number of processes in the queue for the algorithm to deal with right away. In the Spiked data set the processes are spread out evenly, with the exception of several large spikes of processes. This creates an instantaneous load on the algorithm that must be dealt with. Finally, the Long data set has very few processes, but each has a very long execution times. This tests an algorithm's ability to work with smaller numbers of processes.

Solid

The processes in this data set try to create a solid load over the processor at all times. The processes are spaced out in a manner to try and move another one into a processing position when the first one leaves. With this set it can be seen how the algorithm performs under heavy operation in an environment where large amounts of processing is done, and how well the jobs are managed. This data set does not take

into account how many OS calls are made or how many of the cycles the processes can actually use. This creates a situation where the processor has the potential to get backed up, and that is where the SMT will be useful.

Simultaneous

This data set puts all of the processes in the very beginning of time scale, and then lets the processor work on them and try to disperse them as quickly as possible. This will test the ability of the processor to handle large numbers of processes and see how well it can deal with such a task. This creates an opportunity for starvation in the Load Top scheduling algorithm, when the number of processes is a multiple of the number of processes loaded into the processor.

Spiked

In this data set there are four spikes followed by a short down time, where there are a few smaller processes. This is a realistic environment for a personal computer user. In a personal computer the user typically uses large amounts of processing time (e.g. downloading and displaying a webpage) and then has large amounts of downtime (e.g. reading the webpage). In this downtime for the user the OS user space processes will typically do various smaller maintenance tasks for the system. This data set looks closely at the ability of the scheduling algorithm to handle several large tasks with smaller tasks entering into the system. This is a situation where the Load Top algorithm might succeed as it can move through the list of processes quickly. Other algorithms might have longer wait times as these processes cannot be serviced as quickly.

Long

There are very few processes in this data set. The time is divided up among a very few processes that all start at the same time. The number of processes that will be tested are 1, 2, 4, 8, and 16. This data set is one that is more likely to be found in a research computing environment, where a small set of users issue tasks that can have very long execution times. This data set will have interesting effects on the algorithms who have had the Persistent Data method applied to them. When scheduling exactly the number of processes for which the original algorithm had slots (4), how will the persistent algorithm compare? Even with more processes to continue to create OS calls, the service speed and context switching could become and issue in the overall performance of the system.

All Data Sets

For all of the data sets the percentage of memory calls, percentage of OS calls, frame size, and context-switch time will be varied. The memory call percentage will be varied from 40% to 100% in steps of 10%. This value also includes the amount of ILP in the process, which can be very low in processes that execute serially. This can also be very good, especially with the new breed of compilers being developed for the IA-64 architecture. The OS call percentage will be varied from 0% to 0.05% in steps of 0.01%. Although this percentage may seem unusually low, it is important to remember that this is evaluated at every cycle that the process executes. Five hundredths of a percent and a 200 instruction OS call would mean that ten percent of the instructions would be OS calls. The context switch time is going to be varied from zero cycles to fifteen cycles in steps of five cycles. And the frame size will

have the values of 100, 200, 500, 1000, and 5000 cycles. The smaller frame sizes should decrease the wait time, but will increase the effect of the context switching. This variable will have to be watched closely because it has potential to effect the benefits and downfalls of all the scheduling algorithms that are being modeled.

Short Form

Table 3.2 includes a short description of each of the data sets and quickly looks at the importance of each.

Table 3.2 Comparison of the Data sets.

	Solid	**Simultaneous**	**Spiked**	**Long**
Description	Solid load of processes, one starts as another ends	Large number of processes, all started at the same time	Short bursts of large processes with smaller processes in between	Small numbers of large processes
Shows	Ability of the algorithm to manage processes continuously	How large numbers of processes can increase wait time	Real time environment testing for algorithms	How the processor will handle small number of treads
Algorithms Tested	Provides a comparison of Shift In vs. Load Top in handling large numbers of similarly sized processes	Persistent has the possibility of doing well with the large number of OS calls that will be generated	All the algorithms will be equally tested on this dataset	Traditional processor may win, especially on the lower number of processes
Primary Values Watched	Wait Time, Response Time, Throughput	Wait Time, Total Time,	Throughput, Response Time, CPU Utilization	CPU Utilization, Total Time

Statistics Gathering

After each simulation is run, data needs to be gathered to evaluate the effectiveness of the scheduling algorithm on that data set. These values are the numbers that will be gathered from each of the combinations. This data is gathered by Matlab to be viewed as a graph. These values are taken from Chapter 5 of Silberschatz's book[16] on operating systems.

CPU Utilization

CPU Utilization is a measurement of how much of the overall time the CPU actually spent processing data. That time does not include context-switching or time that is wasted by the processes doing I/O. This is expressed as a percentage of the total time used by the simulation. The design goals of SMT are to increase this utilization so any scheduling algorithm should be consistent with that goal. If the CPU utilization is less than that of the traditional processor, the algorithm is destructive to the SMT. Previous research has shown the effect of adding SMT can be an increase in processor utilization by 52%[17] over single chip multiprocessor systems.

Response Time

The time that the process is put into the OS or ready queue is not the time that

---

[16] Silberschatz, Abraham and Peter Baer Galvin, Operating System Concepts, (Reading, Mass.: Addison Wesly Longman, Inc., 1998).

[17] Lo, Jack L., et al., "Converting Thread_Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," http://www.digital.com/alphaoem/papers/smt-vs.pdf (20 July 1999).

the CPU starts to work on the process. This difference is the response time. The response time would ideally be below one frame size, but practically it will be much higher than that. One would want the response time to be as small as possible, especially for user interactive programs. The data sets, which start significant numbers of processes at the same time will see an increase in response time (not to mention wait time). Response time will decrease in scheduling algorithms that can quickly get to processes when they are added to the queue.

Wait Time

Wait time represents the total amount of time that the process spent waiting in queues. This includes both the OS and the internal processor queues when more than one process is in the processor. The minimum, maximum, and averages are also taken on this value. Because this is the total wait time, even things outside of the control of the algorithm (context switching) are included. The wait time should be related to the context switch time by the number of times the processes gets an opportunity in the processor. The maximum wait time can also increase if some processes are being starved in the processor.

Throughput

Throughput is a measurement of the overall scheduler performance. The throughput measures the number of processes that are sent through the processor per cycle. It is more of a compound statistic in that algorithms that reduce wait time, response time, and turn around time will in turn have a greater throughput. Throughput is more useful as an overall statistic. Typically a significant increase in

the throughput is at the cost of increasing the wait time.

Assumptions

In order to make the model less complex, several assumptions have to be made. Those assumptions are listed here along with an explanation on why it was felt that they were not crucial to the experiment at hand. The assumptions in this research included: a limited process blocking mechanism will not effect results; OS calls can be limited in complexity; and that the round robin scheduling algorithm will be removed by looking at relative data.

Process Blocking

Part of any threaded application is synchronization issues on commonly used data items. Many of the threads themselves end up in a blocked state, waiting for another thread to finish its processing. This synchronization is not handled in this model of the processor. The only processes blocking that is modeled is blocking using the OS call. Although waiting for another process is very different than an OS call in a real system, for this model it is not much different. The OS call blocks the processes, but if it is the only processes in the system they can be scheduled concurrently. For the data sets that are being discussed this situation would be rare (except with the one processes long data set where there is no other processes to be blocked by anyway).

Simplicity of OS call

Real systems have several kinds of calls that can be made to the operating system. While they all vary in length, they all also block the process that they are called by. The OS calls in the model do not represent either of these functions. The OS calls do get put on the top of the OS stack after each time slice that the processor goes through. This in effect makes them finish before any of the other processes in the system get a chance to executed. This gets further augmented by the fact that OS calls cannot make other OS calls, and they always have a 100% memory usage. While this does block the process in the traditional sense, the OS call will probably get processed before the process gets any more time.

Round Robin "divides out"

In the discussion about the traditional process scheduler, it is noted that the OS segment of the model is using a round robin scheduler. This scheduler is not one that is typically used in real systems. To remove this bias from the results, all of them are being analyzed relative to the traditional processor results. This assumption allows the results to be examined without looking at the scheduler in the operating system itself.

Overview

The model created for this research provides a flexible environment for testing the scheduling algorithms discussed previously in this chapter. The four SMT based scheduling algorithms all have different strengths and weaknesses which will allow them to perform well on different data sets. The data sets that have been chosen to

push the algorithms, to find weaknesses, and to represent some real world computing environments. As the statistics are gathered from the model, the data shall show where each algorithm's abilities are limited.

Chapter 4: Data Analysis

After creating the model, the data was generated. This chapter contains analysis of the data that is generated by the model for the various data sets. All of the graphs are scaled relative to the data generated by the traditional processor. This creates odd looking graphs, but removes the bias discussed in Chapter 3. Analysis is done by data set, looking at each of the scheduling algorithms on that data set.

Long

One of the first things that is visible from all of the graphs in the long data set is that all of the SMT scheduling algorithms are better than the traditional processor. This is unusual as it was expected that the traditional processor had potential to do much better than the SMT algorithms on the long data set. It was hypothesized that the traditional processor would do better with the smaller number of processes. What tended to happen is that the OS calls played a greater role on the overall ability of the processor than was originally theorized.

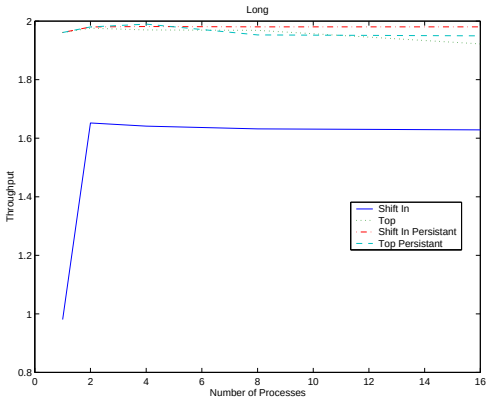

Figure 4.1 Graph showing throughput while varying the number of processes in the data set on the long data set.

One of the places where the large numbers of OS calls made an impact is shown in Figure 4.1. This graph shows the relative throughput of the four SMT algorithms versus the traditional processor. With some of the algorithms, this can be as high as two times the amount of throughput. But with throughput calculated as the number of processes over time, the number of OS processes becomes a significant factor. The significance of the OS call can be seen in Figure 4.2, where the OS call variable is the one that is varied. This graph shows the significance of the OS calls as they are cut out. The throughput of the different SMT scheduling algorithms is very close to that of the traditional processor, even with only eight processes in the system. These graphs show the large impact that the amount of OS calls have on the processor.

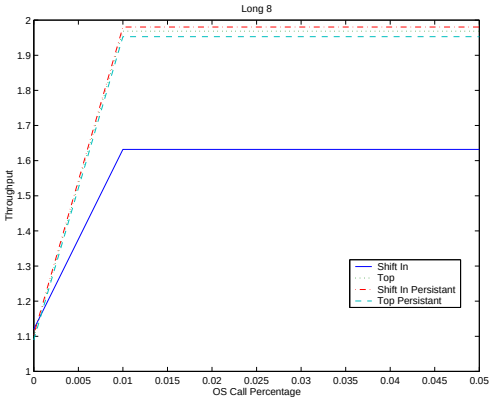

Figure 4.2 Graph showing throughput while varying the chance of an OS call on the long data set with 8 processes.

Another interesting result on these two graphs is the poor performance of the Shift In algorithm. The Shift In algorithm is consistently 30% slower when there are OS calls and more than two processes in the system. This happens at the same time as the Shift In Persistent algorithm dominates throughout the throughput graphs. This

shows that the Shift In algorithm cannot maintain a high throughput with large numbers of processes. Since the majority of the processes are OS calls, the persistent part of the algorithm handles this deficiency with the Shift In algorithm.

The wait times are less affected by the number of OS calls in the system. Both of the Load Top algorithms perform better than the Shift In schedulers. This occurs as the wait times for the OS calls are also calculated. The most interesting part of Figures 4.3 and 4.4 is that the Load Top algorithm had shorter wait times than the Shift In algorithm. The Top algorithm was expected to perform better in this regard and does so through out all of the Long data set. The Shift In persistent algorithm gets closer to the values generated by the Load Top algorithms, especially in Figure 4.4. By allowing the Shift In Persistent algorithm to handle the OS calls quickly the wait time is reduced. This is also show by the Load Top Persistent algorithm outperforming the standard Load Top.

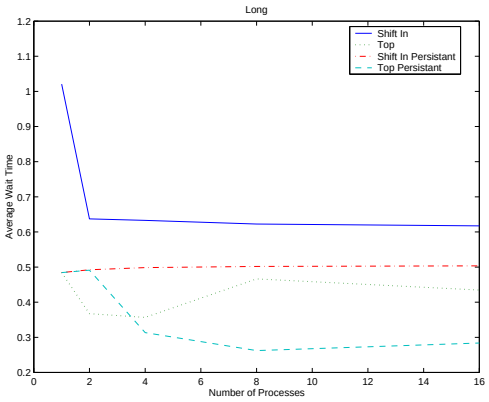

Figure 4.3 Graph showing average wait time while varying the number of processes in the data set on the long data set.

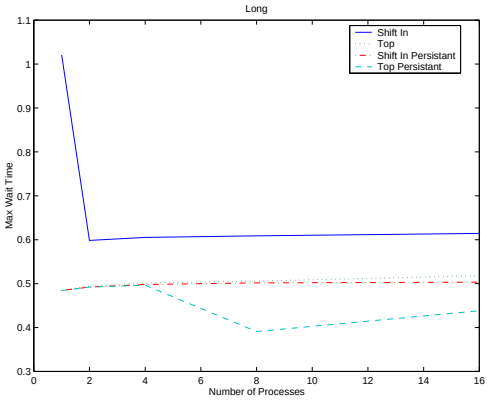

Figure 4.4 Graph showing the maximum wait time varying the number of processes in the data set on the long data set.

Simultaneous

As was hypothesized in Chapter 3, the persistent algorithms did very well on the simultaneous data set. With all of the processes being issued at the same time, the original OS queue is rather large. The problem is that it only continues to grow as further OS calls are made. This is where the persistent algorithms can, and do, shine.

One of the most evident places that this occurs is in the average and maximum wait times. In Figures 4.5 and 4.6 one of the first things that is noticed is how well the persistent algorithms perform. In the Maximum Wait graph the Shift In Persistent algorithm operates at only one half of the maximum for the traditional processor, and the Load Top closer to 45%. How does this happen on the simultaneous data set? This is generated by the Persistent's ability to handle the OS calls quickly. Interestingly, the Load Top algorithm performs well also. In the average wait time it slightly out performs the Shift In Persistent algorithm. The algorithm is able to do this by getting to the processes quickly, and not making them wait as long to receive processing time.

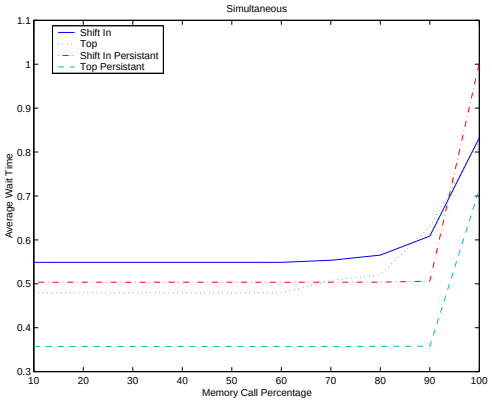

Figure 4.5 Graph showing average wait time while varying the memcall attribute in the simultaneous data set.

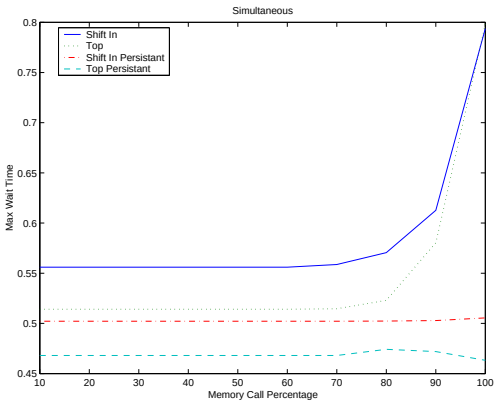

Figure 4.6 Graph showing maximum wait times while varying the memcall attribute on the simultaneous data set.

The throughput for the Simultaneous data set also has interesting results in relation to the memory call percentage. Overall the memory percentage did not affect the relative throughput of the algorithms at lower values of the memory call parameter. When the value gets above 70%, things change. As the parallelism in the processes increases the SMT architecture becomes less useful, so the throughput starts to dive down to the level of the traditional processor. At 100% the nonpersistent SMT algorithms slightly out perform the traditional processor by keeping the extra contexts. As the processes are being sent through the processor fewer context switches are required as several processes contexts are already loaded. Even at the higher levels of the memory call attribute the persistent algorithms are still able to maintain a high throughput. Being able to handle the OS calls quickly and without a context switch helps even in a largely parallel environment.

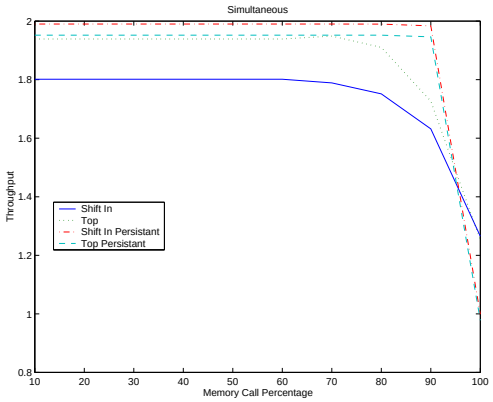

Figure 4.7 Graph showing throughput while varying the percentage of memory calls on the simultaneous data set.

Solid

The solid data set was created to show a constant load on the processor for an extended amount of time. Looking at this data set shows a separation between the Shift In and Top scheduling algorithms emerged, but not always in the most expected ways. For the most part the persistent algorithms performed the best, as the OS calls

play a significant part of any real world situation and this simulation.

One of the more interesting graphs is shown in Figure 4.8. Here the wait time appears to remain constant as the context switch time increases. That is not entirely true. Since all the graphs are relative to the traditional processor case, this graph shows that context switch time affects the average wait time of all of the algorithms the same. The persistent algorithms consistently have lower wait times than the standard scheduling algorithms. This is important with this data set as the processes are coming continuously, and how much time each of them waits can be a significant problem. The persistent algorithms handle the small processes well, without making them consistently wait around in queues. This keeps the average wait time down. Another interesting component of this graph is how close the two persistent algorithms are. When looking at wait time on this data set it does not matter which scheduling algorithm is being used.

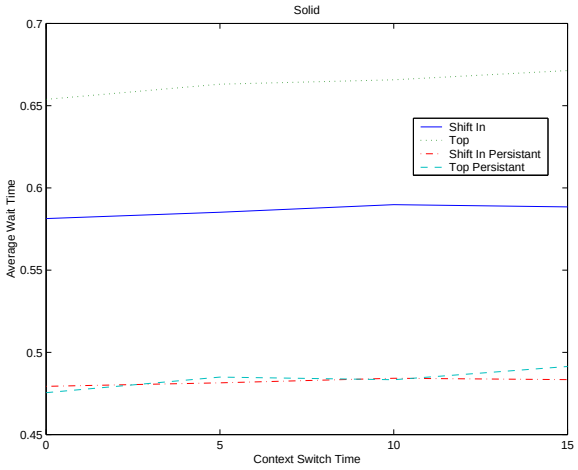

Figure 4.8 Graph showing average wait time while varying context switch time on the solid data set.

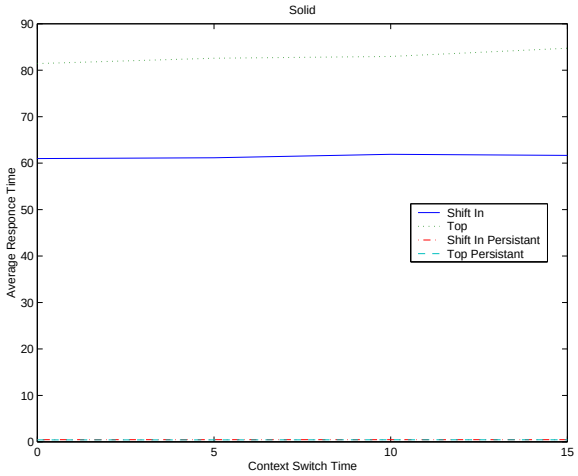

Figure 4.9 Graph showing the average response time while varying context switch time on the solid data set.

The ordering is similar in the next graph, Figure 4.9. This graph shows the response time for the differing processes, and the Top algorithm does not do what was expected, it increases the response time. The reason that the Top algorithm did not do significantly better than the traditional processor on this data set is that there are not many simultaneous processes. This minimizes the benefits that the algorithm brings. An interesting result on these graphs, is how slow to respond the standard algorithms are. It is apparent that the OS calls again are playing a large role in the data. With the smaller numbers of simultaneous processes being able to get to these quickly sets the average response time. Figure 4.10 examines the response time from the perspective of the frame size. This shows the difference between the persistent and standard algorithms start to shrink. The distance drops dramatically between 200 and 100 cycles, insinuating that the OS call size of 200 cycles might have some effect.

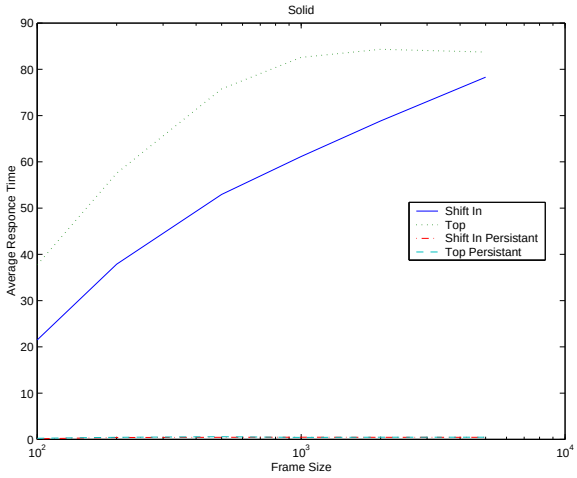

Figure 4.10 Graph showing average wait time while varying the frame size on the solid data set.

Spiked

The spiked data set was created to be the most realistic data set in the series. It was unknown exactly how this data set would work with the relevant scheduling algorithms, and how they would respond to it. All of the scheduling algorithms performed well, the persistent algorithms out performing the “traditional” ones.

Figure 4.11 is the graph of the CPU usage versus the context switch time. This graph shows exactly what was hypothesized. As the context time increases, the Top Select algorithms start to use less and less of the CPU time. This comes from the large number of context switches that both of these algorithms have. As the length of the context switch increases, the amount of productive CPU usage decreases. The two Shift in algorithms are also very close, with the standard Shift In algorithm slightly edging out the persistent one. As the persistent algorithm only has three spots in which processes can be executed, when there are not enough OS calls to fill the last slot some CPU time gets wasted. Most of the time the fourth slot is filled with an OS call, and that is why the two are so close.

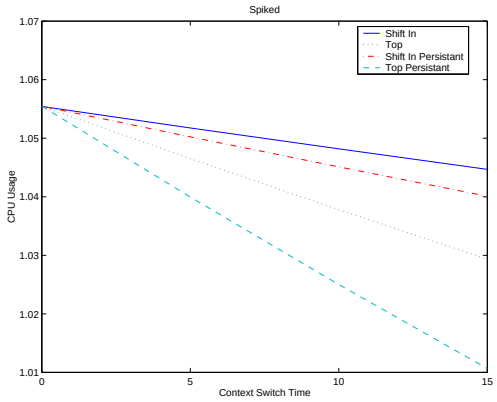

Figure 4.11 Graph showing CPU usage while varying context switch time on the spiked data set.

The throughput curves tell a different story than the CPU utilization. In Figure 4.12 it can be seen that the persistent algorithms maintain a much higher throughput even with increasing context switch time. The Load Top algorithms both have a greater slope than the Shift In algorithms. As the context switch time is increasing the throughput is decreasing for both quickly. This branches

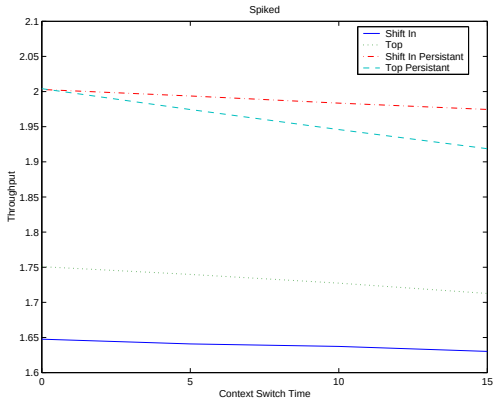

Figure 4.12 Graph showing throughput while varying context switch time on the spiked data set.

from the large number of context switches that these algorithms perform.

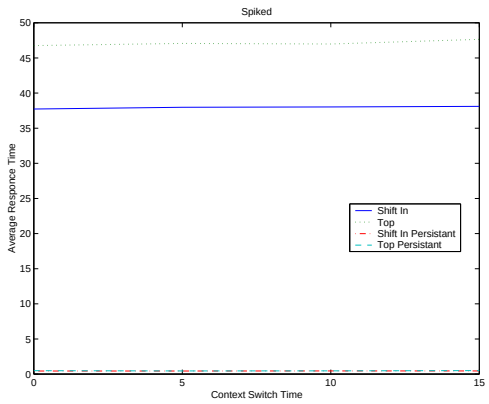

Figure 4.13 Graph showing the average response time while varying the context switch time on the spiked data set.

Real differences between the standard algorithms and the persistent ones on the graph of response time in Figure 4.13. On this graph the standard Shift In and Load Top algorithms look like they are performing very poorly. The problems are created through the OS calls once again. With the amount of OS calls that occur in a given run, they need to be handled in an efficient manner. Persistent algorithms do this. When the OS calls add up, like in the spiked data set, the results can be unbecoming.

Overall

Overall the scheduling algorithms performed consistently across the different data sets. Their performances can be seen in Table 4.1.

Table 4.1 Table of results comparing scheduling algorithms to data sets.

	Long	**Solid**	**Simultaneous**	**Spiked**
Shift In	Performed worst in every category except CPU Usage. Good if you need a heater	Wait time is good, but response time is abysmal. Throughput is also low	Good performance relative to the traditional processor, but the worst SMT algorithm	Achieved the highest CPU usage but fell down on the throughput
Load Top	Did mediocre overall, wait times better than either Shift In algorithm	Wait times were bad and response times were worse	Throughput was high and wait time low, it started to increase wait time early in memory call percentages	CPU usage decreased sharply with context switch time
Shift Persistent	Performed very well in all categories, best in throughput regardless the number of processes	Solid performance, comparable with Top Persistent, slightly edged out in throughput	Wait times were okay, throughput was the highest of all algorithms	Performed with the highest throughput and close to highest CPU usage
Top Persistent	Average performance only wait times were very good, especially relative to the traditional processor	Solid performance, got slightly edged out in throughput	Wait times were excellent but throughput was slightly outperformed by Shift Persistent	Did much better in throughput than CPU usage, but was mediocre overall

Chapter 5: Conclusion

Unfortunately by the nature of scheduling algorithms there is rarely an obvious answer of which algorithm is the best. Typically one algorithm will be better under a certain set of conditions than another. This thesis is not an exception to that pattern. Table 4.1 is an excellent example of how the differing algorithms perform very differently given the type of data set, and even the parameter that is being examined. One algorithm does not dominate response time, nor does one algorithm dominate a particular data set. That does not mean that conclusions are inconceivable.

Jack Lo stated that in the operating system "there may be some threads that are more important than others."[18] In fact he was right, but the important thread is the OS itself. On all of the data sets the persistent algorithms performed very well, most of the time the best. This shows that although a potential thread context is lost, the ability to handle OS calls quickly is of greater benefit. This is best shown in the decreased wait and response times that the persistent algorithms are able to maintain.

Comparing the Load Top and Shift In algorithms is less straight forward. Seeing as the persistent algorithms were shown to be more effective, those two derivatives of Load Top and Shift In should be discussed here. The Shift In algorithm

---

[18] Lo, Jack L., et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," http://www.digital.com/alphaoem/papers/smt-vs.pdf (20 July 1999).

consistently achieved a higher throughput, while the Load Top algorithm has shorter wait and response times. The throughput is a better measure of overall system performance, which would lead to weighing it more heavily in discussion of the algorithms. The Load Top algorithm also has the potential (although remote) for starvation (Load Top, p. 19), which is a concern of implementing any algorithm is a real system. These factors indicate that the Shift In algorithm is a better overall choice for a system with an SMT processor at its core.

Further Research

After showing that one persistent thread is beneficial to the overall system, the next question becomes how about more? If some processes were determined to have high priority in the OS, should those also gain "persistent status." What conditions promote or demote them from this status? With many researchers discussing SMT systems with eight thread contexts this becomes even more of a reality. Another question raised from this research moves into OS design. Currently most OS have a monolithic kernel which handles all of the systems itself; could this be threaded to increase performance? Is there a possibility of a driver having its own thread in the operating system? Does that destroy the effectiveness that the persistent algorithms present in this thesis? The more questions that get answered the more questions that there will always be. While many of these issues have been discussed in research for supercomputers, they still need to be applied to SMT, and different computing environments.

List of References

List of References

Ahuja, Sudhir R. and Abhaya Asthana. "A Multi-Microprocessor Architecture with Hardware Support fro Communication and Scheduling." *Symposium on Architectual Support for Programming Languages and Operating Systems.* Chairman: David R. Ditzel. New York: ACM Press, 1982.

Butler, Michael, et al. "Single Instruction Stream Parallelism Is Greater than Two." *Proceedings of the 18th Annual Symposium on Computer Architecture.* New York: ACM Press, 1991.

Dybvig, R. Kent. *The Scheme Programming Language.* 2nd ed. Upper Saddle River, N.J.: Prentice Hall PTR, 1996.

Eggers, Susan J., et al. "Sumultaneour Multithreading: A Platform for Next-Generation Processors." *IEEE Micro* September/October 1997: 12-19.

Feitelson, Dror G. and Larry Rudolph. "Mapping and Scheduling in a Shard Parallel Environment Using Distributed Hierarchical Control." *1990 International Conference on Parallel Processing.* University Park, Penn.: Pennsylvania State University Press, 1990.

Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* Cambridge, Mass.: The MIT Press, 1992.

Galvin, Peter Baer and Abraham Silberschatz. *Operating System Concepts.* Reading, Mass.: Addison Wesley Longman, 1998.

Hirata, Hiroaki, et al. "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads." *Proceedings of the 19th Annual International Symposium on Computer Architecture.* New York: ACM Press, 1992.

Intel Corporation. *IA-64 Application Developer's Architecture Guide, Rev. 1.0.* Intel Corporation, 1999.

Leutenegger, Scott T. and Mary K. Vernon. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies." *Proceedings of the Conference on Measurement and Modeling of Computer Systems.* New York: ACM Press, 1990.

Lo, Jack L., et al. "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading." http://www.digital.com/alphaoem/papers/smt-vs.pdf

Lo, Jack L., et al. "Tuning Compiler Optimization for Simultaneous Multithreading."

http://www.cs.washington.edu/research/smt/

Patterson, David A. and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, Calif.: Morgan Kaufmann Publishers, 1994.

Saavedra-Barrera, Rafael H., David E. Culler and Thorsten von Eicken. "Analysis of Multithreaded Architectures for Parallel Computing." *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithims and Architectures.* New York: ACM Press, 1990.

Tullsen, Dean M., Susan J. Eggers and Henry M. Levy. "Retrospective: Simultaneous Multithreading: Maximizing On-Chip Parallelism." *25 Year of the International Symposia on Computer Architecture.* Ed. Gurinder Sohi. New York: ACM Press, 1998.

Tullsen, Dean M., Susan J. Eggers and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." http://www.cs.washington.edu/research/smt/

Tullsen, Dean M., et al. "Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor." http://www.cs.washington.edu/research/smt

Waldspurger, Carl A. and William E. Weihl. "Register Relocation: Flexible Contexts

for Multithreading." *Proceedings of the 20th Annual Symposium on Computer Architecture.* New York: ACM Press, 1993.

Zahorjan, John and Cathy McCann. "Processor Scheduling in Shared Memory Multiprocessors." *Proceedings of the Conference on Measurement and Modeling of Computer Systems.* New York: ACM Press, 1990.

Appendices

Appendix A: Scheme Source Code

Description of Code

When executing the Scheme code all that is required to execute is `run.ss`. This file loads the variables, and all of the other source code into Chez Scheme to be executed. The `variables.ss` file is automatically generated, but it contains the data set parameters, and which data set and scheduler to use. The files that start with `dataset_` are all scheme source code that defines a variable called `dataset` using the data set variables. All of the files that start with `schedule_` are the different schedulers. Each defines two functions: `schedule` and `schedule-realtime`. These two functions are called by the code in `core.ss`. `core.ss` contains the basis of all of the simulation loops and the functions that actually execute the simulation. When the simulation completes `display.ss` is executed to calculate and display all of the statistics on the run of the model.

core.ss

```
(define expire-list '()) ; where all of the processes go when they are
done.
(define used-time 0) ; the ammount of time that is used in actual
processing by the processesor

(define run-block
; This function takes a set of processes and runs through them giving
; them as much time as they will take.
; if they expire they will be put in the expire-list
   (lambda (numcycle proclst)
      (if (or (<= numcycle 0) (null? proclst))
         numcycle
         (let*
            ((executecyc
               (let
                  ((numcyc (floor (* (/ ((process-memcall (car
proclst))) 100) numcycle))))
                     (if (> numcyc numcycle) numcycle numcyc)
                     ))
             (runengret ((process-engine (car proclst))
```

```
                (cycles->enginetime executecyc)
                (lambda (ticks value) value)
                (lambda (x) x)
                )
             )
          )
          (if (list? runengret)
             (begin
                (set-process-output! (car proclst) runengret)
                (set-process-engine! (car proclst) 'elvisrocks)
                (set! expire-list (cons (car proclst) expire-list))
                )
             (set-process-engine! (car proclst) runengret)
             )
          (run-block (- numcycle executecyc) (schedule-realtime
proclst))
          )
       )
    )
 )

; (define processes_running '())

(define simulate
; This is the main function for the whole program, it simulates the
; core of the processes, in it's first run (ie, actual users interface)
; it should probably be run with the second and third arguements as
; null lists.  When it completes it will return the symbol 'done' and
; have all of the processes originally passed in in the expire-list
   (lambda (unrun-in-proc running-proc cpu-in-proc)
      (if (and (null? unrun-in-proc) (null? running-proc) (null?
cpu-in-proc)) 'done
         (let*
            ((tempvar (start-proc global-time unrun-in-proc))
             (run-t-proc (append running-proc (cadr tempvar)))
             (urun-proc (car tempvar))
             (tempvartoo (schedule run-t-proc cpu-in-proc))
             (run-proc (car tempvartoo))
             (cpu-proc (cadr tempvartoo))
             (execution-time (- frame-size (/ (run-block (* frame-size
4) cpu-proc) 4.0)))
             )
            (set! processes_running (cons (+ (length run-proc) (length
cpu-proc)) processes_running))
            (set! global-time (+ global-time frame-size))
            (set! used-time (+ used-time execution-time))
            (set! os-stack (prune-expired os-stack))
            (simulate urun-proc (prune-expired run-proc) (prune-expired
cpu-proc))
            )
         )
      )
   )

(define prune-expired
   (lambda (proclst)
      (letrec
         ((helper
            (lambda (proclst retlst)
               (if (null? proclst) (reverse retlst)
                  (if (eq? (process-engine (car proclst)) 'elvisrocks)
                     (helper (cdr proclst) retlst)
                     (helper (cdr proclst) (cons (car proclst) retlst))
```

```
                    )
                 )
              )
           ))
        (helper proclst '())
        )
     )
  )

;(define process_launch '())

(define start-proc
; This function figures out what processes should be started and returns
; them, along with the new list of unstarted processes.
   (lambda (in-time proclst)
      (letrec
         ((helper
            (lambda (proclst retlst)
               (if (null? proclst) (list '() retlst)
                  (if (< (process-starttime (car proclst)) in-time)
                     (begin
                        (set-process-engine! (car proclst)
(process->engine (car proclst)))
                        (helper (cdr proclst) (cons (car proclst)
retlst))
                        )
                     (list proclst retlst)
                     )
                  )
               )
            ))
         (let
            ((retlst (helper proclst '())))
;           (set! process_launch (cons (length (cadr retlst))
process_launch))
            retlst
            )
         )
      )
   )
```

dataset_long1.ss

```
(define dataset
   (make-data-set 0 0 1 0 10000000 memory-calls os-calls os-length))
```

dataset_long2.ss

```
(define dataset
   (make-data-set 0 0 2 0 5000000 memory-calls os-calls os-length))
```

dataset_long4.ss

```
(define dataset
   (make-data-set 0 0 4 0 2500000 memory-calls os-calls os-length))
```

dataset_long8.ss

```
(define dataset
   (make-data-set 0 0 8 0 1250000 memory-calls os-calls os-length))
```

dataset_long16.ss

```
(define dataset
   (make-data-set 0 0 16 0 625000 memory-calls os-calls os-length))
```

dataset_simultaneous.ss

```
(define dataset
   (make-data-set 0 0 1000 0 10000 memory-calls os-calls os-length))
```

dataset_solid.ss

```
(define dataset
   (make-data-set 0 0 1000 10000 10000 memory-calls os-calls
os-length))
```

dataset_spiked.ss

```
(define make-spike
   (lambda (start startid size)
      (append
         (make-data-set start startid 100 10 18000 memory-calls
os-calls os-length)
         (make-data-set (+ 18000 start) (+ startid 100) 100 17000 2000
memory-calls os-calls os-length)
         )
      )
   )

(define dataset
   (append
      (make-spike 0 0 2000000)
      (make-spike 2000000 201 2000000)
      (make-spike 4000000 402 2000000)
      (make-spike 6000000 603 2000000)
      (make-spike 8000000 804 2000000)
      )
   )
```

display.ss

```
(define average
     (lambda (lst)
          (/ (apply + lst) (length lst))
          )
     )

(define process-turnarround
     (lambda (proc)
          (- (process-finishtime proc) (process-starttime proc))
          )
     )

(define process-finishtime
     (lambda (proc)
          (+ (caar (process-output proc)) (cadar (process-output proc)))
          )
     )

(define process-realstarttime
     (lambda (proc)
          (caar (reverse (process-output proc)))
          )
```

```
     )

(define process-responcetime
     (lambda (proc)
          (- (process-realstarttime proc) (process-starttime proc))
          )
     )

(define process-waittime
     (lambda (proc)
          (letrec
               ((helper
                    (lambda (last next accum)
                         (if (null? next) accum
                              (helper (car next) (cdr next)
                                   (+ accum (- (caar next) (+ (- (cadr
last) (cadar next)) (car last))))
                                   )
                              )
                         )
                    )
                )
               (helper (car (reverse (process-output proc))) (cdr
(reverse (process-output proc))) '0)
               )
          )
     )

(printf "[")
(printf "~a, " (exact->inexact (* (/ used-time global-time) 100)))
(printf "~a, ~a, ~a, " (apply min (map process-turnarround expire-list))
(average (map process-turnarround expire-list)) (apply max (map
process-turnarround expire-list)))
(printf "~a, ~a, ~a, " (apply min (map process-responcetime
expire-list)) (average (map process-responcetime expire-list)) (apply
max (map process-responcetime expire-list)))
(printf "~a, ~a, ~a, " (apply min (map process-waittime expire-list))
(average (map process-waittime expire-list)) (apply max (map
process-waittime expire-list)))
(printf "~a, " (exact->inexact (/ (length expire-list) global-time)))
(printf "~a" (- (length expire-list) (length dataset)))

(define print-sequencial
   (lambda (lst)
      (if (null? lst) '()
         (begin
            (printf "~a, " (car lst))
            (print-sequencial (cdr lst))
            )
         )
      )
   )

(printf "]~%")

(printf "[")
(printf "5")
;(print-sequencial (reverse processes_running))
(printf "]~%[")
(printf "5")
;(print-sequencial (reverse process_launch))
(printf "]~%")
```

load.ss

```
(load "record-def.so")
(load "os.so")
(load "proc-run.so")
(load "core.so")
```

os.ss

```
(define os-stack '())

(define os-makecall
     (lambda (number-o-cycles)
          (let
               ((bob (make-process 0 0 number-o-cycles -1 '() '()
(lambda () '100) global-time)))
               (set-process-engine! bob (process->engine bob))
               (set! os-stack (append os-stack (list bob)))
               )
          )
     )
```

proc-run.ss

```
(define global-time '0) ; the 'time' it is to everyone

(define run-func
; This is the function that processes run to continue going.  This is
; the function that checks on all of the OS calls.
     (lambda (numcycles lasttime retlst inprocess)
          (if (eq? numcycles 0)
               retlst
               (if (< (random 100) (process-oscall inprocess))
                    (begin
                         (os-makecall (process-oscalllength inprocess))
                         (engine-block)
                         (run-func (- numcycles 1) lasttime retlst
inprocess)
                         )
                    (run-func (- numcycles 1) global-time
                              (if (eq? lasttime global-time)
                                   retlst
                                   (cons (list global-time numcycles)
retlst)
                                   )
                              inprocess
                              )
                    )
               )
          )
     )

(define process->engine
; This takes in a process record and then creates it's engine to be
; run by the simulation
     (lambda (proc)
          (make-engine
               (lambda ()
                    (run-func
                         (process-cycles proc)
                         '-1
                         '()
                         proc
                         )
```

```
                    )
               )
          )
     )

(define cycles->enginetime
; Because run-fun isn't just one cycle this is the conversion factor
; from 'logical' units to 'engine units' that are used by scheme.
     (lambda (cyc)
          (* cyc 28)
          )
     )
```

record-def.ss

```
(define-record process (
          oscall ; the percent of time the function makes os calls
          oscalllength ; the number of cycles for the average os call
          cycles ; the number of cycles the process will use
          id ; this is an inique id number
          output ; the return of the whole thing
          engine ; this is what is run
          memcall ; a function to return...
          starttime ; the time the proc starts
          )
     )

(define make-data-set
     (lambda (
          start_time     ; The time that they start generating from
          start_id       ; The first id number to use
          proc-num       ; number of procs to generate
          proc-space     ; the spacing between them
          run-length     ; average running time +- 10%
          mem-usage ; again +- 10%
          os-call        ; again +- 10%
          os-call-l ; length of said os calls
          )
          (letrec
               ((helper (lambda (curtime proc-num cur-id retlst)
                              (if (eq? 0 proc-num) (reverse retlst)
                                   (helper
                                        (+ curtime proc-space)
                                        (- proc-num 1)
                                        (+ cur-id 1)
                                        (cons
                                             (let
                                                  ((runt (+ run-length
(* run-length (/ (- (random 20) 10) 100)))))
                                                  (make-process
                                                       os-call
                                                       os-call-l
                                                       runt
                                                       cur-id
                                                       'null
                                                       'null
                                                       (lambda () (+
mem-usage (* mem-usage (/ (- (random 20) 10) 100))))
                                                       curtime
                                                       )
                                                  )

                                             retlst)
```

```
                                        )
                                   )
                              )
                         ))
                    (helper start_time proc-num start_id '())
                    )
               )
          )
```

run.ss

```
(load "load.so")
(load "/tmp/ted.variables.ss")
(load scheduler)
(load dataset)
(simulate dataset '() '())
(load "display.so")
(exit)
```

schedule_one.ss

```
; need defined ahead of time:
; proccess
; context-switch-time
; global-time
; proc-os

; This implements shifting in the processes.

(define schedule
; uses os-stack
     (lambda (os_proc prossessor_proc)
          (let*
               ((myos (append os_proc prossessor_proc))
                )
               ; (printf "schedule: ~a ~a ~a~%" (length os_proc) (length
prossessor_proc) (length os-stack))
               (if (null? os-stack)
                    (begin
                         (if (null? os_proc) '() (set! global-time (+
global-time context-switch-time)))
                         (if (null? myos)
                              (list '() '())
                              (list (cdr myos) (list (car myos)))
                              )
                         )
                    (let
                         ((temp (car os-stack)))
                         (set! os-stack (cdr os-stack))
                         (set! global-time (+ global-time
context-switch-time))
                         (list myos (list temp))
                         )
                    )
               )
          )
     )

; This leaves the lists as they are.  Normally it could stop for OS
; calls or other things, but it isn't.
(define schedule-realtime
     (lambda (proc-list)
          (cdr proc-list)
```

```
          )
     )
```

schedule_shift.ss

```
; This scheduling program goes through and shifts in the processes as
; they become available

(define shift-last -10)

(define schedule
     (lambda (os_proc processor_proc)
          (if (and (null? os_proc) (null? processor_proc))
               (list '() '())
               (let
                    ((proclst (append processor_proc os_proc))
                     (origid (sort < (map process-id processor_proc)))
                     )
                    (let
                        ((proclst
                             (if (eq? (process-id (car proclst))
shift-last)
                                  (append os-stack (cdr proclst) (list
(car proclst)))
                                  (append os-stack proclst)
                                 )
                             ))
                    (set! shift-last (process-id (car proclst)))
                    ;(set! proclst (append os-stack proclst))
                    (set! os-stack '())
                    (letrec
                         ((helper
                              (lambda (proclst retlst number)
                                   (if (null? proclst)
                                        (list '() (reverse retlst))
                                        (if (eq? number 0)
                                             (list proclst (reverse
retlst))
                                             (helper (cdr proclst) (cons
(car proclst) retlst) (- number 1))
                                             )
                                        )
                                   )
                              ))
                         (letrec
                              ((retlst (helper proclst '() 4))
                               (countdiff
                                   (lambda (number lsta lstb)
                                        (if (or (null? lsta) (null?
lstb)) number
                                             (if (eq? (car lsta) (car
lstb))
                                                  (countdiff (- number
1) (cdr lsta) (cdr lstb))
                                                  (if (< (car lsta) (car
lstb))
                                                       (countdiff number
(cdr lsta) lstb)
                                                       (countdiff number
lsta (cdr lstb))
                                                       )
                                                  )
                                             )
```

```
                                        )
                                   )
                              )
                              (set! global-time (+ global-time (*
context-switch-time (countdiff 4 origid (sort < (map process-id (cadr
retlst)))))))
                              retlst
                              )
                         )
                         )
                    )
               )
          )
     )

(define schedule-realtime
     (lambda (proc-list)
          (cdr proc-list)
          )
     )
```

schedule_shift_persist.ss

```
; This scheduling program goes through and shifts in the processes as
; they become available

(define shift-last -10)

(define removenegones
     (lambda (lst)
          (letrec
               ((helper
                    (lambda (lst retlst)
                         (if (null? lst) (reverse retlst)
                              (if (eq? -1 (car lst))
                                   (helper (cdr lst) retlst)
                                   (helper (cdr lst) (cons (car lst)
retlst))
                                   )
                              )
                         )
                    ))
               (helper lst '())
               )
          )
     )

(define schedule
     (lambda (os_proc processor_proc)
          (if (and (null? os_proc) (null? processor_proc))
               (list '() '())
               (let
                    ((proclst (append processor_proc os_proc))
                     (origid (sort < (map process-id processor_proc)))
                     )
                    (let
                        ((proclst
                             (if (eq? (process-id (car proclst))
shift-last)
                                  (append os-stack (cdr proclst) (list
(car proclst)))
                                  (append os-stack proclst)
                                  )
```

```
                              ))
                         (set! shift-last (process-id (car proclst)))
                         ;(set! proclst (append os-stack proclst))
                         (set! os-stack '())
                         (letrec
                              ((helper
                                   (lambda (proclst retlst number)
                                        (if (null? proclst)
                                             (list '() (reverse retlst))
                                             (if (eq? number 0)
                                                  (list proclst (reverse
retlst))
                                                  (helper (cdr proclst) (cons
(car proclst) retlst) (- number 1))
                                                  )
                                             )
                                        )
                                   ))
                              (letrec
                                   ((retlst (helper proclst '() 3))
                                    (countdiff
                                        (lambda (number lsta lstb)
                                             (if (or (null? lsta) (null?
lstb)) number
                                                  (if (eq? (car lsta) (car
lstb))
                                                       (countdiff (- number
1) (cdr lsta) (cdr lstb))
                                                       (if (< (car lsta) (car
lstb))
                                                            (countdiff number
(cdr lsta) lstb)
                                                            (countdiff number
lsta (cdr lstb))
                                                            )
                                                       )
                                                  )
                                             )
                                        )
                                    )
                                   (set! global-time (+ global-time (*
context-switch-time (countdiff 4 origid (sort < (map process-id (cadr
retlst)))))))
                                   retlst
                                   )
                              )
                              )
                         )
                    )
               )
          )


(define schedule-realtime
     (lambda (proc-list)
          (if (null? os-stack)
               (cdr proc-list)
               (let
                    ((ret (append os-stack proc-list)))
                    (set! os-stack '())
                    ret
                    )
               )
```

```
          )
     )
```

schedule_top.ss

```
; This is a scheduling algorithim that always puts the top four
; processes in the os que in to the processor.

(define schedule
     (lambda (os_proc processor_proc)
          (letrec
               ((myos (append os-stack (append os_proc processor_proc)))
                (origid (sort < (map process-id processor_proc)))
                (helper
                    (lambda (proclst retlst number)
                         (if (null? proclst)
                              (list '() (reverse retlst))
                              (if (eq? number 0)
                                   (list proclst (reverse retlst))
                                   (helper (cdr proclst) (cons (car
proclst) retlst) (- number 1))
                                   )
                              )
                         )
                    ))
               ; (printf ".")
               (set! os-stack '())
               (letrec
                    ((retlst (helper myos '() 4))
                     (countdiff
                         (lambda (number lsta lstb)
                              (if (or (null? lsta) (null? lstb)) number
                                   (if (eq? (car lsta) (car lstb))
                                        (countdiff (- number 1) (cdr
lsta) (cdr lstb))
                                        (if (< (car lsta) (car lstb))
                                             (countdiff number (cdr
lsta) lstb)
                                             (countdiff number lsta (cdr
lstb))
                                             )
                                        )
                                   )
                              )
                         )
                     )
                    (set! global-time (+ global-time (*
context-switch-time (countdiff 4 origid (sort < (map process-id (cadr
retlst)))))))
                    retlst
                    )
               )
          )
     )

(define schedule-realtime
     (lambda (proc-list)
          (cdr proc-list)
          )
     )
```

schedule_top_persist.ss

```
; This is a scheduling algorithim that always puts the top four
; processes in the os que in to the processor.

(define removenegones
     (lambda (lst)
          (letrec
               ((helper
                    (lambda (lst retlst)
                         (if (null? lst) (reverse retlst)
                              (if (eq? -1 (car lst))
                                   (helper (cdr lst) retlst)
                                   (helper (cdr lst) (cons (car lst)
retlst))
                                   )
                              )
                         )
                    ))
               (helper lst '())
               )
          )
     )

(define schedule
     (lambda (os_proc processor_proc)
          (letrec
               ((myos (append os-stack (append os_proc processor_proc)))
                (origid (sort < (map process-id processor_proc)))
                (helper
                    (lambda (proclst retlst number)
                         (if (null? proclst)
                              (list '() (reverse retlst))
                              (if (eq? number 0)
                                   (list proclst (reverse retlst))
                                   (helper (cdr proclst) (cons (car
proclst) retlst) (- number 1))
                                   )
                              )
                         )
                    ))
               ; (printf ".")
               (set! os-stack '())
               (letrec
                    ((retlst (helper myos '() 3))
                     (countdiff
                         (lambda (number lsta lstb)
                              (if (or (null? lsta) (null? lstb)) number
                                   (if (eq? (car lsta) (car lstb))
                                        (countdiff (- number 1) (cdr
lsta) (cdr lstb))
                                        (if (< (car lsta) (car lstb))
                                             (countdiff number (cdr
lsta) lstb)
                                             (countdiff number lsta (cdr
lstb))
                                             )
                                        )
                                   )
                              )
                         )
                     )
                    (set! global-time (+ global-time (*
context-switch-time (countdiff 4 (removenegones origid) (removenegones
(sort < (map process-id (cadr retlst)))))))))
```

```
                    retlst
                    )
               )
          )
     )


(define schedule-realtime
     (lambda (proc-list)
          (if (null? os-stack)
               (cdr proc-list)
               (let
                    ((ret (append os-stack proc-list)))
                    (set! os-stack '())
                    ret
                    )
               )
          )
     )
```

variables.ss

```
(define frame-size 1000)
(define context-switch-time 10)

(define memory-calls 100)
(define os-calls 1.000000e-02)
(define os-length 200)

(define scheduler "schedule_top.so")
(define dataset "dataset_simultaneous.so")
```

Appendix B: Matlab Source Code

Description of Code

The Matlab source code is all based around `makegraph`. This is the function that most users of this code will call to get the data that they need. The `makegraph` function coordinates the data creation process and cycles through the different scheduling algorithms. The `getdataset` and `getscheduler` function take in an integer and return the textual representation of the data set or scheduler respectively. In the `getgraph` function takes a set of parameters, and then builds the `variables.ss` file for the simulation being run (a sample is in Appendix A). It the runs the `getdata.m` script which pulls the data out of the Scheme code into Matlab. The data are returned to `makegraph`, which creates a graph with labels and a title for the user.

getdata.m

```
[s, r] = unix('scheme run.ss |
/afs/rose-hulman.edu/users/class00/gouldtj/bin/tail -3 >
/tmp/ted.tempfile.matlab');
[s, r] = unix('/afs/rose-hulman.edu/users/class00/gouldtj/bin/head -1
/tmp/ted.tempfile.matlab');
stats = eval(r);
[s, r] = unix('/afs/rose-hulman.edu/users/class00/gouldtj/bin/tail -2
/tmp/ted.tempfile.matlab |
/afs/rose-hulman.edu/users/class00/gouldtj/bin/head -1');
run = eval(r);
[s, r] = unix('/afs/rose-hulman.edu/users/class00/gouldtj/bin/tail -1
/tmp/ted.tempfile.matlab');
launch = eval(r);
```

getdataset.m

```
function name = getdataset(number)

   switch number
      case 1,
         name = 'dataset_long1.so';
      case 2,
         name = 'dataset_long2.so';
```

```
      case 4,
         name = 'dataset_long4.so';
      case 8,
         name = 'dataset_long8.so';
      case 16,
         name = 'dataset_long16.so';
      case 6,
         name = 'dataset_simultaneous.so';
      case 7,
         name = 'dataset_solid.so';
      case 5,
         name = 'dataset_spiked.so';
      end
```

getgraph.m

```
function graphdata = getgraph(dataset, memory_calls, os_calls,
context_switchs, frame_sizes, schedulers, datatype)
%     function getgraph
%
%  This function graphs scheduling data for the Ted Gould
% SMT scheduler.  The six parameters (yes six) are mostly values
% except for the first and last which are these values:
%    dataset:     1 - dataset_long1.ss
%                 2 - dataset_long2.ss
%
%    scheduler:   1 - schedule_one.ss
%                 2 - schedule_shift.ss
%

numberofentries = length(dataset) * length(memory_calls) *
length(os_calls) * length(context_switchs) * length(frame_sizes) *
length(schedulers);
graphdata = zeros(1, numberofentries);
x = 1;

for ds = dataset
for mc = memory_calls
for oc = os_calls
for cs = context_switchs
for fs = frame_sizes
for sc = schedulers

   varfp = fopen('/tmp/ted.variables.ss', 'w');
   fprintf(varfp, '(define frame-size %d)\n', fs);
   fprintf(varfp, '(define context-switch-time %d)\n\n', cs);
   fprintf(varfp, '(define memory-calls %d)\n', mc);
   fprintf(varfp, '(define os-calls %d)\n', oc);
   fprintf(varfp, '(define os-length %d)\n\n', 200);
   fprintf(varfp, '(define scheduler \"%s\")\n', getscheduler(sc));
   fprintf(varfp, '(define dataset \"%s\")\n', getdataset(ds));
   fclose(varfp);

   getdata;

   graphdata(x) = stats(datatype);
   x = x + 1;

end % sc
end % fs
end % cs
end % oc
end % mc
```

```
end % ds

bob = 5;
```

getscheduler.m

```
function name = getscheduler(number)

switch number
   case 1,
      name = 'schedule_one.so';
   case 2,
      name = 'schedule_shift.so';
   case 3,
      name = 'schedule_top.so';
   case 4,
      name = 'schedule_shift_persist.so';
   case 5,
      name = 'schedule_top_persist.so';
   end
```

makegraph.m

```
function bob = getgraph(dataset, memory_calls, os_calls,
context_switchs, frame_sizes, datatype)

if (length(dataset) > 1)
   varied = dataset;
   xlab = 'Number of Processes';
   end
if (length(memory_calls) > 1)
   varied = memory_calls;
   xlab = 'Memory Call Percentage';
   end
if (length(os_calls) > 1)
   varied = os_calls;
   xlab = 'OS Call Percentage';
   end
if (length(context_switchs) > 1)
   varied = context_switchs;
   xlab = 'Context Switch Time';
   end
if (length(frame_sizes) > 1)
   varied = frame_sizes;
   xlab = 'Frame Size';
   end

datafile = zeros(5, length(varied));

for i = 1:5
   datafile(i, :) = getgraph(dataset, memory_calls, os_calls,
context_switchs, frame_sizes, i, datatype);
   end

figure;
;plot(varied, datafile(1, :), varied, datafile(2, :), varied,
datafile(3, :), varied, datafile(4, :), varied, datafile(5, :));
;legend('Traditional', 'Shift In', 'Top', 'Shift In Persistent', 'Top
Persistant', 0);
plot(varied, datafile(2, :)./datafile(1,:), '-', varied, datafile(3,
:)./datafile(1,:), ':', varied, datafile(4, :)./datafile(1,:), '-.',
varied, datafile(5, :)./datafile(1,:), '--');
```

```
legend('Shift In', 'Top', 'Shift In Persistant', 'Top Persistant', 0);
xlabel(xlab);

switch datatype
   case 1,
      ylabel('CPU Usage');
   case 2,
      ylabel('Min Turnarround Time');
   case 3,
      ylabel('Average Turnarround Time');
   case 4,
      ylabel('Max Turnarround Time');
   case 5,
      ylabel('Min Responce Time');
   case 6,
      ylabel('Average Responce Time');
   case 7,
      ylabel('Max Responce Time');
   case 8,
      ylabel('Min Wait Time');
   case 9,
      ylabel('Average Wait Time');
   case 10,
      ylabel('Max Wait Time');
   case 11,
      ylabel('Throughput');
   case 12,
      ylabel('Number of OS Calls');
   end


bob = datafile;
```