Published in the Proceedings of ASPLOS IX, November 2000

Symbiotic Jobscheduling for a Simultaneous Multithreading Processor

Allan Snavely
University of California, San Diego
9500 Gilman Drive
La Jolla, California 92093-0505
allans@sdsc.edu

Dean M. Tullsen
University of California, San Diego
9500 Gilman Drive
La Jolla, California 92093-0114
tullsen@cs.ucsd.edu

ABSTRACT

Simultaneous Multithreading machines fetch and execute instructions from multiple instruction streams to increase system utilization and speedup the execution of jobs. When there are more jobs in the system than there is hardware to support simultaneous execution, the operating system scheduler must choose the set of jobs to coschedule

This paper demonstrates that performance on a hardware multithreaded processor is sensitive to the set of jobs that are coscheduled by the operating system jobscheduler. Thus, the full benefits of SMT hardware can only be achieved if the scheduler is aware of thread interactions. Here, a mechanism is presented that allows the scheduler to significantly raise the performance of SMT architectures. This is done without any advance knowledge of a workload's characteristics, using sampling to identify jobs which run well together.

We demonstrate an SMT jobscheduler called SOS. SOS combines an overhead-free sample phase which collects information about various possible schedules, and a symbiosis phase which uses that information to predict which schedule will provide the best performance. We show that a small sample of the possible schedules is sufficient to identify a good schedule quickly. On a system with random job arrivals and departures, response time is improved as much as 17% over a schedule which does not incorporate symbiosis.

1. INTRODUCTION

Simultaneous Multithreading (SMT) [32, 31, 18] 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; jobs move in and out of the running set, at the discretion of the OS, at a much coarser granularity. In this situation, the jobscheduler decides which jobs should be coscheduled in the running set.

The term *symbiosis* has been used to refer to the effectiveness with which multiple jobs achieve speedup when coexecuted on multithreaded machines [24]. Jobs in an SMT processor can conflict with each other on various shared system resources. Throughput may actually 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.

This paper presents a symbiotic OS-level jobscheduler for SMT that dynamically adjusts its scheduling decisions to enhance throughput and lower response time. The scheduler begins to coschedule jobs according to some fairness policy. By sampling hardware performance counters, and by periodically randomly perturbing the sets of coscheduled jobs, it discovers a job schedule for the entire jobmix that increases system performance over what would be expected if scheduling were left to chance. We call our scheduler 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 can improve system response time by as much as 17% over a naive scheduler.

We first describe our methodology and experimental setup in Section 3, then introduce a metric in Section 4 to measure progress through the execution of a jobmix. Section 5 shows how SOS discovers efficient coschedules using runtime counters to predict which combinations of jobs will run well together, and that it exhibits gains in throughput on several jobmixes. Sections 6 and 7 discuss the performance of SOS on jobmixes which include parallel multithreaded programs, and Section 8 explores cache coldstart effects related to these techniques. Lastly, Section 9 shows that SOS improves response time as well as throughput on a system with random job arrivals and runtimes.

2. RELATED WORK

A simultaneous multithreading processor [32, 31, 18, 14, 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 has the potential to more than double the throughput of the processor without excessive increases in hardware [31].

The techniques described here also apply to other multithreaded architectures [3, 11, 2]; however, the SMT architecture is most interesting because threads interact at such a fine granularity in the architecture, and because it is closest to widespread commercial use, having been announced for the next Alpha processor [10]. 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 [26], Gupta, et al., [12], 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 [30]; the idea is to migrate threads to under-utilized processors. Others have concentrated on keeping the cache warm by favoring the mapping of threads to processors where they have executed before [6] [29] [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 *Multi-level Feedback*, implemented in several flavors of Unix ([28] 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 ([27] [16]). Patterson [19] describes 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 [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 [22] describes a system that dynamically coschedules jobs that communicate frequently to increase system utilization and job response time. Sobalvarro [25] improves upon gang scheduling to dynamically produce emergent coscheduling of the processes constituting a parallel job. Silva [21] improves 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 [15] evolves 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 [13] accounts 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 [8] describes 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 [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 harwdware, or focused on mechanisms to pack low priority jobs around high priority jobs to raise utilization on hardware-single-threaded machines. This work breaks new ground by considering O/S 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. 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. Many of these interactions are phenomena particular to multithreaded systems.

3. EXPERIMENTAL SETUP

An SMT processor comes equipped with some number of hardware contexts (roughly a program counter and a set of state-holding registers). The number of contexts determines how many threads can be co-executed; this number is referred to as the multithreading (or SMT) level. The jobscheduler selects, from the pool of jobs ready to run, a number of jobs to coschedule less than or equal to the multithreading level. Every so often, for fairness, this running set is swapped out and replaced with a new set of jobs from the ready pool.

Our simulator (based on SMTSIM [33]) models an out-of-order processor based on the Compaq Alpha 21264 with modest hardware additions to support multithreading. The 21264 comes equipped with performance counters which can be used to capture dynamic execution information. We model 21264 instruction latencies, functional units (fully pipelined), sizes of instruction queues, sizes and associativities of caches, and TLB capacity. This is not a particularly aggressive architecture for this study, but provides some resource contention even at the most modest levels of multithreading. The same effects demonstrated in this work will be evident with wider processors, but may happen at higher levels of multithreading. In the experiments detailed below, we use hardware multithreading levels of 2, 3, 4, and 6. Because our simulator, despite its detail, still simplifies the interactions between CPU resources, we expect results on real SMT machines to produce more potential gain with symbiotic scheduling than demonstrated here.

We present our jobscheduler with multiprogrammed workloads made up of single-threaded and multithreaded jobs from the SPEC INT, SPEC FP, and NPB (NAS Parallel Benchmarks [1]), and a parallel program (ARRAY) which does a parallel prefix operation on an array. We assume the jobscheduler is required to make progress through all of the jobs in a strictly fair manner; all jobs must be scheduled on the CPU for the same number of cycles over the course of a run. Every 5 million cycles, which would correspond to a 10 millisecond timer interrupt on a 500 MHz system, the jobscheduler receives a clock pulse; if runnable jobs are available that were not scheduled during the previous timeslice, it swaps out one or more of the jobs that ran in the last timeslice, replacing these with jobs that did not.

A schedule is a covering set of coschedules such that every job appears in an equal number of coschedules. The jobs that make up a coschedule compete with each other for system resources cycle by cycle during their scheduled timeslice. Our goal is to find a schedule that exhibits the highest average speedup across all jobs.

The jobscheduler runs in two phases called Sample and Symbios. In what follows, we label experiments by a tuple Jmn(X,Y,Z) where X is the number of runnable jobs, Y the multithreading level, and Z the number of running jobs swapped out and replaced with jobs from the runnable pool at the expiration of the timeslice. 'm' is a character from $\{s,p\}$. An s indicates a multiprogrammed workload made up of single-threaded applications and a p indicates that the workload includes parallel (multithreaded) jobs. 'n' is a character from b,l where b(ig) indicates that a timeslice of 5 million cycles was used for coschedules and l(ittle) indicates that a smaller timeslice was used. So for example, Jsb(6,3,1) is a jobmix of 6 single-threaded jobs that are run 3 at a time; at the end of 5M cycles (b) 1 job is swapped out and replaced with 1 job that was not running. Since jobs are selected to be swapped FIFO, this makes the effective resident timeslice for each job equal to 15M cycles. As a further example, Jpb(10,2,2) is a jobmix of 10 jobs, some of which (ARRAY) are parallel jobs; jobs are run 2 at a time and the entire running set (2 jobs) is swapped out every 5M cycles. J2pb(10,2,2) is a *different* jobmix run the same way as Jpb(10,2,2) (see Section 6).

The exact jobs used in each throughput experiment are given in Table 1. The jobmix for each experiment is meant to provide computational diversity. Each jobmix has a combination of high IPC floating point programs typical of scientific computing (FP, MG, FT etc.) and lower-IPC integer intensive codes typical of workstation tasks (GCC, GO etc.). No particular effort was made to represent jobs evenly; FT for example, only appears in one jobmix while FP appears several times.

In each experiment we compare the performance of 10 different schedules. We run for a number of cycles in the sample phase sufficient to profile 10 schedules and then for 2 billion cycles in the symbios phase. The number of cycles required to profile 10 schedules given a timeslice of 5 million cycles varies according to the size of the jobmix, the multithreading level and the job replacement policy. So the number of cycles spent in the sample phase depends on the experiment and is given in Table 2. In the sample phase the jobscheduler randomly permutes the sets of coscheduled jobs and records dynamic execution information to observe how the coschedules are performing.

In all but one of our experiments, the jobscheduler generates and evaluates 10 random schedules in the sample phase. For Jsb(4,2,2) there are only 3 possible schedules. See Table 2. Schedules are represented by permutations of X job identifiers starting at 0 parsed by underbars to delineate coschedules. So, for example, 012_ 345 is the schedule for Jsb(6,3,3) (6 jobs taken 3 at a time) that runs jobs 0, 1, and 2 together as a tuple and then swaps these out replacing them with jobs 3, 4, and 5. We consider jobschedules to be identical if they coschedule the same tuples regardless of the order in which the tuples are scheduled. All jobs are required to run before any job runs again, and we swap a fixed number of jobs per timeslice. This means that average pressure on the memory subsystem is the same regardless of tuple order. Furthermore, cache-sweeping interaction between jobs cannot be avoided simply by changing the order of tuples; every schedule is a circular sequence of tuples; a job that sweeps another job's cache does so in every schedule. In Section 8, we examine the interaction of symbiosis scheduling and cache coldstart effects more closely.

We begin simulation with each benchmark partially executed. This avoids phase changes at the beginning of execution, but in a real system, at most one benchmark in a sample phase will typically be starting up.

4. WEIGHTED SPEEDUP

When jobs are co-executed 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.

Experiments	Jobs
Jsb(4,2,2)	FP,MG,GCC,IS
Jsb(5,2,2),Jsl(5,2,1)	FP,MG,WAVE,GCC,GO
Jpb(10,2,2),J2pb(10,2,2)	FP,MG,WAVE,SWIM,SU2COR,TURB3D,GCC,GCC,ARRAY,ARRAY
Jsb(6,3,3),Jsb(6,3,1),Jsl(6,3,1)	FP,MG,WAVE,GCC,GCC,GO
Jsb(8,4,4), Jsb(8,4,1), Jsl(8,4,1)	FP,MG,WAVE,SWIM,GCC,GCC,GO,IS
Jsb(12,6,6), Jsb(12,4,4)	FP, MG, WAVE, SWIM, SU2COR, TURB3D, GCC, GCC, GO, IS, CG, EP
SMT level 2	CG,mt_ARRAY,EP
SMT level 3	FP,MG,WAVE,mt_EP,CG
SMT level 4	FP,MG,WAVE,mt_ARRAY,EP,CG
SMT level 6	FP,MG,WAVE,GO,IS,GCC,mt_ARRAY,EP,CG,FT

Table 1: The set of applications used in all experiments in this paper. FP is fpppp and MG is mgrid from SPEC95.

Experiment	Distinct Schedules	Million Sample Cycles
Jsb(4,2,2)	3	30
Jsb(5,2,2)	12	250
Jsb(5,2,1)	12	250
Jpb(10,2,2)	945	250
J2pb(10,2,2)	945	250
Jsb(6,3,3)	10	100
Jsb(6,3,1)	60	300
Jsl(6,3,1)	60	100
Jsb(8,4,4)	35	100
Jsb(8,4,1)	2520	400
Jsl(8,4,1)	2520	100
Jsb(12,4,4)	5775	150
Jsb(12,6,6)	462	100

Table 2: The number of distinct possible schedules for each jobmix, and the time to run at most 10 schedules during the sample phase.

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 a while, 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. In fact, WS(t) of a time-shared single-threaded system, even with unfair scheduling, is still 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 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 might hope to 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 1 shows the worst and best weighted speedup observed when 13 different combinations of jobmix, SMT multithreading level, and job replacement policy are run with permuted coschedules. Weighted speedup varies, depending on which jobs run simultaneously. The symbiotic (or anti-symbiotic) behavior of jobs causes speedup to vary by an average of 8% and a maximum of 25%, even for the limited number of samples we take. Clearly, performance is quite sensitive to the actual schedule in all cases, and the potential for a symbiosis-sensitive jobscheduler is significant.

5. SOS

With SOS, the jobscheduler begins to run jobs in groups equal to the multithreading level, using some fair policy to allow all of the runnable jobs to make progress. The initial phase is called the sample phase. Here the scheduler permutes the schedule periodically, changing the jobs that are coscheduled. As it proceeds in the sample phase, SOS gathers dynamic execution profiles of the jobs being run by referencing hardware performance counters. After sampling the performance of several schedule permutations, SOS picks one that it thinks will be optimal and proceeds to run it

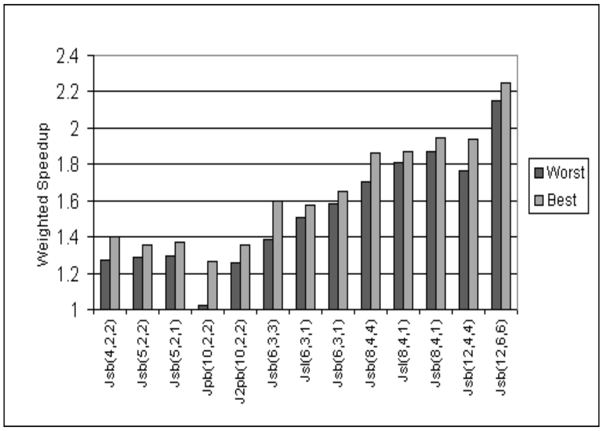

Figure 1: Worst and best weighted speedup of several jobmixes, multithreading levels, and job replacement policies.

in the symbios phase. The only overhead is the occasional reading and resetting of the counters, which is typically necessary infrequently.

The optimal ratio for the durations of the symbios phase and the sample phase depends on how often the jobmix landscape changes. From time to time, jobs will terminate and new jobs will enter the system. Jobs will naturally pass through different phases of execution where their resource utilization and IPC profiles change. We begin with experiments where the ratio of symbios to sample is approximately 10 to 1 and the jobmix stays constant during that interval. Section 9 models a more realistic system with random job arrivals and departures.

5.1 Shared System Resources and Predicting Future Performance

Our system needs the ability to find an accurate predictor of future performance of jobschedules from a current snapshot of counters. Hardware resources that can be shared among the running threads of the SMT cycle by cycle include functional units, instruction queues, caches and memory and interconnections between them, the TLB, renaming registers, and branch prediction tables. As part of sharing and competing for these resources, threads will sometimes interact in ways that contribute to enhanced throughput and sometimes will not. When one thread uses a system resource that would otherwise have gone unused, system utilization and thus system throughput goes up. But if threads conflict on resources, utilization can drop.

We can conjecture that symbiosis is a function of 3 interrelated attributes of the jobmix and schedule:

Diversity

The instructions in the window of instructions being considered for execution on the current cycle should be diverse. Because the goal is to keep all of the functional units as busy as possible, we need instructions for each.

Balance

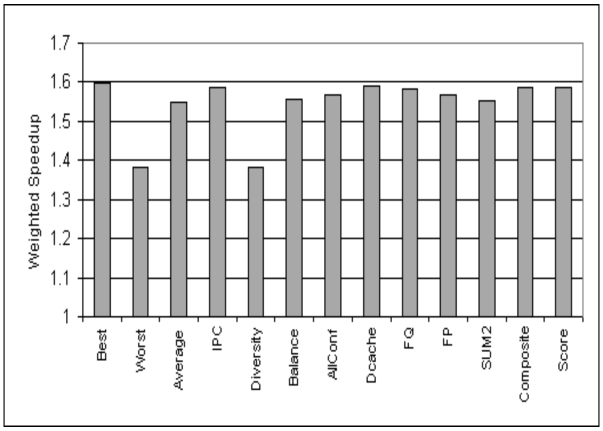

Figure 2: Weighted speedup achieved with several dynamic predictors on Jsb(6,3,3).

A schedule that alternates timeslices of over-subscription with timeslices of under-subscription is unlikely to outperform a more balanced schedule. When the system is under-subscribed, utilization and throughput are low. When the system is over-subscribed conflicts are high, and little benefit accrues to having *more* than sufficient work for the functional units; the resources are oversaturated in one timeslice and under-utilized in the next.

Low Conflicts

Between two fair schedules for the same jobs, one with lower conflicts is likely to perform better. Conflicts can lower system utilization. Furthermore, conflicts are correlated to the two previous attributes. Diverse instructions do not conflict; smooth schedules lessen conflicts by load-balancing demand for resources.

The validity of these conjectures is explored next.

5.2 An Example - Jsb(6,3,3)

Table 3 shows dynamic predictor data gathered by SOS in the sample phase for a jobmix of 6 threads with a multithreading level of 3. The possible ways of dividing 6 threads into 2 sets of 3 are enumerated in the first column. There are only 10 possible fixed schedules of 6 jobs when coscheduling 3 jobs at a time and replacing all 3 each timeslice. In the sample phase we run each possible schedule for 10 million cycles (the minimum time required to evaluate the schedule with a swap timeslice granularity of 5 million cycles). So, in this case, after a 100 million cycle sample phase, we have run all possible schedules and have predictions for the performance of all possible schedules. Columns 2 through 9 are predictors gathered in the 100 million cycle sample phase. The best score in each column is shown in bold font; the predictors can be used to guess how a schedule will perform in the subsequent 2 billion cycle symbios phase. The last column is the weighted speedup of each schedule in the 2 billion cycle symbios phase.

Figure 2 shows weighted speedups obtained in the symbios phase by SOS, using the several different predictors of Table 3 and a vote tallying one (Score). The predictors are:

Schedule	IPC	AllConf	Dcache	FQ	FP	Sum2	Diversity	Balance	Composite	WS(t)
012_345	3.007	146.14	97.5	37.04	17.36	54.4	**0.15**	0.24	1.45	1.38
013_245	3.266	146.6	97.5	9.68	31.66	41.34	0.18	**0.10**	3.30	1.56
014_325	2.865	**129.52**	97.5	20.77	**16.74**	37.51	0.17	0.61	1.15	1.57
015_342	3.223	147.72	97.6	9.06	32.09	41.15	0.18	0.86	2.14	1.52
023_145	3.321	146.14	**98.1**	7.51	28.93	36.44	0.18	0.27	2.90	1.59
024_315	3.462	140.4	97.4	8.6	17.73	26.33	0.18	0.21	2.73	**1.60**
025_341	3.453	140.07	97.4	6.69	16.82	**23.51**	0.17	0.55	2.93	1.55
034_125	3.28	140.52	97.6	7.61	22.73	30.34	0.18	1.34	2.47	1.53
035_124	3.333	139.82	97.4	**6.42**	21.7	28.12	0.17	0.52	3.06	1.58
045_123	**3.532**	158.45	97.9	6.8	31.02	37.82	0.16	0.13	**3.70**	1.59

Table 3: Detailed results for jobmix Jsb(6,3,3), including performance data collected by various predictors during the sample phase, and the weighted speedup of that schedule in the symbios phase.

IPC

A schedule with observed high IPC in the sampling phase is predicted to be highly symbiotic.

AllConf

A schedule with a low sum of conflicts on the integer queue, the floating point queue, the integer renaming registers, the floating point renaming registers, scoreboard entries , integer units, floating point unit and load store units in the sampling phase is predicted to be highly symbiotic. We sum the percentages of cycles for which the schedule conflicts on each of these resources. The schedule with the lowest sum is deemed best.

Dcache

A schedule with a high overall hit-rate in the L1 data cache is predicted to be highly symbiotic.

FQ

A schedule with low total conflicts on the floating point queue is predicted to be highly symbiotic. A queue conflict arises when instructions cannot be placed in the queue because it is full.

FP

A schedule with low conflicts on the floating point units is predicted to be highly symbiotic.

Sum2

A schedule with a low sum of conflicts on the floating point units and the floating point queue is predicted to be highly symbiotic.

Diversity

A schedule with a diverse mix of instructions in all of its timeslices is predicted to be highly symbiotic. The schedule with the lowest absolute difference between percentage of floating point and integer instructions is deemed best.

Balance

A schedule with little variation in IPC between consecutive timeslices is predicted to be symbiotic. The schedule with the lowest standard deviation in IPC between coschedules is deemed best.

Composite

A schedule with the *highest* score of

$$\frac{0.9}{MIN\{(FQ/LowestFQ),(FP/LowestFP),(SUM2/LowestSUM2)\}} + \frac{0.1}{Balance}$$

is predicted to be highly symbiotic. The *Lowest* terms are the lowest of these values observed for any of the schedules in the sample phase. This predictor is an experimental fit that correlates data gathered in the sample phase to later performance. It gives most weight to load balance but some weight to low conflicts on critical resources. It was developed using data for Jsb(6,3,3). This predictor in the sample phase is highly correlated to performance in the symbios phase. The next section will evaluate the same predictor on other jobmixes.

Score

A schedule which is voted best by the majority of the other predictors is predicted best. Ties are broken by relative magnitude of *goodness* predicted.

The first bar in Figure 2 gives the highest weighted speedup obtained by any of the 10 possible schedules, and the second bar gives the lowest. There is a 17% difference between the two. The third bar is the average weighted speedup of all 10 schedules and can be thought of as the *expected* throughput that an oblivious jobscheduler would obtain. The best schedule is 9% better in terms of weighted speedup than the average. If a smart jobscheduler can find the best schedule, it can boost throughput. The rest of the bars show the weighted speedup obtained by SOS using the various dynamic predictors to guess at good schedules.

In this experiment, all but one of the predictors (Diversity) avoided the worst schedule. IPC, Dcache, FQ, Composite, and Score all achieved within 2% of the best schedule (a 9% gain over the expected value of speedup). For this particular jobmix, then, we can expect as much as a 9% performance gain over the average schedule a naive jobscheduler would choose, and up to 17% over an unlucky schedule choice. The next section shows that similar results are achieved over a much wider range of workloads and architectures.

5.3 Other Jobmixes

Figure 3 shows the weighted speedup achieved with SOS scheduling for 13 combinations of jobs, multithreading levels and job replacement policies. The jobs in each jobmix can be found in Table 1. In each case but one, we have sampled 10 schedules (Jsb(4,2,2) has only 3 possible schedules). We then predict which would be best using the dynamic predictors gathered in the sample phase and then run all 10 for 2 billion cycles in the symbios phase to see how they actually perform. Thus, we use the information gathered in the sample phase to predict the performance of 10 random schedules and then proceed to run each one to validate our guesses. Note that while there are only 10 possible schedules for Jsb(6,3,3), there are many more for other experiments (see Table 2) so that 10 schedules is just a statistical sample of the performance space in most cases. Figure 3 confirms our finding that 10 random schedules of any given jobmix produced a substantial difference between a best and a worst schedule and even between a best and an average schedule, and thus was sufficient to identify a *good* schedule. More sampling would likely find a better schedule, but at a greater cost for the sampling phase. It also shows that SOS can discover a schedule with better than average performance using dynamic predictors and that some dynamic predictors are better than others.

One interesting result is that IPC alone is not a particularly good predictor. This is non-intuitive, since high IPC means high system utilization, which is essentially the goal of this work. We found IPC to be quite variable, even over the granularity of the timeslices we used. Other metrics were typically much more stable from sample to sample of the same mix, and thus often provided better predictions of future performance. Also, a temporarily high IPC schedule does not always equate to the highest system-throughput schedule; it can happen that high IPC threads monopolize system resources to the detriment of low IPC threads in the sample phase.

The Diversity predictor is not effective; the variance in the Diversity predictor was not very great for these experiments. It likely was insufficiently large to manifest as an important effect.

The Balance predictor is quite effective. Good schedules balance the demand for system resources across timeslices. The fact that a schedule is balanced is a good indication that it will deliver high throughput.

The AllConf predictor was not very effective; paradoxically, high conflicts are often a symptom of high system utilization. However, conflicts on the floating point queue and floating point unit are especially to be avoided in our processor model; low scores for the FP, FQ, and SUM2 predictor are correlated to good performance.

The Dcache predictor was an inconsistent performer. In some cases it chose the *worst* schedule. All of the kernels get good cache reuse, and none are large enough to seriously stress the capacity of the cache even when run in combination. The hit rate is high and there is little variation in this predictor between schedules.

The Composite predictor, which uses criteria of smoothness and low conflicts, is the most consistent individual performer. We tried several composite predictors. Intuitively it might seem that a predictor that gives more weight to events that detract the most from performance (e.g. dcache misses) might work well. However we did not find a correlation between the conflict penalty and the weight that should be given to it in a composite predictor. Conflicts only cause a drop in throughput if *no* job can make progress. Otherwise, a loss to one job is a gain to another and may not negatively impact weighted speedup.

Score, which tallies votes from all of the other predictors is the best overall performer. It appears that symbiosis leading to increased throughput is a function of several factors. Score usually discovers the best of the 10 schedules. Ignoring for a moment the gains exhibited for Jpb(10,2,2) (a special case that will be explored in the next section), SOS with score predictor boosts weighted speedup by 22% over unlucky schedules and by 7% over the expected value of random schedules.

The best predictors will, in general, be very architecture-dependent. We have discovered effective ones for our particular simulation of SMT. The important result is that good predictors are not difficult to find, and they do have an impact on performance.

6. PARALLEL WORKLOAD SCHEDULING

It can be seen in Figure 3 that the most dramatic gain for SOS was for Jpb(10,2,2), where the score-predicted schedule increased the gains due to multithreading over the average by almost 400%. However this is an artifact of random scheduling. ARRAY does tight synchronization between its threads. If these threads are not coscheduled, very poor performance results. Most of the random schedules did not coschedule the threads of ARRAY. However, any reasonably designed jobscheduler for a multithreaded system will likely coschedule parent and child threads from the same job. SOS dynamically determines that parent and child threads should run together in this case. However we do not feel justified in claiming the 400% gain as a breakthrough result! On the other hand, if the parent and child thread do not communicate often, coscheduling may not be the best option. J2pb(10,2,2) uses a variant of ARRAY that does little synchronization. In this case, the score-predicted schedule does *not* coschedule the parent and child threads of ARRAY and outperforms one that does by 13%.

Thus we see the advantage of a dynamic scheduler such as SOS over a scheduler which uses a rule-of-thumb *always schedule parent and child threads together*. SOS determines whether it makes sense to schedule threads together for performance. Not all of our SOS predictors chose the right combination. If we were to depend on SOS to schedule parallel jobs, we would want to make sure that IPC was a significant

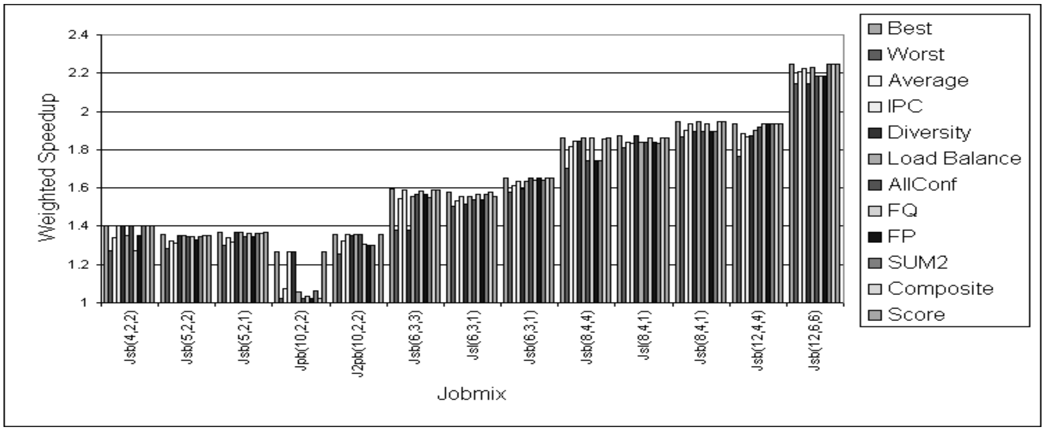

Figure 3: Weighted speedup achieved by SOS for several different jobmixes.

component of the prediction.

7. HIERARCHICAL SYMBIOSIS

As automatic multithreading compiler technologies mature, SMT workloads will include more multithreaded jobs like ARRAY. If the compiler is sophisticated enough to generate code that can adapt to the number of hardware contexts available at runtime (the Tera MTA compiler does this), then the jobscheduler has an additional degree of freedom when allocating resources–it can decide how many contexts to assign to each multithreaded job. Consider a machine with an SMT level of 3 and a workload that includes two multithreaded jobs, ARRAY and a multithreaded version of EP. Assuming that SOS decides to coschedule ARRAY and EP, it can also decide whether to devote 2 contexts to ARRAY and 1 to EP or vice-versa (of course, it could also keep both single-threaded and coschedule a third job.) It turns out that the coschedule that devotes 2 contexts to ARRAY is 8% more symbiotic than the complementary coschedule. If the jobmix is exactly EP and ARRAY, we can consider a third possibility, alternating 3 threads of EP with 3 of ARRAY. However, this last schedule is 9% worse than the best.

SOS could implement symbiosis at 2 levels by deciding which jobs to coschedule and then deciding how many contexts to give multithreaded jobs. In the absence of an MTA-like compiler for SMT, we have hand-coded several multithreaded versions of the benchmarks with different levels of multithreading and evaluated the potential benefits of giving the jobscheduler this extra degree of freedom. Interestingly, although the above division of resources is optimal for just EP and ARRAY together, it is not necessarily optimal in a larger schedule. If the jobmix is single-threaded CG with multithreaded EP and ARRAY on a machine with an SMT level of 4, the optimal schedule is 1 context for CG, 2 for EP, and 1 for ARRAY. The resources used by a job impact every other coscheduled job. In this case, adding one extra job to the mix changes the optimal resource allocation between the first two. SOS could heuristically approximate a solution to this global optimization problem by trying different ways of dividing up contexts in the sample phase.

We extend the definition of WS(t) to include multithreaded jobs by making the denominator term equal to the issue rate of the job running alone, with no other jobs in the coschedule. Figure 4 shows the average percent improvement in weighted speedup achievable by SOS using the score predictor for various levels of SMT if, besides deciding which threads to coschedule, SOS determines how many threads to devote to each parallel job. It can be seen that these two levels of choice allow SOS a significant advantage over random (average) or unlucky (worst) schedules. The *SMT level* entries in Table 1 give the jobs used in these experiments.

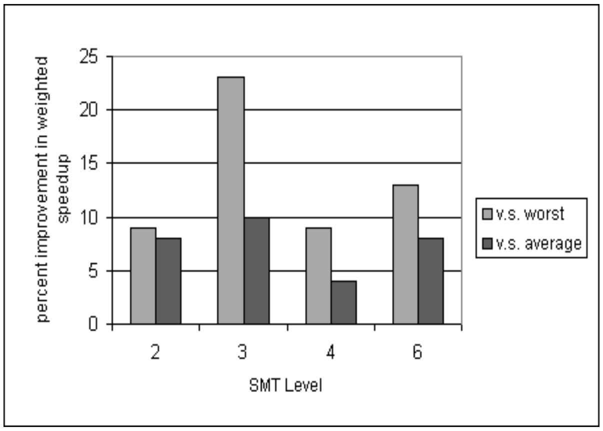

Figure 4: Improvements in weighted speedup potentially achievable by SOS using hierarchical symbiosis with different levels of multithreading.

8. WARMSTART SCHEDULING

In example Jsb(6,3,3), we used a scheduling policy that replaced all of the members of the running set each timeslice; a thread had fixed partners in a schedule. This limits the number of possible schedules and the time required to sample a schedule, thus making exhaustive sampling possible. But a system may prefer to swap only one job at a time to reduce pressure on the memory subsystem. Furthermore, while our experiments have focused on CPU-bound jobmixes with infrequent I/O, a system may use an I/O event as an opportunity to swap the job out, bring in a new job, and sample a new coschedule. SOS applies equally well to a

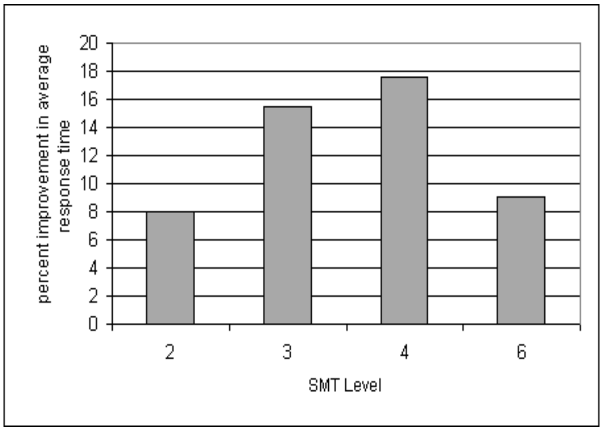

Figure 5: Response time improvements obtained by SOS over a random jobscheduler for various levels of multithreading.

scheme where only one job is swapped at a time (which we call warmstart scheduling); SOS samples a number of such schedules and picks a good one based on observation.

Benefits of warmstart scheduling can be seen by comparing Jsb(5,2,2) to Jsb(5,2,1) of Figure 3 and also by comparing Jsb(6,3,3) to Jsb(6,3,1) and Jsl(6,3,1) and also by comparing Jsb(8,4,4) to Jsb(8,4,1) and Jsl(8,4,1). Recall that for the experiments with Z (the last number in the triple) = 1, we swapped only 1 job per timeslice. There are two effects likely to increase utilization associated with this style of scheduling. First, the resident timeslice for a single thread increases. Second, pressure on the memory subsystem during a context switch decreases. The first effect is well known. It is particularly important on an SMT processor. If only 1 job is cold-starting, the other jobs can hide those latencies more effectively. The longer a job stays resident the better it amortizes the cost of warming up the memory subsystem. The experiments labeled with 'n' = b benefit from both effects. To isolate the effect of simply reducing the pressure on the memory subsystem by swapping only one job at a time, we shortened the swap timeslice in the experiments labeled with 'n' = l where Z = 1. It can be seen that there is a modest gain in symbiosis associated with warmstart scheduling. It averages 7% for the experiments labeled 'n' = b but is negligible in the other cases.

The scheduling policy can change cold-start behavior. More importantly, for this study, results show symbiosis scheduling is effective for both scheduling policies that attempt to minimize cache coldstarts and those that do not.

9. RESAMPLING AND RESPONSE TIME

Because job resource utilization profiles do not remain static over time, and because jobs come and go as they enter the system and complete, it is necessary to repeat the sample phase from time to time. There is tension in choosing the rate of resampling. We want to sample as little as possible because we want to maximize the ratio of the duration of the symbios phase to the sample phase. This allows us to amortize the cost of sampling. But we want to sample often to catch changes in job execution profiles and jobmix. The system we have designed will adjust the duration of each phase dynamically. If the jobmix is observed to be changing rapidly (things have changed a lot since the previous sample phase), sampling frequency goes up. If the jobmix seems stable, sampling frequency goes down.

We model a system where jobs enter and leave the system with exponentially distributed arrival rate λ and exponentially distributed average time to complete a job T. 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.

We randomly generated 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 of Table 1. So, for the purposes of these experiments, a job is about 2 billion cycles worth of instructions from one of the jobs in Table 1. We then used a job arrival rate (λ) with an exponential distribution that would cause the system to remain stable with N about equal to double the SMT level (based on Little's law [17] $N = \lambda * T$). 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 SOS and a *control group* scheduler. The *control group* scheduler is a random, or naive, scheduler in the sense that it simply coschedules jobs together in tuples equal to the SMT level in the order in which they arrive. SOS, on the other hand, selects a symbiotic schedule based on sampling. Three events trigger a new sample phase: a job arrival, a job departure, or the expiration of the symbiosis phase timer. For these experiments we used λ as the default symbiosis interval. If no new job arrives after λ cycles and the new prediction is the same as the old one, SOS employs exponential backoff by doubling the time it will run before sampling in the absence of a new job. When a job arrives or departs, or the new prediction does not agree with the old one, the duration of the symbiosis phase reverts to λ. For these experiments both the SOS scheduler and the baseline scheduler swap out all jobs from the running set each timeslice, when possible.

Figure 5 compares the average response time delivered by a random jobscheduler to that delivered by SOS for four different levels of SMT multithreading, with response time improvement varying from 8 to nearly 18%. The response time improvement includes the performance of the sampling phases, when no speedup is expected. Figure 6 shows response time improvements for SOS over a random scheduler with various job arrival rates and the SMT multithreading level held constant at 3. The arrival rates shown are the mean of an exponential distribution centered at the value indicated. The improvements shown in Figure 6 differ from those of Figure 5 for SMT level of 3 simply because the ex-

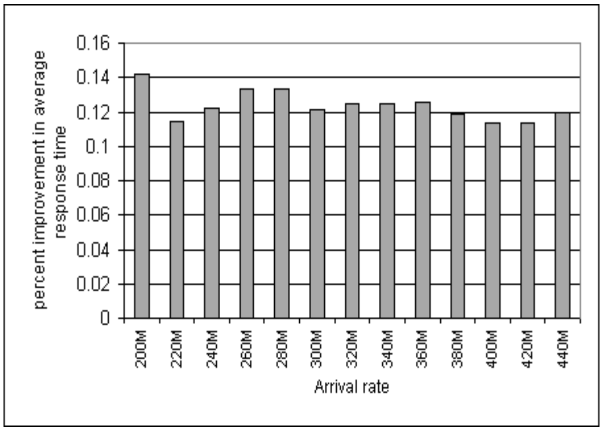

Figure 6: Response time improvements obtained by SOS over a random jobscheduler for various values of λ (in cycles); with SMT level held constant at 3.

periments are different in each case, with different jobs, random job lengths, random orders of arrival, and random rates of arrival. After many such experiments, we have concluded that: 1) SOS boosts response time substantially. 2) It is always worthwhile resampling when a new job comes in. The old schedule 'adjusted' to accommodate a new job is typically no better than random. Thus, we should go straight to the sampling phase. 3) Resource utilization profiles of the SPEC and NPB benchmarks are quite stable. In most of our experiments, resampling when no new job arrived or old job terminated did not result in dramatic gains. But this is an artifact of the stability of the SPEC and NPB benchmarks in terms of resource utilization. We expect other workloads will experience more *phased* behavior.

10. CONCLUSION

This paper demonstrates that performance on a multithreaded processor is sensitive to the set of jobs that are coscheduled by the operating system jobscheduler. It presents a mechanism that allows the scheduler to exploit this phenomenon to arrive at a schedule which can significantly improve performance. This is done without any advance knowledge of an application's characteristics, using sampling to identify jobs which run well together. It can even identify multithreaded (parallel) jobs that perform better if not coscheduled.

SOS combines a sample phase which collects information about various possible schedules, and a symbiosis phase which uses that information to predict which schedule will provide the best performance. We show that a small sample of the possible schedules is sufficient to identify a good schedule quickly. There is no cost to the sample phase, because the performance of the sample phase is equivalent to what would be expected of a naive scheduler. On a system with random job arrivals and departures, response time is improved as much as 17% over a schedule which does not incorporate symbiosis.

11. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their useful comments. We thank Larry Carter; symbiosis was his brainchild and he has provided guidance and good suggestions throughout. We thank Wayne Pfeiffer for his support. We thank Geoff Voelker for his careful reading and thoughtful observations on the work in progress. This work was supported in part by NSF CAREER grant No. MIP-9701708 and an equipment grant from Compaq Computer Corporation. This work was also supported in part by NSF award ASC-9613855 and DARPA contract DABT63-97-C-0028.

12. REFERENCES

[1] http://science.nas.nasa.gov/software/npb.

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. 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/meetings/mug99/Delany/mt/reentrant.htm* May 1999.

[10] K. Diefendorff. Compaq chooses smt for alpha. *Microprocessor Report*, 13(16), Dec. 1999.

[11] 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.

[12] A. Gupta, A. Ticker, and S. Urushibara. The impact of operating scheduling policies and synchronization methods on the performance of parallel applications. In *Signetrics*, pages 392–403, June 1999.

[13] 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.

[14] 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 *isca92*, pages 136–145, May 1992.

[15] 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.

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

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

[18] 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.

[19] 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.

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

[26] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *IPPS95*, pages 63–75, Apr. 1995.

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

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

[29] J. Torrelas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling issues for multiprogrammed shared memory multi-processors. In *1993 ACM Sigmetrics*, May 1993.

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

[31] 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.

[32] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA95*, pages 392–403, June 1995.

[33] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[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.