Adaptively Scheduling Processes on a Simultaneous Multithreading Processor

Omer Zaki Matthew McCormick Jonathan Ledlie
{ozaki, mattmcc, ledlie} @cs.wisc.edu

Computer Sciences Department
1210 West Dayton St.
Madison, WI 53706, U.S.A.

Abstract

Within recent years the concept of the simultaneous multithreading (SMT) processor has been gaining in popularity. This hardware allows multiple processes to run on the processor at the same time providing more potential for instruction level parallelism. These new processors suggest that the rules an operating system (OS) scheduler follows need to be changed or at least modified. Our study shows the combination of jobs selected to run on these threads can significantly affect system performance. Our research shows that scheduling policies are greatly affected by the system workload and there most likely does not exist a single, best scheduling policy. However, it can be shown that a scheduler that tries to schedule processes doing a large number of loads and stores together with jobs doing few loads and stores consistently performs at levels close to or better than all other scheduling policies examined. It can also be seen that the more possibilities there are for scheduling, the more necessary it is to have an intelligent scheduler. In contrast, the few number of decisions to make (few threads and/or few processes) the less important the decision of a scheduler becomes.

1 Introduction

Simultaneous Multithreading (SMT) processors enable more than one process to have instructions in flight at the same instant [Tullsen96]. Given that the maximum number of processes which can be run is built in to the hardware's implementation, this begs the question: what policy should be used to pick the processes to run together? Because different processes have different resource requirements, intuition would suggest that if we can find processes that are not using the same resources they will conflict the least, they will be able to get the most instructions through the pipeline, and, as a consequence, we will achieve higher overall instructions per cycle. We show that if a mixture of resource−using jobs is available and if we minimize the structural conflicts through scheduling, we obtain a small but consistent gain in overall IPC over a less intelligent policy.

The essential concept behind SMT is that only so much parallelism can be extracted from a given process but, if more processes contribute their instructions to the pool from which the processor can choose, more parallelism and, therefore, more throughput, will result.

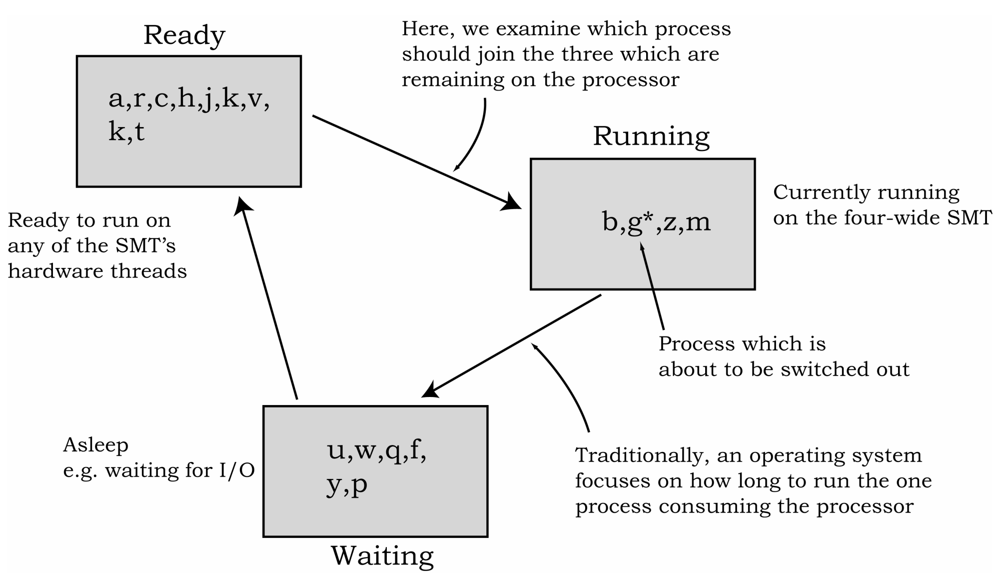
In an SMT, other processes' instructions can flow around this stalled one. It is much less likely that interprocess dependencies exist between some process B and A, than the intraprocess dependency just witnessed.

Two or more ***processes*** can conflict with one another if they are attempting to perform the same kind of instruction at the same time. For example, many concurrent loads and stores or ALU operations can clog these particular parts of the processor whether they come from one process or several. This project seeks to ameliorate these structural bottlenecks by running processes which have different resource demands at the same time.

Before proceeding into how this scheduling is distinct from traditional operating systems scheduling, let us establish our nomenclature. For the purposes of this paper, a ***process*** or ***job*** is a software construct with long–term state; the distinction between lightweight software threads and processes is ignored. Here, ***threads*** refer to hardware contexts, of which a processor has a fixed number (1, 2 and 4 in our tests). Processes run on threads for a short period called a quantum. While each process is running, hardware counters track each process's resources usage. After its quantum expires, software retains this state. Additionally, our schedulers are able to record and associate hardware state like floating point usage with a particular process. This is further explained in Sections 3 and 4.

Operating systems traditionally handle process scheduling by concentrating on the amount of time a process is allowed to stay on a processor. The system keeps a history of the job's behavior and tries to give it a quantum just long enough to accomplish what it needs to and no more. Generally, it does not track what kinds of instructions the process is executing, only how long it demands to use the processor. For example, a job which has, in its recent past, only needed the processor to set up a new I/O request after which it relinquishes the processor, is given higher priority than a seldom–yielding, computationally intensive process.

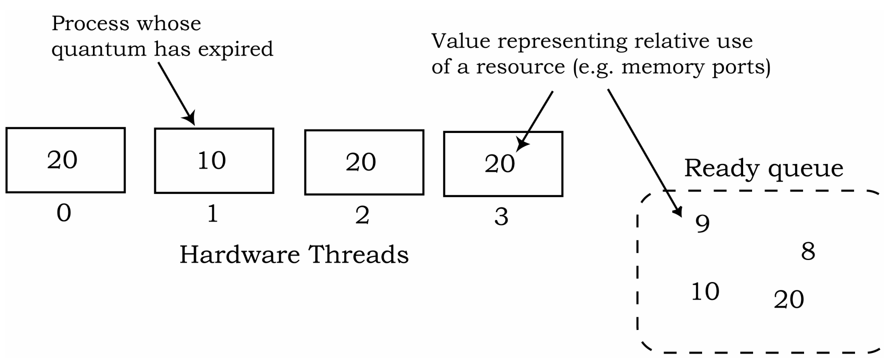
The concept is simple: because these processes can quickly set up their work and then can go to sleep, waiting for the next keystroke or disk interrupt, for example, they should be given higher priority because their work can be done ***in parallel*** with a processor intensive job. This shortest job first policy leads to higher throughput than some schedule like round robin. Operating system schedulers generally also include measures to allow for other means of prioritization and to prevent starvation. In the figure below, the foci are when to remove a process and how to allow short jobs with small quanta to go first.



On an SMT, the problem changes considerably. Here, an OS must not only determine which processes should run first and for how long, but also which run best together. In our study we have ignored the former problem and instead focused exclusively on the latter. We have developed and evaluated three new ways for scheduling on an SMT and compared them with each other and with a random and round robin scheduler.

1.1 A Simple Example

To form a more concrete image in the reader's mind, we would like to go through a simple example. Imagine that there is some CPU resource whose usage can be enumerated and compared to that of other processes. Examples of such resources include any resource that could potentially become a bottleneck: the result bus, load and store queues, any or all of the integer or floating point units, the register file. Because we are not varying the amount of time a process lives on the CPU, each thread is relieved of its process in round robin fashion. As portrayed in the picture below, thread 1 has reached the end of its quantum. The other three threads will continue to run their resource intensive (20 each) processes. The advantage of switching only one process at a time is that it lessens the deleterious impact of saving the process's registers and squashing its in–flight instructions [Tullsen00]. Thread 1 is the victim and we must choose from the processes which have consumed 9, 10, 8, or 20 units of the resource during their last quanta.



Given that the process which is using 10 units of the resource's quantum has expired, which process would you choose to make the average usage of the running hardware threads closest (currently 70/4 = 17.5) to the overall average of all possible runners (117 / 8 = 14.6) ?

If we choose the process with 20 from the ready queue (for a total of 80 for the runners), the resource we are metering may become a structural bottleneck and stem the flow of instructions through the pipeline. We argue that the best process to mix with the resource−intensive processes is 8 because this will keep the usage of the resource as low as possible when the heavy users are running. Ideally the system would soon achieve equilibrium where two 20s and two from { 8, 9, 10, 10 } alternate, minimizing the resource as a potential bottleneck while preventing starvation. The schedulers which we developed and describe in Section 3.2 achieve this alternating equilibrium.

Section 2 discusses previous work on SMT scheduling and how the schedulers we present are more flexible. Section 3 examines the schedulers themselves and how we came to choose them from other resource possibilities. Section 4 describes how we extended an existing SMT simulator to include these scheduling policies. Section 5 shows the schedulers' performance on combinations of SPEC 2000 benchmarks. Section 6 looks at future work to build on the performance results. The last section draws some conclusions about the SMT schedulers we have developed.

2 Related Work

Scheduling jobs on a SMT processor has only recently received attention while the SMT model has been studied for well over five years.

In a single threaded architecture the objective is to have a single job running on the processor at all times. When a running job has to wait or is made to wait, a ready job is allocated the processor. Work in this area has focused on overlapping waits with computation, thereby attempting to utilize the processor to its fullest extent. Scheduling jobs on a single threaded processor has been well researched and several successful schemes have been proposed. Most schemes attempt to increase throughput by overlapping I/O of one job with the calculations of others. For instance, the scheduling policy chosen for several flavors of Unix (4.3 BSD Unix [Thompson74], Unix System V, and Solaris TS (timesharing scheduling class)) is ***Multi−level Feedback*** that encourages I/O bound jobs to run more frequently, thus leading to higher overall machine utilization.

In the case of a SMT processor such a policy would not be sufficient, since several jobs can be coscheduled on the processor. The jobs executing together on the processor compete for hardware resources making the choice of jobs to coschedule an important one, affecting processor utilization. Thus, the full benefits of SMT hardware can only be realized if the scheduler is aware of the interactions between the coscheduled jobs [Tullsen00]. Scheduling jobs on a SMT thus, requires finer−grained scheduling by efficiently allocating processor hardware resources among the coscheduled jobs.

We are aware of only one other work that has studied scheduling for an SMT. Tullsen, et al., [Tullsen00] propose a scheme named ***SOS*** that schedules jobs in a two−step process. The first step, called the ***sampling phase,*** collects information on various possible schedules while the second step, called the ***symbiosis phase***, uses sampled information to predict the schedule that would provide best performance and runs it for a certain number of cycles until the next sampling phase is triggered. The sampling phase runs for a small number of cycles when compared to the symbiosis phase. Success of this scheme thus, largely depends on the quality of schedule chosen in the sampling phase. To reduce the overhead of sampling, only a small number of schedules can be considered from the huge space of possible schedules. Information is gathered for each chosen schedule by running the jobs accordingly.

2.1 Motivation

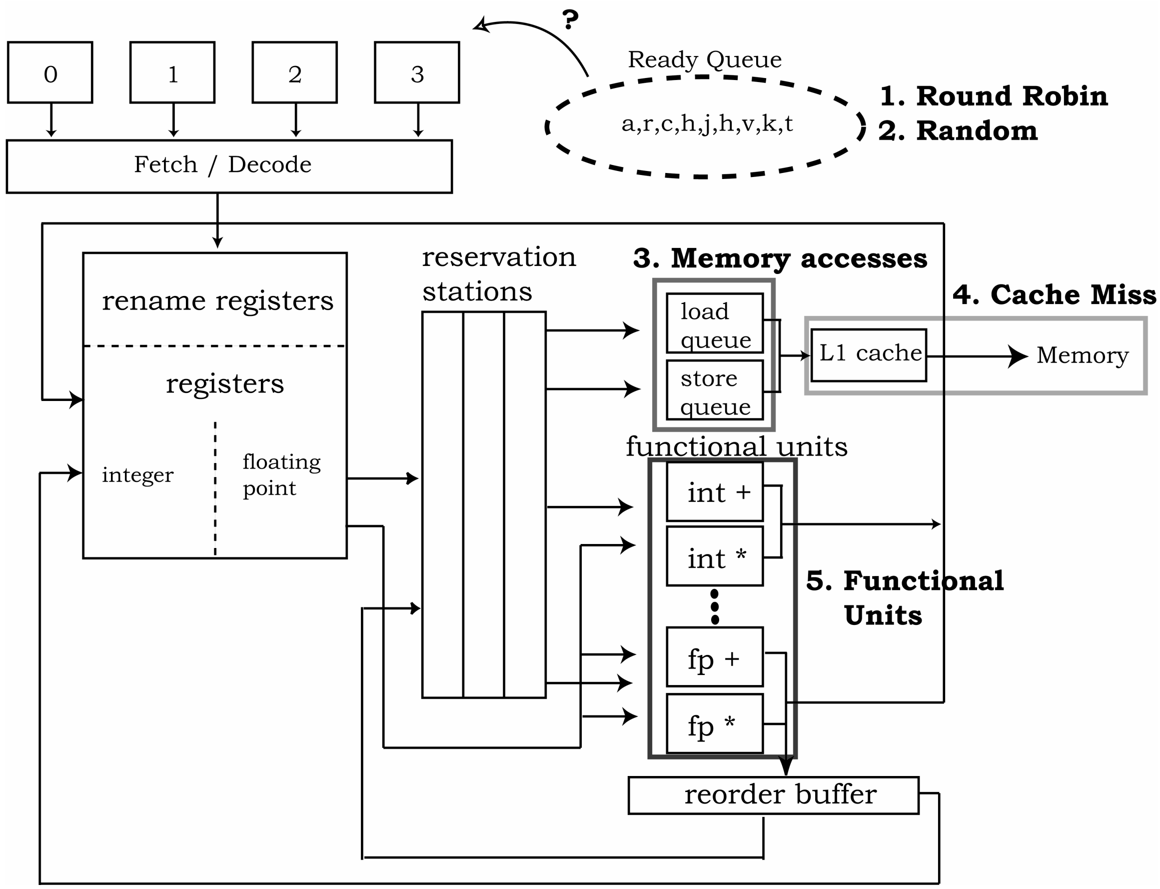
The SOS work demonstrates that the performance of a SMT processor is sensitive to the set of jobs that are coscheduled by the operating system job scheduler. This forms the basis of our project. There are however, two major limitations, we identify in SOS: 1) The strict two step decision making process can potentially make scheduling less responsive to job workload changes and, 2) choosing a good schedule from the large space of possibilities can be unwieldy in realistic scenarios. We propose a different scheme that chooses jobs to run on an SMT processor based on the information collected on each of the ready jobs as well as the information collected on the executing jobs. Although, like the SOS scheme, ours involves collecting information on the jobs, we believe it is more adaptive to workload changes than SOS as it bases the scheduling decisions on the most current state of the jobs. This differs from the strict two–step scheme of SOS.

3 Adaptive Scheduling

There exists a huge space potential scheduling policies from which to choose. As a result, a good portion of our project involved trying to come up with possible solutions to the coscheduling problem. The initial phase involved writing several micro–benchmarks designed to test very specific process interactions. For example, one micro–benchmark attempted to exploit spatial locality by accessing a large array in column–major order. To complement this test, another program was written to access the same array in row–major order. This gave us two programs with very different usage characteristics of the same resource.

Our other micro–benchmarks included programs to to test temporal locality, branch prediction capabilities, and the number of loads and stores. Floating–point and integer versions of all of these programs were constructed. At this point, we randomly tested combinations of these benchmarks with each other. It was the results of these micro–tests that lead us to select our scheduling policies. The policies we decided to implement were a memory usage scheduler, a cache miss scheduler, and a functional unit usage scheduler. We also implemented two simple, uninformed schedulers, round–robin and random, to use as a baseline for our results.

Before discussing the policies, it is worthwhile to mention what additions would be necessary to the hardware to support these scheduling policies. First of all, the processor must be capable of doing context switching. Should all four processes be stopped? should two? or only one? In what order should they be preempted? Our scheduler assumes that only one thread will be stopped at a time and the selection of threads to stop is done in a purely round–robin fashion. Secondly, resource counters are needed for each thread. In this way, whenever a process running on a thread uses a resource, like a floating point adder or a memory read port, the resource counter for that thread would be incremented by one. This data could then be retrieved by the operating system whenever it desired the information.



The first scheduler, MEM, attempts to schedule jobs doing a lot of loads and stores with jobs doing few loads and stores. Along the same vain is the MISS scheduler that tries to schedule jobs with a high cache miss rate with jobs achieving a low cache miss rate. The implementation for both of these schedulers is almost identical. They both calculate an average for all processes in the entire system. This average is based on the resource being examined – MEM averages memory accesses and MISS averages cache misses. Each waiting job in its turn is then averaged in with the job(s) that are to remain running on the processor. The process most recently removed from the processor is not considered during this process. The waiting job that, if it were run, would give the running processes an average closest to the system average is the one chosen to run.

The functional unit scheduler, FU, is based on the concept that the best scheduling policy will provide the best possible balance across all of the functional units. The FU scheduler works by going through each of the waiting jobs, assuming it would run with the job(s) remaining on the processor and calculating the standard deviation across all of the functional units. Again, the job most recently removed from the processor is not figured into these calculations. The waiting process that provides the lowest standard deviation is the one selected to run.

4 Methodology

We evaluate our adaptive scheme by carrying out a series of experiments using benchmarks from the SPEC 2000 suite on an extended version of Zilles' SMT simulator [Zilles99].

4.1 Simulation infrastructure

To evaluate our adaptive scheme, we extended Zilles's SMT simulator [Ziles99]. Prior to our extension, the simulator did not have support to schedule more jobs than hardware threads. Zilles' simulator models a SMT processor similar to the one proposed in [Tullsen96] along with several enhancements. We configured the core of the processor to have 1, 2, or 4 threads. Instructions are scheduled out–of–order and all threads share a single fetch unit, branch predictor, decoder, centralized instruction window (128 entries), scheduler, memory system and a pool of functional units. The memory system is comprised of separate 64KB L1 data and instruction caches, and a 1 MB L2 unified cache. The functional units are 3 integer units, 6 floating–point units, 1 load unit and 1 store unit. It is capable of fetching 4 instructions per cycle and has 5 pipe stages between fetch and decode, 7 pipe stages between decode and execute (for a total 12 cycle mispredict penalty). Other specifications of the simulated machine are described in [Zilles99].

In extending Zilles' simulator to support scheduling we added a:

- ***Context switch mechanism***: A desired job running on a given hardware thread can now be interrupted and replaced.

- ***Scheduler mechanism***: A higher number of jobs than hardware threads can now be run on the simulated processor. The jobs get to use the processor based on policies that can be specified.

- ***Statistics gathering mechanism***: To simulate hardware counters as in a real processor we added support to keep track of the resource consumption for every job while it used the processor. In our scheduling schemes we rely on these gathered statistics to guide take well–informed decisions.

4.2 Benchmarks

Ten benchmarks were selected from the SPEC CPU2000 suite [SPEC CPU 2000]. We report data from six of them in Table 1. Three of these benchmarks are integer intensive and the remaining are floating point intensive. Our primary criterion for choosing these benchmarks was their run–time behavior. ***Crafty***, for instance, heavily stressed the integer functional units by having a significant number of logical operations such as and, or, exclusive or and shift, while going easy on the store unit. Table 2 gives the functional unit usage for two sample INT and FP benchmarks.

For each of these benchmarks we use the reference input set and generate EIO [Burger97] traces of 100 million instructions (on a Linux system). To take into account the initialization phase of each benchmark we skip the first 1 million instructions.

In addition to these established benchmarks we created microbenchmarks to decide upon a set of heuristics to use in making job scheduling choices.

Name	IPC	Loads (million)	Stores (million)	L miss (million)	S miss (million)	Branches (million)	Branch hits (million)	Hit ratio

Crafty	2.770	31.466	7.244	0.208	0.157	9.989	9.546	0.957
Gcc	2.290	28.053	13.039	0.224	0.195	13.487	12.796	0.949
Perl	2.814	25.803	13.460	0.134	0.847	13.831	13.488	0.975
Mgrid	3.750	43.820	8.650	0.136	3.800	1.859	1.809	0.973
Swim	3.286	22.423	4.249	0.028	1.504	1.624	1.609	0.991
Facerec	3.551	22.430	10.406	0.136	0.406	9.557	9.462	0.990

Table 1: Characteristics of some benchmarks. The data was collected for approx. 100 million instructions. The ones shaded are INT and the others are FP.

FU type	Crafty	Mgrid
Null	1141	390
int−ALU	73,344,019	21,892,622
int−multiply	339,519	163,574
int−divide	0	0
FP−add/sub	115	56,655,872
FP−comparison	0	0
FP−conversion	36,199	0
FP−multiply	25	8,388,608
FP−divide	2	0
FP−sqrt	0	0
rd−port	32,063,507	43,869,561
wr−port	7,298,864	8,650,004

Table 2: Functional unit usage of two benchmarks. The values represent the number of instructions that used the given unit. The data was collected for approx. 100 million instructions.

5 Results

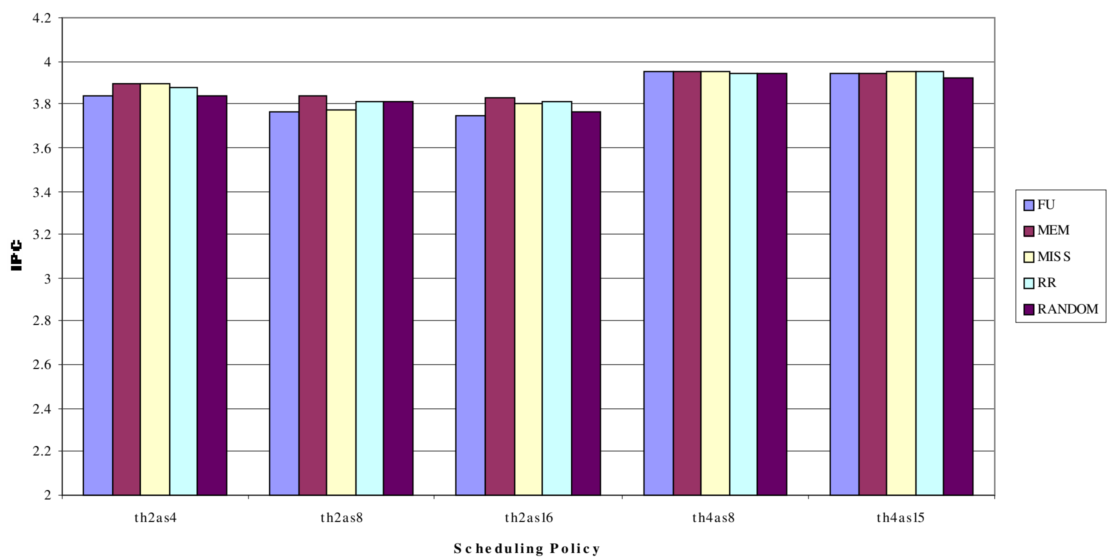
To test the various scheduling algorithms, we chose 6 benchmarks. These consisted of 3 integer intensive and 3 floating−point intensive benchmarks. Some relevant characteristics of these benchmarks together can be found in Table 1. There were three basic sets of benchmark groupings: integer intensive only, floating−point intensive only, and a combination of floating–point and integer intensive benchmarks. We ran each of these groups with every scheduling algorithm. All tests were repeated on 5 different hardware configurations (2 threads–4 processes, 2 threads–8 processes, 2 threads–16 processes, 4 threads–8 processes, and 4 threads–16 processes), for a total of 75 tests. A representative set of all of this data is presented in the following sections.

5.1 Floating–Point Only Benchmarks

Graphs 1 represents the average IPC for all of the FP intensive benchmarks running under a given set of test conditions. Note that whichever scheduler is run, if the only jobs in the system are floating point intensive, high IPCs result. In fact, there is very little difference between any of the policies when scheduling a floating–point benchmark. We believe this is because of three factors. First, floating–point programs still involve a fair amount of integer work. Integer intensive programs, on the other hand, involve very little, if any, floating point work. Hence, floating–point programs have an inherent amount of balance built into them. This results in good parallelism between programs regardless of how they are scheduled.

Second, Table 1 shows that the floating–point programs we examined tend to do relatively few branches and have a high branch prediction accuracy when compared to the integer benchmarks used. Fewer branches and higher accuracy lead to a higher IPC because there are fewer wasted cycles due to misprediction. When dealing with an SMT processor, it also means that fewer system resources are being unnecessarily consumed.

Average IPC (200 million instructions, floating point benchmarks only)

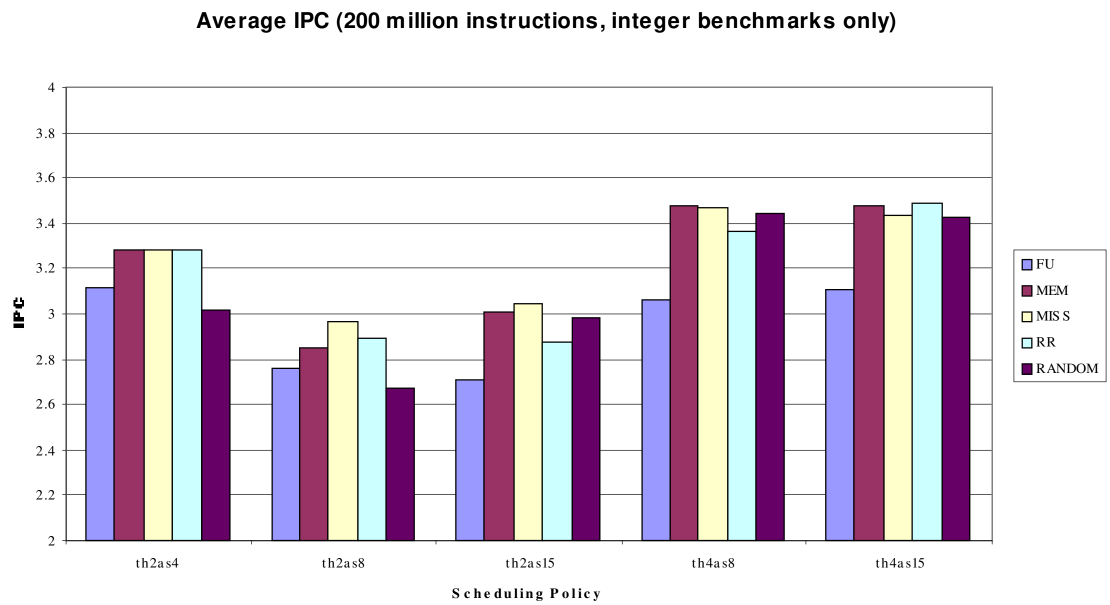


Graph 1

Last, there are twice as many floating-point units as integer units in our simulator (6 floating-point ALU's and 3 integer ALU's). So if there is a mix as to the type of floating point operations being conducted, these instructions are more likely to find an idle functional unit. Because the scheduling policy does not appear to impact floating-point intensive programs, the rest of this section will focus on the integer only and mixed tests.

5.2 Integer Only and Mixed Benchmarks

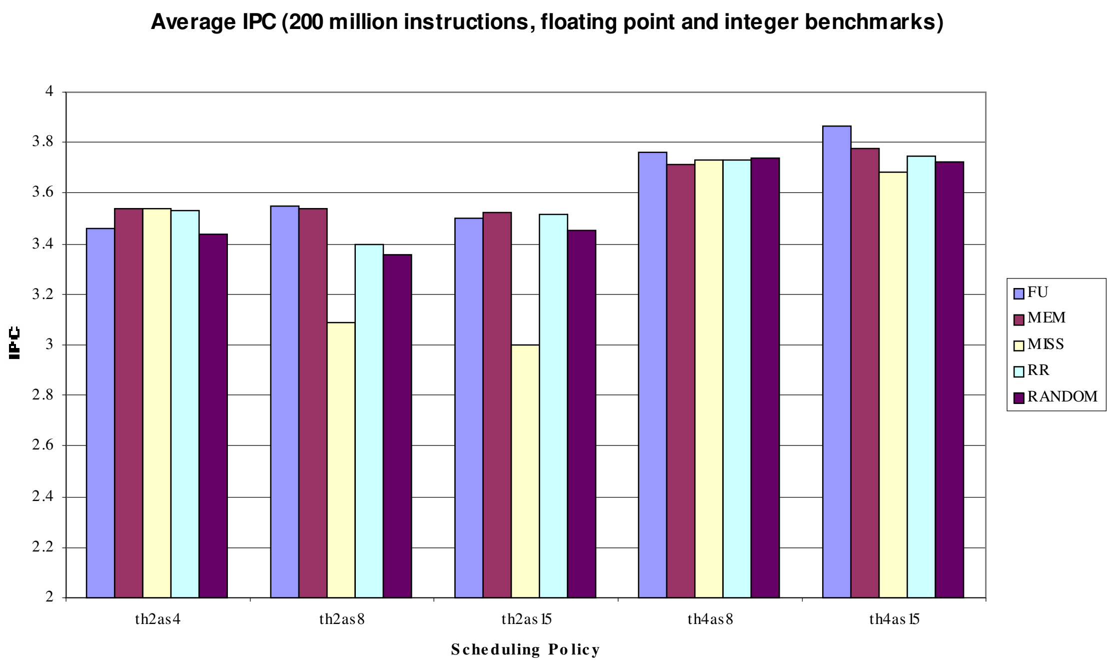
We will begin our discussion of the mixed and integer only tests by considering Graphs 2 and 3. Note that almost all of the schedulers noticeably outperform the rest of the schedulers for some given configuration, and noticeably underperform for a different configuration. The one exception to this seems to be the MEM scheduler. In almost every benchmark examined, it either performed the best, or within a small fraction of the best policy. Intuitively, this result makes sense. If a process, which is in the midst of doing frequent memory references (call it P1), is scheduled with a job doing more infrequent references (call it P2), the latencies of P1's memory references can be "covered" by the instructions in P2. This is true regardless of the type of jobs being done.



Graph 3

In contrast to the MEM scheduler is the FU scheduler. It performs very well for all of the mixed scheduling jobs, but performs very poorly for all of the integer only jobs. When dealing with a mixed bag of floating-point and integer jobs, it makes sense that the FU scheduler should perform well. There is a good deal of contention for all of the resources in the system with some processes requiring one particular type of resource (i.e. floating-point functional units) and other processes demanding a different resource (i.e. integer functional units). This situation provides many opportunities for instruction level parallelism across functional units, which the FU scheduler is built to exploit.

In contrast, the situation involving only integer benchmarks provides much less opportunity for parallelism between functional units since all of the processes are competing for a much smaller number of resources. The reason MEM performs well in this situation is because it is considering a different resource, namely memory. While the integer ALUs will continue to be heavily loaded, different benchmarks will have different memory requirements at different times in their execution. As a result, a process that is currently in the midst of heavy memory usage (i.e. a loop multiplying two arrays together and placing the result in a third array) can be scheduled with a job doing a math intensive operation (i.e. repeatedly solving a set of complex equations).



The reason the FU scheduler does not work well in the integer–only situation is because it only attempts to keep a good balance between all of the functional units and does not consider the total number of operations a process will add to a functional unit. This means a job executing a thousand instructions on each of the functional units will look very similar to a job executing ten thousand instructions on each functional unit – at least, they will look the same to the scheduler used in our research. Once the functional units have become saturated, it does not matter how balanced the requirements to each unit are and it is more important to try to use another system resource – like memory. An FU scheduler based more on the total number of instructions a process is trying to commit to the functional units would probably be more effective than one based on balance.

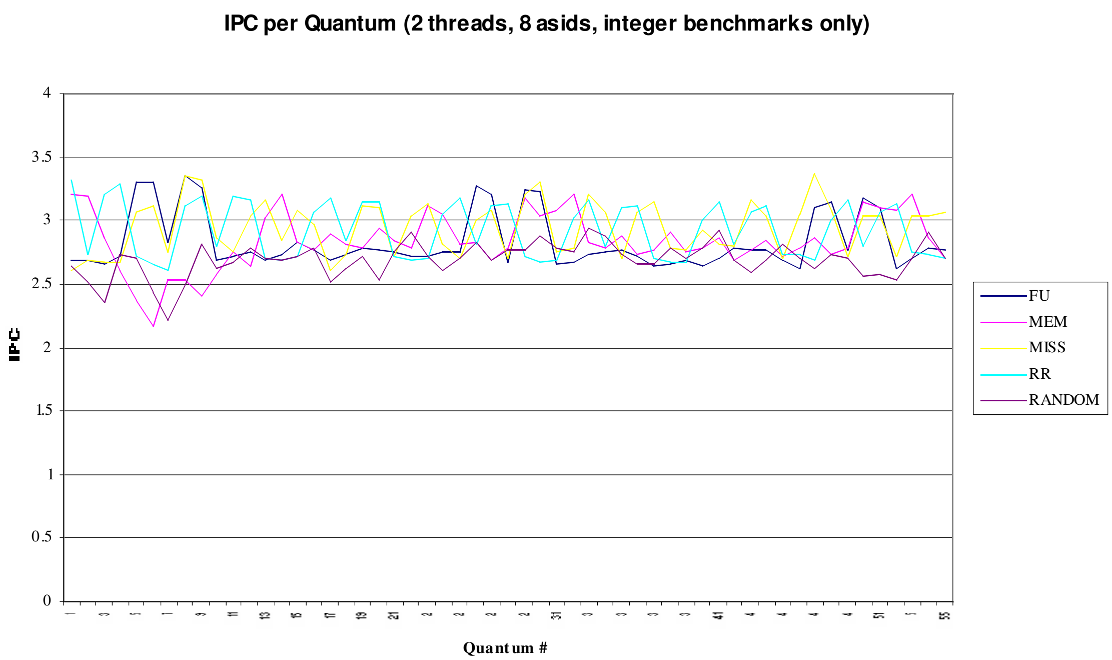
The MISS scheduler's results correlate well with those found in [Tullsen00]. Here Tullsen and Snavely discovered: "The Dcache predictor was an inconsistent performer… in some cases it chose the ***worst*** schedule." When only a few threads are available, there are many processes (th2as8 and th2as16), and the processes running are a mix of floating-point and integer, the miss scheduler performs very poorly. This indicates that picking a job to schedule because it has a lot of misses in the L1 cache is a bad idea if there are not a large number of other threads to cover its latency (remember: the idea is to schedule jobs of complementary types, so if a job with good cache hit ratios is selected, one with a poor cache hit ratio will be scheduled when using MISS). For instance, if there is only one thread other than the one with the large number of cache misses and it stalls or mispredicts a branch, the overall impact on IPC is going to be huge. The reason this problem is not encountered in the 2 thread-4 process configuration is because there are so few threads to choose from that the scheduling policy seems to have little overall effect. The reason it is not a problem with 4 threads is because there are 3 times as many threads to cover the cache miss latency if a miss occurs.

Why does the cache policy seem to outperform all of the other schedulers when running few threads and many processes in an integer only configuration? Because it performs almost exactly the same for the integer only tests as it does for the mixed tests when running in this configuration. This is interesting because it is the only scheduling policy that performs the same for a mix of processes as well as for integer only for ***any*** of the processor configurations. We conclude from this that processes tend to have similar capabilities for covering latency across differing workloads when there are few threads.

The last two schedulers are round–robin and random schedulers. These were included because they are simple schedulers and would provide a baseline for the more informed schedulers to try and surpass. The graphs seem to show that in most instances, round–robin and random perform as well as our more sophisticated schedulers. First of all, when there are many more processes than threads and those processes are of different types (floating–point and integer), the FU scheduler consistently achieves approximately a 5% increase over RR and RANDOM. This can be seen from the th2as8 and th4as15 data sets. This makes sense since these are the cases that require the most decision making for scheduling and will benefit the most from an intelligent policy. The reason these same benefits are not seen in an all integer benchmark test (and indeed, a drastic decrease in the FU scheduler is seen) is because there are many fewer resources to schedule for and a simpler, less informed policy is going to do relatively well.

The RR scheduler tended to be fairly dependant on the way the processes were presented to it. The performance could vary by approximately 5% for the exact same set of processes. All of the other schedulers performed the same regardless of the order they were presented.

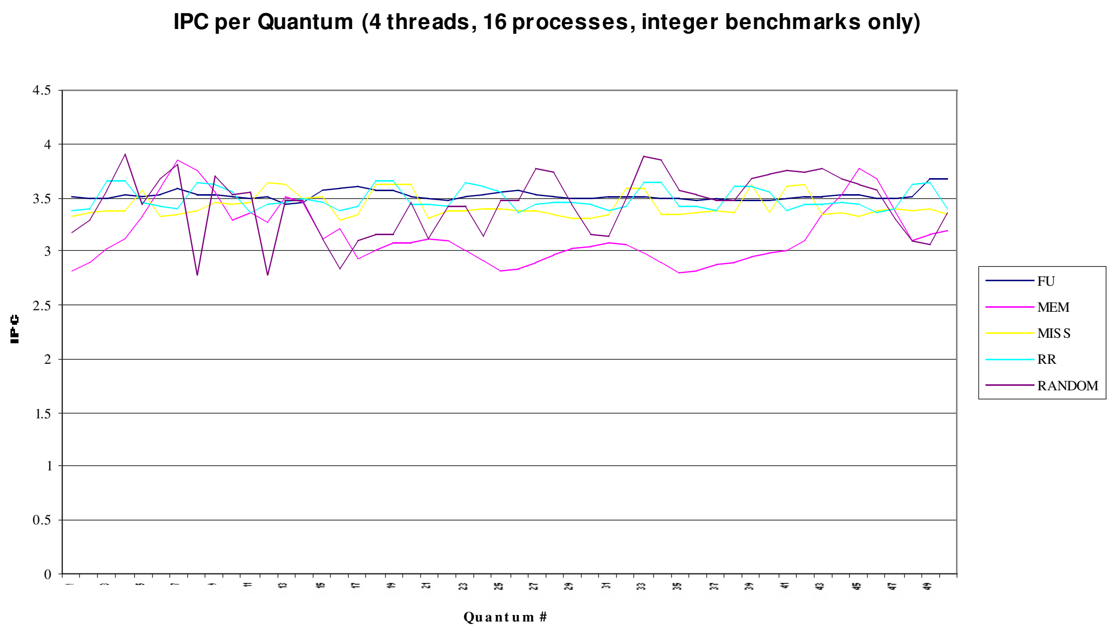
One final interesting observation is the drastic drop–off for the random scheduler for the integer only benchmarks with the th2as4 and th2as8 configurations. Unlike the mixed (or floating–point only) when running only integer jobs, the possibility for a very high scheduling value is much lower. As an example, it can be seen from Graph 4 that random scheduling can produce some spikes in performance that produce much better and much worse instantaneous IPCs than all of the other policies. Many threads and/or many processes means there usually exists a very good scheduling policy that none of the other scheduling heuristics uncover. Of course these spikes are offset by the very low performance schedules RANDOM finds. However, when there are very few processes and very few threads and the process mix is one not conducive to good parallelism (e.g. all integer processes), there does not exist the possibilities for this high parallelism. Yet, there does still exist the possibility for very poor parallelism (see Graph 5). This means that the average IPC for a random scheduler with few scheduling options and a difficult to parallelize workload is going to suffer greatly.



Graph 4

Graphs 4 and 5 demonstrate several interesting patterns. RANDOM produces very erratic results, so the usually competitive IPC's it produces are balanced out by its unpredictable behavior on a per quantum basis. RR produces a repeatable pattern of

IPC values. This is completely intuitive when one considers that RR is simply cycling through all of the processes in exactly the same order. While there are a few spikes in the FU scheduling policy when using the 2 threads−8 processes configuration, we feel that most of these are due to the starvation prevention we built into the scheduler. Notice that shortly after a spike, FU tends to level out for quite a few cycles. In the 4 thread−16 process configuration, there are no spikes at all for the FU scheduler.



Graph 5

6 Future Work

In this study we present a number of heuristics that can be used to pick jobs to coschedule. We would like to see a scheme that would use a combination of these heuristics to make scheduling decisions. In the same vein, given a number of different schedulers, a scheme that chooses an appropriate scheduler depending upon the current workload may also be worthwhile.

We do not consider the time the applications spend in the operating system code or the scheduling overhead caused by the operating system. Focusing entirely on user–mode execution is not sufficient. To understand the scheduling problem for SMT a thorough execution and measurement of the operating system and the application is required.

Through our experiments we tried to capture a few example workloads that may be coscheduled on a SMT processor. While this is a good starting point, we would like to consider applications from other domains that are gaining a lot of attention like multimedia, database and mobile computing.

Since, the power consumption of microprocessors is becoming increasingly important in design decisions, power–aware scheduling could be reasonable direction to investigate.

Lastly, we would like to study how well our schemes perform in a scenario in which jobs arrive and depart. This will be a good test for the adaptation.

7 Conclusions

Our results show that a scheduler based strictly on the number of memory accesses seems to perform well in all of the tests. A scheduler based on balancing the load across functional units outperforms all schedulers when there are a large number of scheduling decisions – many threads, many processes, and a diverse workload. A scheduler based on the number of cache misses has the potential to perform very poorly when the there are few threads to schedule and a large number of diverse jobs to choose from. A random scheduler will find excellent combinations, because it tries everything, but it will also find terrible combinations of jobs to run, so even though it tends to have a good average, it also has a large variation from quantum to quantum.

The round−robin scheduler also seemed to perform well in most situations, but its performance is dependant on the ordering of jobs.

The research presented here gives a deeper understanding of the concerns involved in trying to schedule an SMT processor. We also propose several different scheduling policies and present the necessary hardware modifications to accommodate these, and possibly other, schedulers. More than anything, this work shows that the issues involved in finding the best scheduler are many and complex. In fact, it is most likely that there does not exist a scheduling policy that performs better than every other for all possible workloads.

Acknowledgements

We thank Craig Zilles for giving us his SMT simulator and helping us get started and Mark Hill for giving us access to an Alpha machine (paul.cs.wisc.edu).

Bibliography

[Burger97] D. C. Burger, T. M. Austin. The Simple Scalar Tool, Version 2.0. ***Technical Report CS−TR−97−1342, University of Wisconsin−Madison***, June 1997.

[SPEC CPU00] CPU−Intensive Benchmark Suite. http://www.spec.org/osg/cpu2000/

[Thompson74] K. Thompson and D. Ritchie. The Unix Time−Sharing System. In ***Communications of the ACM***, July 1974.

[Tullsen96] D. M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In ***ISCA 96***, pages 191−202, May 1996.

[Tullsen00] D. M. Tullsen and A. Snavely. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In ***9th International Conference on Architectural Support for Programming Languages and Operating Systems***, November 2000.

[Zilles99] C. B. Zilles, J. S. Emer, and G. S. Sohi. The Use of Multithreading for Exception Handling. Proceedings for ***Micro−32,*** Nov 1999.