Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor

Allan Snavely
University of California, San Diego
10100 Hopkins Drive
La Jolla, California
allans@sdsc.edu

Dean M. Tullsen
University of California, San Diego
9500 Gilman Drive
La Jolla, California 92093
tullsen@cs.ucsd.edu

Geoff Voelker
University of California, San Diego
9500 Gilman Drive
La Jolla, California 92093
voelker@cs.ucsd.edu

ABSTRACT

Simultaneous Multithreading machines benefit from jobscheduling software that monitors how well coscheduled jobs share CPU resources, and coschedules jobs that interact well to make more efficient use of those resources. As a result, informed coscheduling can yield significant performance gains over naive schedulers. However, prior work on coscheduling focused on equal-priority job mixes, which is an unrealistic assumption for modern operating systems.

This paper demonstrates that a scheduler for an SMT machine can both satisfy process priorities and symbiotically schedule low and high priority threads to increase system throughput. Naive priority schedulers dedicate the machine to high priority jobs to meet priority goals, and as a result decrease opportunities for increased performance from multithreading and coscheduling. More informed schedulers, however, can dynamically monitor the progress and resource utilization of jobs on the machine, and dynamically adjust the degree of multithreading to improve performance while still meeting priority goals.

Using detailed simulation of an SMT architecture, we introduce and evaluate a series of five software and hardware-assisted priority schedulers. Overall, our results indicate that coscheduling priority jobs can significantly increase system throughput by as much as 40%, and that (1) the benefit depends upon the relative priority of the coscheduled jobs, and (2) more sophisticated schedulers are more effective when the differences in priorities are greatest. We show that our priority schedulers can decrease average turnaround times for a random jobmix by as much as 33%.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance Analysis and Design Aids; D.2 [**Software**]: Software Engineering; D.5.m [**Computer Systems Organization**]: Miscellaneous—*operating systems*

General Terms

Design, Performance, Measurement

Keywords

Simultaneous Multithreading, Job Scheduling, Priorities

1. INTRODUCTION

Simultaneous Multithreading (SMT) [33, 32, 17] architectures execute instructions from multiple streams of execution (threads) each cycle to increase instruction level parallelism. When there are more jobs in the system than there is hardware support for simultaneous execution (that is, more than the number of hardware contexts), the jobscheduler implements multiprogramming at two levels. It makes a *running set* of jobs that will be coscheduled and compete in hardware for resources every cycle; it decides which jobs should be coscheduled in the running set at a much coarser granularity. Thus, from among the jobs available to be run, the jobscheduler decides which ones should be run together.

The term *symbiosis* has been used to refer to the effectiveness with which multiple jobs achieve speedup when run on multithreaded machines [24] [25]. Symbiosis can be positive or negative. Jobs in an SMT processor can conflict with each other on various shared system resources. Throughput may go up or down depending on how well the jobs in the running set *symbios* or 'get along'. It therefore makes a difference which jobs are coscheduled. A job scheduler which takes symbiosis into account can yield enhanced throughput and response time.

Previously [25] we presented a symbiotic OS-level jobscheduler for SMT that dynamically adjusts its scheduling decisions to enhance throughput and lower response time over what would be expected if scheduling were left to chance. The scheduler is called SOS (for Sample, Optimize, Symbios) because it first samples the space of possible schedules while making progress through the job mix. It then examines hardware performance counters and applies a heuristic to guess at an optimal schedule, then runs this (presumed to be optimal) schedule to boost system utilization. SOS scheduling was shown to improve system response time by as much as 17% over a naive scheduler.

However, that previous work neglects the important issue of job priority. Left to itself, SOS will fill issue slots as efficiently as it can from available instruction streams without regard to the *importance* of the instructions or the jobs they come from. Users have their own notion of individual job importance which may run counter

to decisions the system would make to boost overall throughput or to diminish average job response time. The preferences of the user are embodied in *priorities*. Modern operating systems such as Unix and NT allow users to specify the relative importance of jobs; consequently, a system that allows coscheduling should provide mechanisms to allow resource allocation to be based upon user specified priorities.

This paper presents several mechanisms for supporting the notion of priority on an SMT processor while still allowing symbiotic jobscheduling. These include a straightforward adaptation of a UNIX priority scheme from a single-threaded machine, more complex software mechanisms suitable for a multithreading machine, a mechanism requiring hardware support for the notion of priority, as well as a hybrid (hardware/software) scheme. Using detailed simulation of an SMT architecture, we show that the promise of SMT can still be realized when jobs of widely different priority are coscheduled together. Low priority jobs can run beneficially with high priority jobs, consuming unused resources without slowing down the high priority jobs. System utilization (and thus overall response time and throughput) can remain high, even when priority jobs are given the majority of system resources. Overall, our results indicate that coscheduling priority jobs can increase system throughput by as much as 40%, and that (1) the benefit depends upon the relative priority of the coscheduled jobs, and (2) more complex schedulers are more effective with greater differences in priorities. We also show that our priority schedulers can decrease average turnaround times for a random jobmix by as much as 33%.

The rest of this paper is organized as follows. Section 2 provides background information on simultaneous multithreading architectures and priority scheduling. Section 3 motivates the opportunities for a scheduler for a multithreading machine to both coschedule jobs and respect job priority, and increase system throughput as a result. Section 4 describes our experimental methodology and evaluation metrics. Section 5 evaluates a series of both software and hardware-assisted priority schedulers for SMT machines, with a focus on system throughput. Section 6 evaluates the impact of these schedulers on response time. Finally, Section 7 describes previous work, and Section 8 concludes.

2. BACKGROUND

This section provides background information on multithreading architectures and the UNIX priority scheduler.

A simultaneous multithreading processor [33, 32] allows multiple threads of execution to issue instructions to the functional units each cycle. This can provide significantly higher instruction throughput than conventional superscalar processors. The ability to combine instructions from multiple threads in the same cycle allows simultaneous multithreading to both hide latencies and more fully utilize the issue width of a wide superscalar processor.

The simultaneous multithreading architecture we assume is the SMT processor proposed by Tullsen et al. [32]. This is an out-of-order issue processor which has the ability to fetch up to a total of eight instructions from the instruction cache each cycle from up to two different threads. Threads are given priority for fetch based on the *Icount* mechanism, which favors those threads which have the fewest instructions in the pre-execute portion of the pipeline. This creates the most even mix of instructions from the various threads and maximizes instruction-level parallelism in the instruction issue queues.

The multiple threads which will run on an SMT processor might come from parallel applications, multiple single-threaded applications, or a combination of both. For this research we assume a multiprogrammed workload of single-threaded applications, because that forces the OS to make more fine-grained scheduling decisions.

Work on operating system issues relating to SMT is timely; Intel has announced their future server and desktop processors will feature simultaneous multithreading (under the marketing name hyperthreading technology) [1], joining the announced SMT-based Alpha 21464.

For further motivation as to the importance of investigations into priorities on multithreaded systems, we found in previous work [19] that users of a production multithreaded system may be unhappy if the jobs of other users adversely affect their job's performance. Also we found that coscheduled jobs on multithreaded systems do affect each other's performance. And they do so more than they do on space-shared multi-user systems (due likely to intimate sharing of resources on multithreaded systems). Also we found users frequently submit jobs in two modes (1) low-priority batch jobs which can be preempted, and (2) high-priority interactive jobs that the user would not wish to have preempted. Thus in brief, users expect priorities, and users frequently use dramatically different priorities for these two broad classes of jobs.

In designing an operating system for a new class of architecture, it is important to understand what mechanisms users expect; the traditional UNIX scheduler, as exemplified by BSD Unix [15], implements a priority scheduler using a mulitilevel feedback queue. Jobs are assigned one of 20 static priority levels. Each timeslice, the job with the highest *dynamic* priority is chosen to run, and that job's dynamic priority is then aged (reduced). The aging mechanism allows low static-priority jobs to eventually run, but still allows high static-priority jobs to get the larger proportional share of processor time. We use a simplified model of the effect of static priorities in this paper for dual purposes. First, the aging mechanism provides fairness and priority over very long runtimes but not necessarily over the small time windows that can be simulated by an instruction-level simulator, so we need a simpler mechanism for our experiments. Second, some of our scheduling algorithms depend on having a model of the effect of priorities so that they can provide the same throughput guarantees on a multithreaded system that a single-threaded system would provide. Note that these algorithms could trivially be adapted to a more complex model of UNIX priorities.

We found experimentally that the long term effect of differing priorities can be accurately modeled as 'proportional sharing' where jobs get a share of the system proportional to their priority. Various numbers of jobs were submitted at different priorities and their share of the system recorded using the UNIX utility 'top' after 60 seconds on a Alpha 21264 processor running Digital Unix V. Figure 1 shows the result for a simple cases of two jobs with different priorities adding up to 19. In the rest of the paper we implement policies that result in 'proportional sharing'.

3. MOTIVATION

The opportunity presented by SMT hardware is to increase utilization of the processor's execution resources by fetching instructions from independent instruction streams. The challenge we address is that jobs of different priority should not be allowed to compete equally for system resources (that would violate the intuition be-

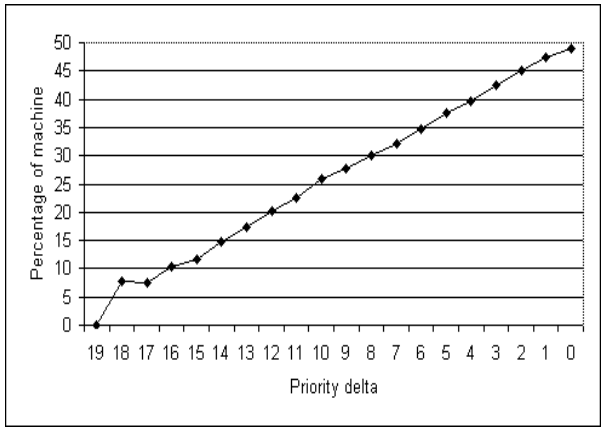

Figure 1: Long-term effect of multilevel queues with feedback mechanism on 2 jobs of differing priority. The X axis is the difference between the priorities of the 2 jobs. The net result over time is 'proportional sharing'.

hind 'priority'), yet straight-forward implementations of priority decrease the opportunity for parallelism and high throughput.

On a conventional single-threaded processor, a job's priority level, when considered with respect to all other jobs' priority levels, provides it two guarantees: (A) access to a certain percentage of the CPU over time, and (B) throughput that is only degraded (relative to exclusive access to the processor) by a certain percentage. For a single-threaded processor, those two guarantees are nearly identical. However, on an SMT machine, they are not. This paper demonstrates that an SMT scheduler that provides the second guarantee, while being allowed to violate the first, provides higher system-level performance than one that preserves the first.

We want to provide the effect of proportional sharing on an SMT machine while not being necessarily bound to the mechanisms of single-threaded machines. The default OS configuration of Digital UNIX V allows the following mechanism for users to manage user-level priorities. Users assign job priority numbers in the range 0-19. Priority 19 is a special-case lowest priority job which sleeps in the presence of any other job having a higher priority (lower priority number). A runnable job *i* with priority p_i is scheduled on the CPU for a fraction of total available cycles equal to

$$FRAC_i = (19 - p_i) \,/\, \sum_{j=1}^{jobs} (19 - p_j)$$

In other words, jobs get scheduled for a fraction of available cycles based on a weighted sum of priority numbers. Note that

$$\sum_{i=1}^{jobs} FRAC_i = 1$$

which is to say that all the available cycles are distributed among the runnable jobs.

On an SMT machine, more than one runnable job is scheduled for execution in the same timeslice. This adds a degree of complexity to the division of system resources among runnable jobs because a job does not get exclusive use of the machine during its timeslice. Jobs of equal priority can presumably be coscheduled in the same timeslice and compete for resources equally. However, what about jobs of different priority? We need a way of enforcing priorities while allowing as much coscheduling as possible (to boost utilization). If we assume that a job coscheduled with P other jobs gets $1/P$ of the system then a job whose priority number entitles it to $FRAC_i$ of the system is willing to be coscheduled for

$$COFRAC_i = MIN(1, (1 - FRAC_i) \,/\, (1 - 1/P)) \tag{1}$$

of the time and should be scheduled to run by itself for

$$SOLOFRAC_i = 1 - COFRAC_i \tag{2}$$

of the time to obtain the additional cycles to which its priority entitles it.

For example, a high priority job with priority entitling it to 75% of the system is willing to be scheduled with a lower priority job (entitled to 25% of the system) for $(1 - .75)/(1 - .5) = .5$ of the time under the assumption that the high priority job gets half the system when run with the low priority job. The high priority job gets 100% of the system .5 of the time and 50% of the system .5 of the time giving it an average of 75% of the system overall.

A *viable* schedule is one that preserves priorities by giving each job at least the fraction of system resources its priority entitles it to. We can use equations 1 and 2 to enforce viability; in section 5 we will show several mechanisms that boost utilization and throughput by solo-scheduling job i for $SOLOFRAC_i$ and coscheduling jobs (when sufficient runnable jobs are available) at the level of multithreading supported by the hardware for the rest of the time CS where

$$CS = 1 - \sum_{i=1}^{jobs} SOLOFRAC_i \tag{3}$$

Scheduling in this way limits the combinatorial space of schedules since we are not considering schedules that coschedule at a level lower than the hardware supports when there are more runnable jobs than that available. Multithreading machines are designed under the assumption that (at least up to some modest level) more multithreading is better. This maximizes the opportunity for latency hiding via issue from independent instruction streams. Previous work has shown that coscheduling to the maximum supported level on modest (up to 8 way) SMT is in general best, [33] and [23] showed that cases of *negative symbiosis* are rare and actually required (to exhibit it in that study) crippling the base SMT configuration to artificially create bottlenecks on system resources.

There are two related factors in a schedule that can boost utilization. We want to dynamically identify these and exploit them while maintaining schedule viability.

First, if we determine that a job gets more than $1/P$ of the resources it would obtain running alone when it runs with P other jobs then its $COFRAC$ can be increased. In other words we can violate property (A) above because property (B), and thus the apparent affect to the user, is preserved. In fact jobs *usually* get more than $1/P$ of system resources (such as issue slots) which they would have obtained if run alone when they are run with P other jobs on SMT hardware. This is because jobs running alone do not fully utilize all system resources and so leave 'gaps' that other jobs can exploit when run contemporaneously. This is the premise of hardware mul-

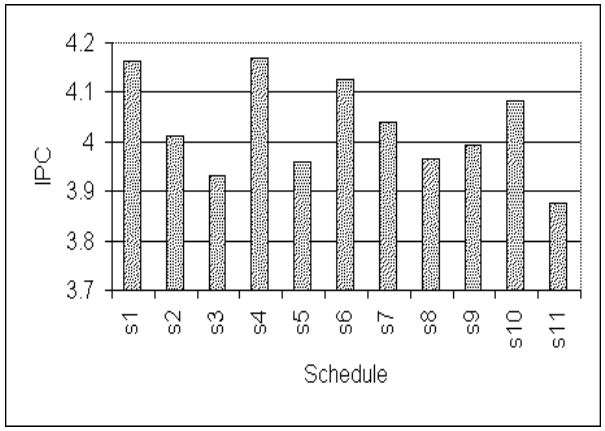

Figure 2: IPC of 8 jobs coscheduled 4 at a time, depending upon the sets chosen for coscheduling.

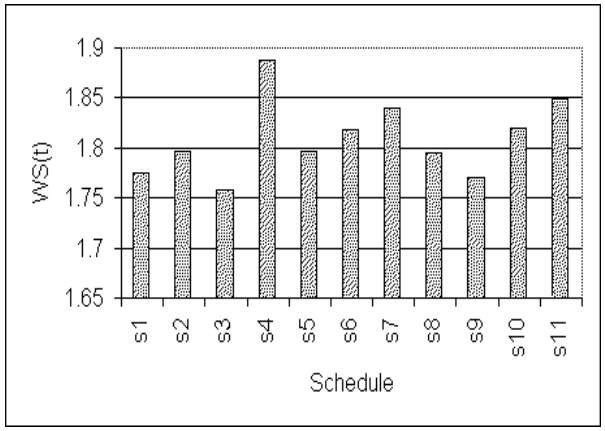

Figure 3: WS(t) of 8 jobs coscheduled 4 at a time, depending upon the sets chosen for coscheduling.

tithreading to begin with; there should be an increase in throughput over singe-threaded hardware due to this effect. The challenge is to determine what fraction of its solo-execution resources a job does obtain when coscheduled so that priorities can be enforced against that baseline. If we know *a priori* that a job will obtain fraction X_i of its solo dedicated IPC when run with P other jobs where $X_i > 1/P$ then that job can be coscheduled for

$$COFRAC'_i = MAX(1, (1 - FRAC_i) \ / \ (1 - 1/X_i)$$

where now $COFRAC'_i > COFRAC_i$ This in turn yields a greater fraction of the time that jobs can be coscheduled

$$CS' = 1 - \sum_{i=1}^{jobs} SOLOFRAC'_i$$

and should boost throughput by boosting the percentage of time when jobs are simultaneously executed.

Second, when the number of jobs in the system P is greater than M where M is the level of multithreading supported in hardware, there is a choice to be made as to which sets of jobs should execute together in the same timeslice. We previously showed how performance can vary depending upon which jobs run together in the same timeslice [24]; changing partners can boost throughput. In section 5 we show how to adapt the same scheme to allow us to compute $COFRAC'_i$ and *further* increase throughput.

4. METHODOLOGY

Table 1 summarizes the benchmarks used in our simulations. All benchmarks are taken from the SPEC2000 and SPEC95 suite and use the reference data set. The benchmarks were fast-forwarded to get out of the startup phase before being simulated for 250 million instructions times the number of threads being simulated.

Execution is simulated on an out-of-order superscalar processor model which runs unaltered Alpha executables. The simulator is derived from SMTSIM [32]. The simulator models all reasonable sources of latency, including caches, branch mispredictions, TLB misses, and various resource conflicts, including renaming registers, queue entries, etc. It models both cache latencies and the effect of contention for caches and memory buses. It also carefully models execution down the wrong path between branch misprediction and branch misprediction recovery.

The baseline processor configuration used for most simulations is shown in Table 2.

4.1 Weighted Speedup

This section explains a previously developed metric for throughput appropriate for SMT architectures [25] and also explains why we use it in this study.

When jobs are run on an SMT machine, processor utilization can go up dramatically. This is because thread level parallelism (TLP) is converted into instruction level parallelism (ILP). The net effect is to increase the pool of available-to-execute instructions and thus the opportunity for the functional units to be utilized on every cycle.

We wish to have a formal measure of the *goodness* or speedup of a coschedule. Intuitively, if one jobschedule executes more useful instructions than another in the same interval of time, the first jobschedule is more symbiotic and exhibits higher speedup. This suggests IPC as a measure of speedup. But an unfair schedule can appear to have good speedup, at least for awhile, by favoring high-IPC threads. To ensure that we are measuring real increases in the rate of progress through the entire jobmix, we define the quantity

$$WS(t) \quad \text{'Weighted Speedup in interval t'} =$$
$$\sum_{i=1}^{n} (\text{realized IPC } job_i / \text{ single-threaded IPC } job_i)$$

WS(t) equalizes the contribution of each thread to the sum of total work completed in the interval by dividing the instructions executing on each job's behalf by its natural offer rate if run alone. Implicit in the definition is a precise idea of the interval *t*. An interval is not just a measure of elapsed time. An interval starts on a certain cycle, but also at a particular point in the execution of each job. An interval ends on a certain cycle and at a specific point of execution of each job.

WS(t) of a single-threaded job running alone is 1. This is intuitive since there is no speedup due to multithreading when running only one thread. More importantly, WS(t) is a fair measure of real work done in processing the jobmix. In order for it to have a value greater

SPEC CPU2000 Benchmarks
mcf, crafty, parser, eon, perlbmk, gap, vortex, bzip2, twolf, wupwise, swim, mgrid, applu, mesa, galgel, art, equake, facerec, ammp, lucas, fma3d, sixtrack, apsi, mcf
SPEC95 Benchmark
li95, hydro2d95, fpppp95, gcc95, tomcatv95, turbo95, go95, su2cor95, wave95

Table 1: Benchmarks used in this study.

Parameter	Value
Fetch width	8 instructions per cycle
Fetch policy	ICOUNT.2.8 [32]
Pipeline depth	8 stages
Min branch misprediction penalty	6 cycles
Branch predictor	2K gshare
Branch Target Buffer	256 entry, 4-way associative
Active List Entries	256 per thread
Functional Units	6 Integer (4 also load/store), 3 FP
Instruction Queues	64 entries (32 int, 32 fp)
Registers For Renaming	100 Int, 100 FP
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache	512 KB, 2-way, 64-byte lines
L3 Cache	4 MB, 2-way, 64-byte lines
Latency from previous level (with no contention)	L2 10 cycles L3 20 cycles Memory 100 cycles

Table 2: Processor configuration.

than 1 it has to be that more instructions are executed than would be the case if each job simply contributed instructions in proportion to its single-threaded IPC.

A short exercise may make WS(t) even more intuitive; if we have one job with single threaded IPC of 2 and another with single threaded IPC of 1 and run them separately each for 1 million cycles then one will have executed 2 million instructions and the other 1 million. Now, if we instead coschedule them for 1 million cycles and the first contributes 1 million instructions and the second contributes 500 thousand then WS(t) will equal 1. This makes sense because the total number of instructions executed was exactly what would be predicted by the natural IPC of each and their fair share of the machine (1/2) when scheduled together. However, if machine utilization goes up due to coscheduling (which is the primary aim and purpose of multithreading to begin with) then we may see something like 1.2 million instructions executed on behalf of the first job and 600 thousand on behalf of the other for a total WS(t) of 1.2. It is also possible for WS(t) to be less than 1 if coscheduled jobs interact in pathological ways [24].

Figure 2 illustrates the danger of using IPC as the objective function. It shows the worst and best IPC observed when 8 jobs are coscheduled 4 at a time on an SMT machine. The jobs are Swim, Mgrid, Applu, Li95, Hydro2d95, Fppp95, Gcc95, and Tomcatv95 from Table 2. The columns represent different choices as to which sets of 4 jobs run together in the same timeslice. IPC varies by as much as 8% in this case, depending on which jobs run together. It is not clear though that a schedule with (possibly temporarily) higher IPC is actually making more rapid progress through an entire jobmix than is a schedule with lower IPC. It could just be preferencing jobs with high solo IPC. Figure 3 shows the worst and best $WS(t)$ observed in the same experiment. Again there is an 8% variation; however, WS(t) varies in different ways across the workload than IPC. We have found weighted speedup correlates well with response-time improvements in open-system experiments – this experiment shows that IPC is not particularly well correlated with that metric.

In the next section we report results in terms of WS(t), which has been shown to fairly measure throughput progress through the jobmix.

5. IMPLEMENTING PRIORITIES WITH SOS

We explore a spectrum of hardware and software that can support priorities on SMT. First we try an obvious way of implementing priority on SMT, then we explore more complicated schemes that could boost throughput while preserving schedule viability.

Recall (from Section 4) that there are 2 different possible meanings of priority:

(A) guarantee a fraction of machine proportional to priority

(B) guarantee a fraction of single-threaded performance proportional to priority

These are indistinguishable implementation-wise on single-threaded machines but possibly different on multithreaded machines.

5.1 A simple priority mechanism

The first priority mechanism we implement is based on the assumption that a job coscheduled to execute with P other jobs gets $1/P$ of the system. We call the scheduling mechanism *Naive*. Based on the assumption, high priority jobs are coscheduled for CS of the time and solo scheduled for $SOLOFRAC_i$ of the time (these

fractions were defined in Section 3). The result is a viable schedule in the sense that jobs get an opportunity to issue instructions proportional to their priority (barring unlikely anti-symbiotic behavior). So *Naive* preserves property (B) above if the assumption holds, though it violates property (A). Also, one can expect throughput due to multithreading to be good during the CS timeslice. Therefore this simple scheduling mechanism works well with SMT hardware. Figure 4 shows the division of job throughput and total job throughput that results from an example of the simplest possible case (coscheduling 2 jobs) using *Naive*. In this case Hydro2d95 and Li95 are run together according to different priorities and resulting fractions of single-threaded throughput guarantees ranging from 90% for Lisp95 and 10% for Hydro2d95 to 90% for Hydro2d95 and 10% for Lisp95. The heights of the bars are all greater than 1, meaning all of the *Naive* schedules with these priority ratios result in an increase in throughput due to multithreading. However, throughput drops significantly 'on the wings' where the priorities are most different. This is because the high priority thread is solo-scheduled a great deal of the time in these cases, so opportunities for symbiosis are reduced.

5.2 A symbiotic priority mechanism

The next priority mechanism we implement is based on the idea that if we can observe how well jobs *symbios* we can do better than *Naive*. SOS (Sample, Optimize, Symbios) is a CPU scheduler that dynamically discovers efficient schedules on the fly and runs these to boost throughput and turnaround. In the sample phase it runs jobs together in different combinations (when $P > M$) to determine the efficiency of groups of candidates for coscheduling; also it can run jobs by themselves to determine their 'natural' IPC as a baseline. After sampling, the scheduler makes a scheduling decision based on observed performance counters (the optimize phase) and then runs a predicted-to-be optimal schedule for a relatively long time (compared to the sample phase) before again sampling to capture changes in jobmix and job execution phase.

The ability of SOS to determine the 'natural' IPC of jobs allows it to enforce priorities by comparing a job's progress to its estimated progress if run by itself. This allows precise definition and implementation of priorities in a multithreaded environment. If a job starts falling behind it can be solo-scheduled for a while to catch up. In fact, SOS implements a notion of priority that is *more* precise than that usually found on single-threaded machines. Traditional machines simply enforce property (A) above via timeslicing with no notion of the natural rate of progress that a job makes if it has the machine to itself, the assumption being that jobs on a timeshared machine do not affect each other. This assumption can be false due to cache or other side effects.

The *Symb* mechanism implements priorities by first doing a sample phase among the runnable jobs to estimate their solo IPC. It then samples the performance of the jobs when run in an (arbitrary) set of groups for equal timeslices at the highest level of multithreading possible. It next uses the information gathered to compute $COFRAC_i'$ and CS' from Section 4. Typically $CS' >= CS$. Every so often it repeats the sample phase to capture changes in 'natural' solo IPC. So *Symb* determines how jobs utilize the system when run alone and in groups and uses this information to schedule the machine as efficiently as possible while still preserving priorities. Typically we can expect *Symb* to outperform *Naive* unless the cost in reduced performance during the solo-scheduled sample phase exceeds the benefits conferred by increasing the timeslice (CS') in which the machine is fully coscheduled. Figure 5 shows the division of job throughput and total job throughput that results from an example of the simplest possible case (coscheduling 2 jobs) using *Symb*. Comparing this to Figure 4 we see that throughput has improved significantly 'on the wings' because *Symb* takes into account how much progress jobs actually make when coscheduled rather than making a naive assumption about it. Since one job typically does not use all available issue slots, there is a symbiotic effect resulting in increased throughput due to coscheduling. *Symb* quantifies this and uses it to coschedule as much as possible while still preserving schedule viability.

5.3 A priority mechanism requiring hardware support

The next priority mechanism we implement is based on the idea that if we could control job issue rate cycle-by-cycle according to priority, we could coschedule more often. The *Icount* mechanism uses additional hardware that was proposed in [32] to support biased fetching of instructions to maximize parallelism in the instruction window. Threads which are least represented in the pre-execute stages of the pipeline have preference for fetch. Here we take advantage of the existence of that hardware – we give fetch preference to threads of higher priority. Each thread receives a handicap number equal to p_i where p_i is its priority number. A job with higher handicap gets fewer fetch opportunities than a job with lower handicap. This enables enforcement of priorities on a cycle-by-cycle basis. This comes very close to enforcing property (A) above except that if the priority job cannot issue, a lower priority job can 'sneak in' and issue. A job thus gets a fraction of fetch opportunities proportional to its $FRAC_i$. In most cases this increases CS' to nearly 1 (little or no solo-scheduling) since a coscheduled low priority job cannot disproportionably interfere with a high priority job. We may expect *Icount* to outperform the software disciplines in cases where it significantly increases CS'. We will see below however that this 'micromanagement' approach may be counterproductive in cases where CS' is already high. However Figure 6 shows that, in the simple case, *Icount* significantly increases throughput 'on the wings' where priorities are most different.

In the above experiments, after accounting for sampling overhead, *Symb* outperforms *Naive* by as much as 20%, as little as 0% (when the priorities are equal), and by an average of 8% over the points sampled. While an 8% performance boost may seem modest it is also cheap. It is brought about by simply adding a bit of sophistication to the software CPU scheduler. Also recall that on a multi-user system it is common to have low priority batch jobs and high priority interactive jobs. So the case where priorities are quite different is a common case. And the symbiotic schemes confer the most benefit in this scenario. *Icount* further boosts performance by as much as 30%, as little as 0%, and an average of 15%. *Icount* requires additional hardware counters to keep track of the count of issued but not-yet-retired instruction per thread and a mechanism for preferential fetch. However, this hardware has been proposed to increase performance for SMT machines in general. We exploit it in another way but with the same result.

5.4 Priority mechanisms for multiprogrammed environments

The next priority mechanisms implemented only apply when there are more runnable jobs in the system than there are hardware contexts available to coschedule them. They are based on the idea that some combinations of jobs 'get along' better than others so it

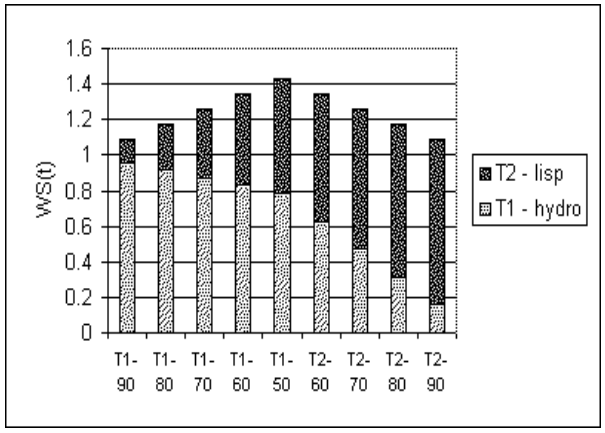

Figure 4: Division of throughput for *Hydro* and *Lisp* with various priorities using the *Naive* priority scheme.

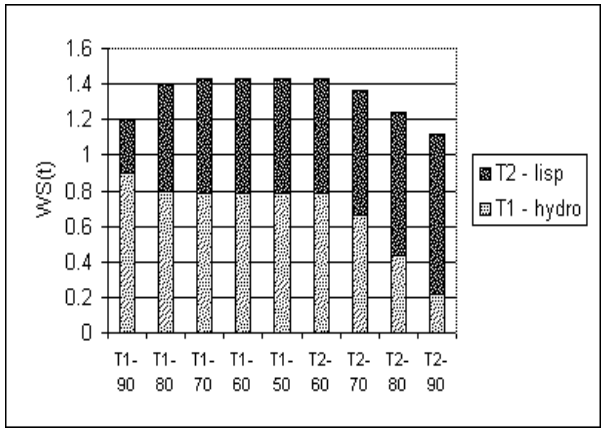

Figure 5: Division of throughput for *Hydro* and *Lisp* with various priorities using the *Symb* priority scheme.

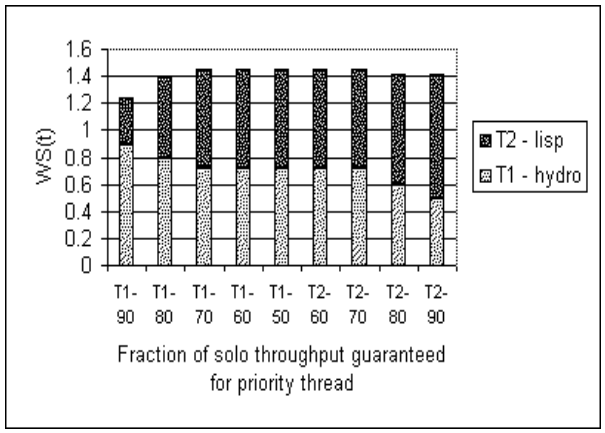

Figure 6: Division of throughput for *Hydro* and *Lisp* with various priorities using the *Icount* priority scheme.

makes a difference which jobs are coscheduled. The *SOS* mechanism works similar to the *Symb* mechanism, except that when $P > M$ it increases the length of the sample phase, trying several different ways of grouping jobs to run together. It uses observations about the solo IPC of jobs to evaluate which schedule has the highest $WS(t)$. It is important to note though that increasing the length of the sample phase to try several different coschedules in this way does *not* increase the overhead associated with sampling (!) because, in the absence of better information, the scheduler is constrained to co-schedule according to *some* grouping. So trying different groupings is no worse than simply picking one at random and progress is still being made through the jobs during the sample phase. Typically, we can expect *SOS* to do better than *Symb* if some ways of groupings jobs to run together are better than others (the symbiosis effect).

SOS can also be combined with *Icount*. The last priority mechanism we implement combines the two. We will see in the next section, however, that controlling priorities through the fetch mechanism is not always most efficient. It may be *unnecessary* to severely handicap a low priority job from being fetched. It may prove more efficient (and still viable) to let it in at a somewhat higher rate if it is not interfering unduly with high priority threads (i.e. is highly symbiotic). *Icount-SOS* handicaps lower priority threads in hardware but only as much as is necessary to enforce priority. It combines sampling to determine 'natural' IPC with handicapping to suppress fetch from low priority threads. When the standard deviation among job priorities is high it degenerates to (essentially) *Icount*. When the standard deviation is low, it degenerates to *SOS*. Typically, we may expect *Icount-SOS* to strikes a dynamic balance between the best throughput achievable with hardware alone and that with software alone.

Figure 7 shows an example of the common case where there are more jobs to be run than there are available hardware contexts to hold their state. In this case there are 8 jobs (from SPEC2000 and SPEC95 *Turbo, Gcc, Go, Hydro, Su2cor, Swim, Tomcatv,* and *Wave*). We simulate an SMT processor with 4 hardware contexts; the scheduler runs 4 jobs at a time for 5M cycles and then has a swap opportunity when it may replace one or more of the jobs in the running set with another runnable job. In this experiment one job is chosen to have high priority and the rest compete among themselves at equal (lower) priority. As before, the benefits of the more sophisticated methods (taken in increasing order of sophistication and complexity, *Naive, Symb, SOS,* and *Icount-SOS*) are greater for greater differences in priority. Over the points measured *Symb* improves on *Naive* by as much as 17% in terms of $WS(t)$ and averages 8% better. *SOS* improves by as much as 23% over *Naive* and averages 14% better. *Icount* improves by as much as 37% and averages 19% better. Figure 7 is fairly typical of results when there are about twice as many runnable jobs as hardware contexts. Although in this case *Icount* did better than either *SOS* or *Symb*, we will see in the next section that when there are lots of jobs in the system and less standard deviation among their priorities, the software methods may do better than the hardware methods. *Icount-SOS* combines the best features of both and does as well as the better of either (in this case the same as *Icount*).

6. RESPONSE TIME

Here we summarize results from many experiments that model the general case where multiprogramming is high relative to hardware available for multithreading and jobs come and go in the system. In order to model a system under load we implement a system where

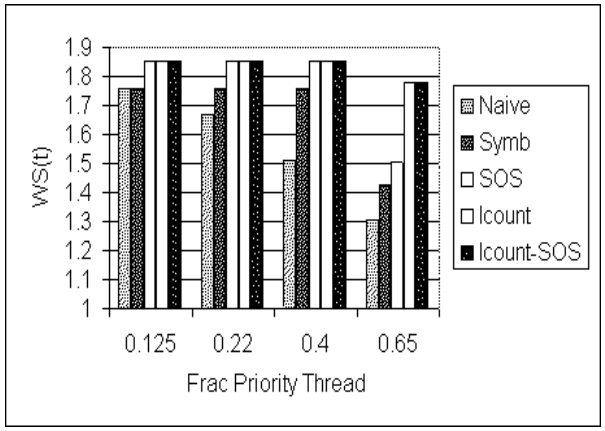

Figure 7: Performance of *Naive, Symb, SOS, Icount,* and *Icount-SOS* for 8 threads coscheduled 4 at a time.

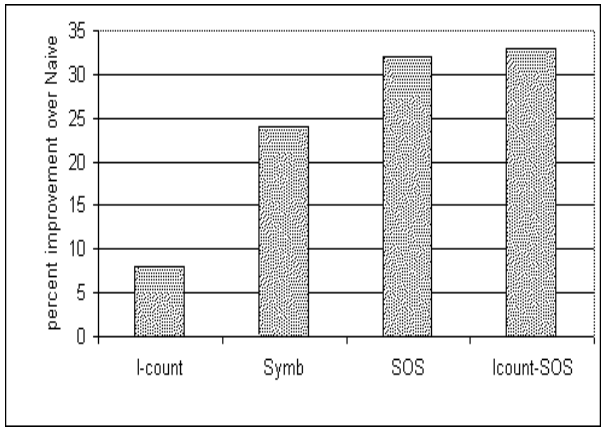

Figure 8: Average percent improvement in turnaround time over *Naive* for *Symb, SOS, Icount,* and *Icount-SOS* with a multithreading level of 8.

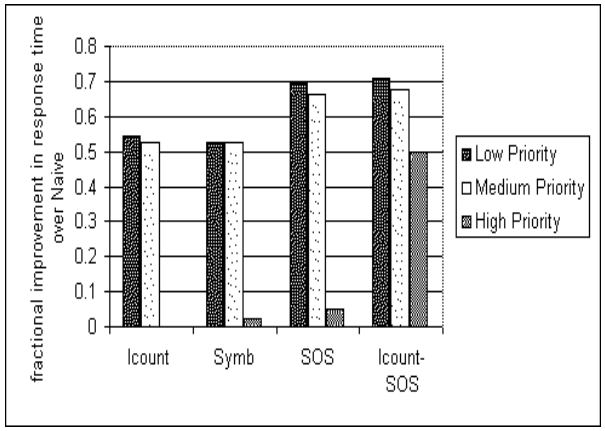

Figure 9: Percent improvement in turnaround over *Naive* depending on priority category.

jobs enter and leave the system with exponentially distributed arrival rate λ and exponentially distributed average time to complete a job T. We further associate a randomly generated priority with each job that comes into the system; jobs have an equal likelyhood of getting any priority in the range 0 to 19. We study a stable system where λ and T are such that the number of jobs in the system (N) does not grow without bound. In such a system it makes sense to measure response time rather than throughput, since throughput cannot possibly exceed the rate of job arrival. If two stable systems are compared and one is faster, the faster one will complete jobs more quickly and thus typically have fewer queued up waiting to run.

The jobs are drawn from Table 1. We randomly generate jobs with an average distribution of T centered around 2 billion cycles by first generating random numbers with this distribution and then fetching that many instructions multiplied by single-threaded IPC from the jobs. So, for the purposes of these experiments, a job is about 2 billion cycles worth of instructions from one of the SPEC95 or SPEC2000 benchmarks.

We use a job arrival rate (λ) with an exponential distribution that will cause the system to remain stable with N about equal to double the SMT level (based on Little's law [16] $N = \lambda * T$). So most of the time there are about N = 2 * SMT-level jobs in the system. To model a random system but produce repeatable results, we fed the same jobs in the same order with the same arrival times to the scheduler.

Figure 8 shows the percent improvement in turnaround time achieved over repeated trials at a multithreading level of 8 by our various priority disciplines.

Notice that in this case, contrary to the results of Figure 7, *Icount* did worse than *SOS* or *Symb*. *Icount* is particularly effective at enforcing priorities when there is a great standard deviation in priorities among jobs in the system. The other disciplines are unable to avoid a substantial amount of solo-scheduling in that scenario. When there are many jobs of many different priorities in the system, no particular job is entitled to a large slice of the system. Under these conditions it appears that *Icount* unnecessarily limits the fetching of independent and heterogeneous instructions from low priority threads. But *Icount-SOS* combines the cycle-by-cycle priority enforcement of *Icount* with the flexibility and dynamic optimizing behavior of *SOS* to obtain the best performance of either (in this case the same as *SOS*, 33%.)

It is reasonable to wonder if increases in average turnaround are coming at the expense of turnaround for high priority jobs. However, this is not the case. Even though in the short term we take resources from high-priority jobs, and make them available to low-priority jobs, this short-term effect is overcome by the long-term benefit of increasing system throughput, and accelerating the completion of low-priority jobs out of the system, decreasing the average queue length. Figure 9 shows the fractional improvement in response time over *Naive* that resulted from the experiment. Jobs are divided into 3 bins; Low Priority (priorities in the range 12 to 19), Medium Priority (priorities in the range 6 to 11), and High Priority (priorities in the range 0 to 5). The lower priority jobs benefited most but the high priority jobs also turned around faster (Icount showed a very small but non-zero improvement for the High Priority bin).

7. PREVIOUS WORK

A simultaneous multithreading processor [33, 32, 17, 13, 35] holds the state of multiple threads (execution contexts) in hardware, allowing the execution of instructions from multiple threads each cycle on a wide superscalar processor. This organization results in more than doubling the throughput of the processor without excessive increases in hardware [32].

The techniques described here also apply to other multithreaded architectures [3, 10, 2]; however, the SMT architecture is most relevant here because threads interact at such a fine granularity in the architecture, and because it is closest to widespread commercial use. By contrast, the Tera MTA supercomputer [3], which features fine-grain multithreading, has fewer shared system resources and less intimate interactions between threads. It issues one LIW instruction per cycle, does not support out-of-order execution, does not have shared renaming registers, and has no data cache.

Snavely, et al., [24] first used the term symbiosis to refer to an increase in throughput that can occur when particular jobs are coscheduled on multithreaded machines, and in [23] exhibit a user-level schedule that boosts throughput on the Tera MTA. But that application is for a massively parallel system which largely protects threads from each other. Thus, while the scale of the scheduling problem is great, the number of factors determining how threads interact are few and relatively straight-forward.

Sobalvarro and Weihl [27], Gupta, et al., [11], and Dusseau, et al., [4] all explore the benefits of coscheduling parallel jobs based on their communication patterns. In fact, a multiprocessor scheduler should solve a similar problem – how to coschedule threads on different processors to maximize efficiency in the face of bottlenecks on shared system resource (such as main memory or communication fabric). Chapin [7] emphasizes load balancing, as does Tucker and Gupta [31]; the idea is to migrate threads to underutilized processors. Others have concentrated on keeping the cache warm by favoring the mapping of threads to processors where they have executed before [6] [30] [34].

Coscheduling on traditional single-threaded architectures often leads to increased throughput due to overlapping of I/O from some job(s) with the calculations of others. The scheduling discipline *Multilevel Feedback*, implemented in several flavors of Unix[29], 4.3 BSD Unix, Unix System V, and Solaris TS (timesharing scheduling class), encourages I/O bound jobs to run more frequently, thus leading to higher overall machine utilization. I/O bound jobs tend to relinquish the CPU as soon as they obtain it. If the hardware and O/S support asynchronous I/O, this allows the CPU to stay busy with the next job while I/O is serviced ([28] [15]). Patterson and Gibson [18] describe an extension to the Mach O/S that does informed prefetching to exploit I/O parallelism in coscheduled jobs to boost throughput on a DEC workstation with multiple SCSI strings.

Several systems schedule software threads on single-threaded processors or clusters of single-threaded processors. Delany [9] explains how the Daylight Multithreading Toolkit Interface does this to overlap I/O with computation and increase system throughput. Blumofe and Leiserson [5] describes a method for scheduling software threads on a hardware single-threaded multiprocessor via a *workstealing* heuristic.

Many scheduling techniques strive to coschedule jobs that communicate frequently on massively parallel (MPP) systems conglomerated from single-threaded processors. Sistare et al. [22] describe a system that dynamically coschedules jobs that communicate frequently to increase system utilization and job response time. Sobalvarro et al. [26] improve upon gang scheduling to dynamically produce emergent coscheduling of the processes constituting a parallel job. Silva and Scherson [21] improve upon gang scheduling to fill holes in utilization around gang scheduled jobs with pieces of work from jobs that do not require all resources in order to to make progress. Lee et al. [14] evolve methods of balancing the demands of parallel jobs waiting to be gang scheduled with those of I/O-bound jobs, which require high CPU priority to achieve interactive response times. The goal is to keep the system highly utilized.

Several works have explored the tension between scheduling a system for high utilization and meeting an objective function on single-threaded hardware devoted to a real-time mix of jobs. Hamidzadeh and Atif [12] account for the scheduling overhead in a system that dynamically schedules real-time applications with a goal of using a multiprocessor single-threaded system efficiently to meet the maximum number of deadlines.

Coffer et al. [8] describe scheduling mechanisms allowing the system administrator to balance the demand for fast turnaround with demand for high throughput. The administrator can over-allocate resources to allow high utilization of system resources on the Origin 2000.

Schauser et al. [20] describes a hierarchical scheduling policy that allowed the TAM machine to schedule logically related threads closely together in time.

Previous work focused on coarse-grained overlapping of I/O with computation on single-threaded hardware, or concentrated on ways to coschedule logically related jobs on MPP systems conglomerated from single-threaded hardware, or focused on mechanisms to pack low priority jobs around high priority jobs to raise utilization on hardware-single-threaded machines. We previously [25] considered OS mechanisms for increasing fine-grained, overlapping, resource utilization on hardware-multithreaded machines for jobs that run well together but have no other reason to be coscheduled. This work considers how to make the scheme work when one must balance decisions that the scheduler would make in terms of jobs to co-execute, with preferences of the user as to how much of the system should be devoted to high or low priority jobs under co-execution. So, while most previous work only takes into account communication interaction and the need to coschedule parallel jobs, this work incorporates much more complex interactions between coscheduled jobs and the desires of the user. Many of these interactions are phenomena particular to multithreaded systems.

8. CONCLUSION

This paper shows that the benefits of multithreading, and those of symbiotic jobscheduling, need not be sacrificed to enforce priorities. A clever jobscheduler keeps track of how much resource a job is consuming, how much its priority entitles it to, how well the job gets along with other coscheduled jobs, and uses this information to make scheduling decisions accordingly.

SOS uses a sample phase in which it collects information about how jobs run when executed by themselves and in groups to produce efficient schedules that keep the machine well utilized while preserving the semantics of priority. SOS works gracefully with hardware support for enforcing priorities. Our results indicated that

coscheduling priority jobs can increase system throughput by as much as 40%. The benefit depends upon the relative priority of the coscheduled jobs, and more complex schedulers are more effective with greater differences in priorities. We also showed that, although our priority schedulers focus on increasing system throughput, as a result they can also decrease average turnaround times for a random jobmix by as much as 33%.

9. REFERENCES

[1] http://developer.intel.com/technology/hyperthread/.

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[4] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Sigmetrics*, 1998.

[5] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994.

[6] R. Chandra, S. Devine, and B. Verghese. Scheduling and page migration for multiprocessor computer servers. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[7] S. Chapin. Distributed and multiprocessor scheduling. *ACM Computing surveys*, Mar. 1996.

[8] H. Cofer, N. Camp, and R. Gomperts. Turnaround vs. throughput: Optimal utilization of a multiprocessor system. In *SGI Technical Reports*, May 1999.

[9] J. Delany. Daylight multithreading toolkit interface. *http://www.daylight.com*, May 1999.

[10] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, Nov. 1995.

[11] A. Gupta, A. Ticker, and S. Urushibara. The impact of operating scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 392–403, June 1999.

[12] B. Hamidzadeh and Y. Atif. Dynamic scheduling of real-time aperiodic tasks on multiprocessor architectures. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, Oct. 1999.

[13] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[14] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of i/o for gang scheduled workloads. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1997.

[15] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[16] J. Little. A simple proof of the queuing formula L = λ W. *Operations Research*, 9:383–387, 1961.

[17] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. In *ACM Transactions on Computer Systems*, Aug. 1997.

[18] H. Patterson and G. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of Third International Conference on Parallel and Distributed Information Systems*, Sept. 1994.

[19] W. Pfeiffer, L. Carter, A. Snavely, R. Leary, A. Majumdar, S. Brunett, J. Feo, B. Koblenz, L. Stern, J. Manke, and T. Boggess. Evaluation of a multithreaded architecture for defense applications. In *SDSC Techical Report*, June 1999.

[20] K. Schauser, D. Culler, and E. Thorsten. Compiler-controlled multithreading for lenient parallel languages. In *Proceedings of FPCA '91 Conference on Functional Programming Languages and Computer Architecture*, July 1991.

[21] F. Silva and I. Scherson. Improving throughput and utilization in parallel machines through concurrent gang. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, May 2000.

[22] S. Sistare, N. Nevin, T. Kimball, and E. Loh. Coscheduling mpi jobs using the spin daemon. In *SC 99*, Nov. 1999.

[23] A. Snavely and L. Carter. Symbiotic jobscheduling on the MTA. In *Workshop on Multi-Threaded Execution, Architecture, and Compilers*, Jan. 2000.

[24] A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Explorations in symbiosis on two multithreaded architectures. In *Workshop on Multi-Threaded Execution, Architecture, and Compilers*, Jan. 1999.

[25] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[26] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. In *SRC Technical Note 1997-017*, Mar. 1997.

[27] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the IPPS 1995 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, Apr. 1995.

[28] K. Thompson. Unix implementation. In *The Bell System Technical Journal*, July 1978.

[29] K. Thompson and D. Ritchie. The unix time-sharing system. In *Communications of the ACM*, July 1974.

[30] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling issues for multiprogrammed shared memory multi-processors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

[31] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared memory multiprocessors. In *Symposium on Operating Systems Principals*, Dec. 1989.

[32] D. 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 *ISCA96*, pages 191–202, May 1996.

[33] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[34] R. Vaswani and J. Zahorjan. The implications of cache-affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Symposium on Operating Systems Principals*, Oct. 1991.

[35] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.