A Non-Work-Conserving Operating System Scheduler For SMT Processors

Alexandra Fedorova†‡, Margo Seltzer† and Michael D. Smith†

†Harvard University, ‡Sun Microsystems

ABSTRACT

Simultaneous multithreading (SMT) processors run multiple threads simultaneously on a single processing core, and the threads compete for the processor's shared resources. Severe resource contention can lead to performance degradation. On SMT processors it is often beneficial to employ a *non-work-conserving* scheduling policy: running fewer threads simultaneously than the processor allows even if there are threads ready to run. In cases of severe resource contention, non-work-conserving scheduling can alleviate the contention significantly enough so as to result in better performance than if the processor were utilized to the full extent. Conventional operating systems typically do not employ non-work-conserving policies. We present a prototype of an operating system thread scheduler that uses a non-work-conserving policy whenever it may result in better performance. To determine when to use the non-work-conserving policy, the scheduler uses an analytical model that, unlike existing models, is sufficiently simple and practical for use inside the operating system. We demonstrate that the scheduler using our model correctly determines when to use the non-work-conserving policy and improves performance in those cases.

1. INTRODUCTION

An SMT processor is equipped with multiple hardware contexts, or *virtual processors*, that enable concurrent execution of multiple threads [1-4,6,22,23]. Although SMT processors differ from conventional processors, it is common to use the operating systems designed for conventional processors on SMT systems. Conventional operating systems are *work-conserving*: they schedule a thread to run on every virtual processor, never leaving a virtual processor idle if there are threads ready to run. On SMT processors work-conserving scheduling can result in sub-optimal performance [12]. The threads running simultaneously contend for the SMT processor's shared resources, and if contention becomes too great the threads may complete altogether less work than they would if some virtual processors were left idle and the competition for the shared resources were less severe. In systems this phenomenon is known as *thrashing*. To prevent thrashing, a scheduler must be able to predict whether using a *non-work-conserving* policy [21,25], i.e., keeping some virtual processors idle even if there are threads ready to run, might result in better performance.

We present a user-level prototype of a non-work-conserving operating system scheduler. At the heart of the scheduler is a new analytical model that helps decide when it is beneficial to employ non-work-conserving scheduling. The model estimates the instructions per cycle (IPC), or throughput, of the workload for a given *degree of concurrency*, i.e. the numbers of threads running simultaneously. The scheduler uses this model to determine the degree of concurrency yielding the highest IPC, and then uses only as many virtual processors as needed to achieve the optimal degree of concurrency.

The analytical model is the key contribution of our work. Similar models proposed in the past [16,17] were designed for offline studies of SMT architectures. Their main objective was accuracy. Efficiency, i.e. the number of steps it takes to produce the IPC estimates using the model, was less important. It was also acceptable to obtain the inputs for the model offline.

In contrast, our model is designed specifically for use inside an operating system scheduler. The scheduler is a performance critical component of the operating system invoked hundreds of times per second, and any computation that it performs must be quick, in order to impose little overhead on performance of user applications. Therefore, our model has two critical requirements of simplicity and practicality: a) the model calculations must be simple enough so they could be performed in a small number of steps and implemented without floating-point operations (many modern operating systems do not permit floating-point operations inside the kernel); b) inputs for the model must be obtainable at runtime or at compilation time.

We achieve simplicity in the model by focusing on those resources, contention for which is most likely to cause thrashing. We model them precisely, while representing the effects of contention for other resources with simple, less precise, models. We identify contention for the L2 cache as the prime cause of performance degradation and thrashing. Our decision is based on previous studies that have shown that contention for the L2 cache can significantly affect throughput on SMT processors [8,9,31] (also see Figures 1a and 1b). Contention for the L2 cache increases the miss rate, requiring more accesses of the main memory. The cost of memory access has now reached 200-300 processor cycles and has been growing at 50% per year [24]. It is, therefore, probable that thrashing due to L2 cache contention remains a problem in the future, and this is another reason why we focus on the L2 cache. Our model precisely expresses the relationship between the L2 cache miss rate and the IPC; we approximate contention for other resources with models derived using linear regression, which significantly simplifies our model. To foster simplicity we model the effect of the L2 cache miss rate on the IPC using basic probability theory, which results in the model that is notably less complex than those based on Markov chains, used in the past [16,17].

We assume the SMT architecture of the UltraSPARC T1® [6] processor released by Sun Microsystems in 2006, and we validate our model for this architecture. This is a first step towards creating a general model. We compare our model's IPC predictions for the SPEC CPU2000 integer benchmarks to actual measurements from an execution-driven simulator of UltraSPARC T1 [5], and find them to be accurate to within 10% for most workloads, with the largest observed difference between predicted and measured values of 31%. This accuracy is comparable to results previously reported in literature for more complex off-line models [16,17].

With our prototype we demonstrate that a scheduler using our model is able to correctly determine the optimal degree of concurrency in most cases. We show that applications using our scheduler achieve throughput improvement of 3 to 56% when

there is benefit to be gained from non-work-conserving scheduling, and perform no worse than with the default scheduler otherwise. In this study we focus on *homogeneous workloads* – multithreaded workloads where all threads are performing similar work. Such workloads are common in data mining and bioinformatics [28-30].

The rest of the paper is organized as follows: In Section 2 we describe our target SMT architecture and the simulator used for our experiments. In Section 3 we present the model and evaluate its accuracy. In Section 4 we present the scheduler prototype and the performance results. In Section 5 we discuss related work, and in Section 6 we conclude.

2. THE SMT ARCHITECTURE

On modern SMT processors switching among the running threads occurs if more than one thread requires exclusive use of a shared resource [6,22,23]. Our model is designed for the UltraSPARC T1 [4,6] architecture where all components of a single-issue pipeline are shared, and switching occurs on every CPU clock cycle. We assume a single-core SMT processor with four virtual processors, shared first-level (L1) instruction and data caches, and a second-level (L2) unified cache. The L2 cache is connected to main memory (DRAM) by a shared bus. We assume a single core, because our model's novelty is in representing the interaction among the threads inside the core. We believe our model can be easily extended to work for multi-core architectures.

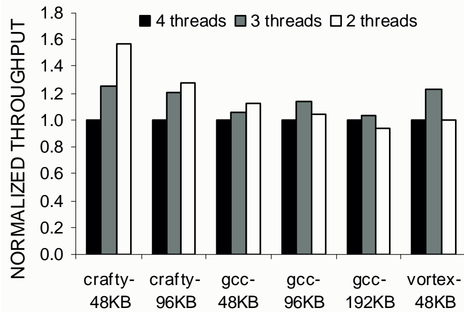

Figure 1a. Normalized throughput for memory-intensive SPEC CPU2000 benchmarks. The experiment is run on a simulated SMT processor with four virtual processors. There is a set of bars for each benchmark/L2 cache size. Each bar within a set corresponds to a different number of concurrent threads. Each thread runs its own benchmark instance. Using a high degree of concurrency causes thrashing due to extreme contention for the L2 cache (see Figure 1b). Lowering the degree of concurrency eliminates thrashing and improves the throughput.

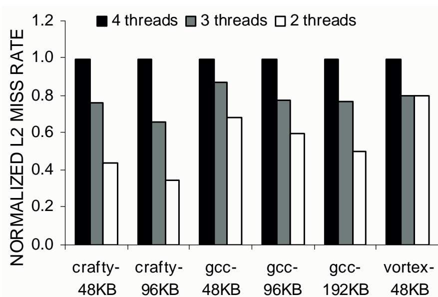

Figure 1b. Normalized L2 miss rate corresponding to the experiments in Figure 1a.

Processor	992 MHz, single processing core, 4 virtual processors
L1 data cache	Shared, 8KB, 4-way set associative
L1 instruction cache	Shared, 16KB, 4-way set associative
L2 cache	Shared, 12-way set associative, size varies from 48KB to 192KB. (We use the L2 cache size that is smaller than a typical L2 cache on modern SMT processors because we eventually want to target architectures that have multiple SMT cores sharing the L2 cache [6,35], and on these architectures, the amount of L2 cache per core is smaller.)
Memory bandwidth	5.2 GB/s

Table 1. Simulator configuration.

We run our experiments on an execution-driven full-system simulator [5] of UltraSPARC T1. Using a simulator allowed us to validate our model for various hardware configurations. We validated the simulator against the UltraSPARC T1 hardware. Table 1 summarizes the architectural characteristics of the simulated processor. The configuration differs from that of UltraSPARC T1 in the number of cores (we simulate a single core, while the T1 has eight), the memory bandwidth, and the clock speed (we could not model the precise clock speed of the T1 due to simulator constraints). Our simulator can execute the Solaris™ operating system and all applications that commonly run on Solaris/SPARC® platforms.

3. THE MODEL

Processor's cycles per instruction can be broken down into two components: 1) the busy cycles 2) the blocked cycles: the cycles spend handling L2 cache misses. We define the first component as *ideal cycles per instruction (ideal CPI)* – the CPI when there are no non-compulsory L2 cache misses: this is when the processor runs at its *ideal IPC*. The second component is determined by the L2 cache miss rate. And so we express the IPC as the function of the ideal IPC and the L2 cache miss rate:

$$IPC = F(ideal_IPC, L2\ miss\ rate)$$

Since our goal is to prevent thrashing due to contention for the L2 cache, we precisely model the effect of the L2 cache miss rate on the IPC. Contention for other shared resources, for which we use simpler models, is captured by *ideal_IPC*. In this section we describe how we model the IPC based on the L2 miss rate, assuming that the ideal IPC is known. We explain how to approximate the ideal IPC in Section 4.

Our model requires the following parameters:

Architectural parameters – obtained on system configuration:

- the number of virtual processors,
- DRAM latency,
- L2 cache size,
- memory bus bandwidth.

Workload parameters – measured at runtime and provided by a compiler:

- the number of concurrently running threads
- ideal IPC – the IPC with an infinitely-sized L2 cache.
- L2 cache miss rate for a given number of concurrently running threads.

In Section 4 we explain how we obtain the workload parameters at runtime.

For a machine with given architectural parameters, our model is expressed as follows:

$$IPC = F(N, L2\ miss\ rate, ideal_IPC(N)),$$

where N is the number of concurrent threads.

We begin by describing the IPC model for a processor running in a single-threaded mode (Section 3.1), then we extend the model for the multithreaded mode (Section 3.2).

3.1 Single-threaded model

For simplicity we explain how we model *cycles per instruction* (CPI), the inverse of *instructions per cycle* (IPC).

$$IPC = \frac{1}{CPI} \quad (1)$$

First, we introduce some definitions:

L2_CPI – for a given L2 miss rate, the number of cycles per instruction that a thread is blocked handling misses in the L2 cache.

Ideal_CPI – the inverse of ideal IPC, the cycles per instruction achieved with an infinitely sized L2 cache.

L2_RMR – L2 read miss rate – the number of L2 read misses per instruction. This includes data read misses and instruction fetch misses.

L2_WMR – the number of L2 write misses per instruction.

L2_MCOST – the cost, in cycles, of handling a miss in the L2 cache. This includes the memory latency and the memory-bus delay. While the memory latency of a given architecture is fixed, the memory-bus delay varies depending on the workload. We now assume that memory-bus delay is zero, and relax this assumption at the end of Section 3.

L2_CPI depends on the L2 miss rate and the cost of each miss:

$$L2_CPI = (L2_RMR + L2_WMR) * L2_MCOST \quad (2)$$

The CPI is comprised of its *ideal_CPI* and its *L2_CPI*:

$$CPI = ideal_CPI + L2_CPI \quad (3)$$

On processors with write-back caches, such as UltraSPARC T1, a cache transaction may trigger a *write-back*: an operation of writing a dirty cache line back to the memory. A write-back may be triggered on a cache read miss or on a write miss. As will become clear soon, we must distinguish among those write-back operations triggered by read misses and those triggered by write misses. We express the respective write-back rates as follows:

L2_WBR_R – the write-back rate (operations per instruction) triggered by read and instruction-fetch misses.

L2_WBR_W – the write-back rate (operations per instruction) triggered by write misses.

We found that the fraction of write-back operations triggered by each type of cache misses can be accurately approximated by the fraction of the corresponding cache misses: for example, if 60% of the cache misses are read misses, then roughly 60% of the write-back transactions are triggered by the read misses.

To account for write-back transactions, we extend our definitions of read/write miss rates as follows:

L2_COMB_RMR – combined L2 read miss rate – the L2 read miss rate that includes the write-back operations triggered by these misses:

$$L2_COMB_RMR = L2_RMR + L2_WBR_R$$

L2_COMB_WMR – combined L2 write miss rate – the L2 write miss rate that includes the write-back operations triggered by write misses:

$$L2_COMB_WMR = L2_WMR + L2_WBR_W.$$

On modern processors, not all L2 cache misses stall the processor. Usually there is a write buffer that absorbs the effects of write misses and the write-back operations triggered by write misses. A processor only blocks when the write buffer becomes full. To account for the write-buffer effect we derive an architecture-specific parameter *write stall coefficient (WSC).* WSC expresses the fraction of write misses that stall the processor for a given write miss rate. We derive this parameter empirically by measuring the fraction of write transactions that stall our processor for SPEC CPU 2000 integer workloads [10]. We found that for workloads with the combined L2 write miss rate of greater than 0.005, 90% of the writes stall the processor, while for workloads with a smaller miss rate, most of the writes are non-blocking. Accordingly, we set WSC as:

$$WSC = \begin{cases} 0.9, & L2_COMB_WMR > 0.005, \\ 0, & L2_COMB_WMR \le 0.005 \end{cases}$$

In our model we multiply the L2 write miss rate by WSC to account only for those write-miss transactions that stall the processor: those writes that do not stall the processor do not have the effect on the IPC, so we do not account for them. Using the WSC coefficient instead of modeling the full complexity of the write buffer is a reasonable simplification: for most workloads, the write buffer absorbs most writes, and, therefore, having a detailed write buffer model is not necessary. WSC needs to be derived for a specific architecture.

Expression for *L2_CPI* changes as follows:

$$L2_CPI = (L2_COMB_RMR + L2_COMB_WMR * WSC) * L2_MCOST \quad (4)$$

Substituting the *L2_CPI,* computed using equation (4), into equation (3) we compute the CPI, and then the IPC (equation (1)).

3.2 Multithreaded model

When the processor executes in multithreaded mode, one or more virtual processors may be blocked handling an L2 cache miss while others continue executing instructions (See Figure 2).

The entire processor is blocked only when all virtual processors are blocked. In Section 3.1 we showed how to model the time that a single virtual processor is blocked on L2 misses (*L2_CPI*). In this section we show how to use *L2_CPI* to estimate the time that the entire processor is blocked due to handling L2 misses.

We introduce an *individual blocking probability* – the probability that an individual virtual processor is blocked on an L2 cache miss. Then we combine individual blocking probabilities to estimate the time that the overall processor is blocked.

3.2.1 Individual blocking probability

We derive the individual blocking probability in *single-threaded* mode using *ideal_CPI* and *L2_CPI* (Section 3.1):

$$prob_blocked_ind = \frac{L2_CPI}{ideal_CPI + L2_CPI}$$

The individual blocking probability in multithreaded mode is expressed using the *individual* ideal CPI of the thread when the processor runs in multithreaded mode, *ideal_CPI_mt_ind*. *Ideal_CPI_mt_ind* is obtained using *ideal_CPI_mt* – the ideal CPI for the entire workload. For our architecture, the relationship between *ideal_CPI_mt_ind* and *ideal_CPI_mt* is determined by the number of virtual processors *V*. Recall that the processor switches threads on every cycle, so for each cycle that a thread is busy, it waits for the rest of the threads to take their turn using the processor:

$$ideal_CPI_mt_ind = ideal_CPI_mt * V \quad (5).$$

The individual blocking probability for the multithreaded mode is expressed as:

$$prob_blocked_mt_ind = \frac{L2_CPI}{ideal_CPI_mt_ind + L2_CPI} \quad (6),$$

3.2.2 Modeling multithreaded IPC

Let *V* be the number of virtual processors. An SMT processor can be in one of these *V+1* states:

(0) All *V* virtual processors are blocked,
(1) Exactly one virtual processor is busy – *(V-1)* are blocked,
(2) Exactly two virtual processors are busy,
…

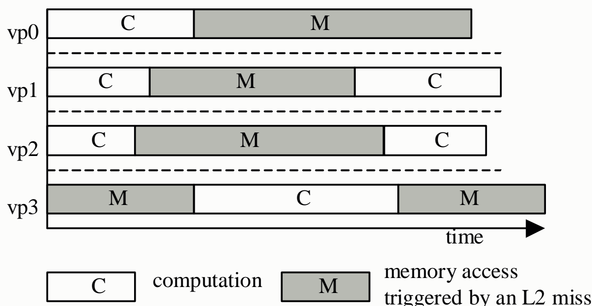

Figure 2. Memory-access latencies overlap in multithreaded processors.

(V) All virtual processors are busy.

In State *V*, the processor is running at IPC equal to *ideal_IPC_mt* (the inverse of *ideal_CPI_mt*). In State *0*, the processor is running at the IPC equal to zero. In the remaining states the processor is running at some IPC that is less than or equal to *ideal_IPC_mt* – we will refer to this quantity as *R_IPC(i)*, where *i* corresponds to the number of virtual processors that are *busy*. So, for example, if exactly three virtual processors are busy and one is blocked (*i=3*), the processor is running at *R_IPC(3)*. We show how we derive *R_IPC* in the next section.

We use the probability that a virtual processor is blocked to compute the corresponding probability that a virtual processor is busy (*prob_busy_mt_ind)*:

$$prob_busy_mt_ind = 1 - prob_blocked_mt_ind .$$

We compute probabilities *P(i)* that a processor is in state *i*, $(0 \le i \le V)$ as follows:

$$P(i) = \binom{V}{i} * prob_busy_mt_ind^{i} * prob_blocked_mt_ind^{V-i}$$

where *i* is the number of virtual processors that are busy in state *i*. We then compute the IPC in multithreaded mode (*IPC_mt*) by multiplying the IPC achieved in each state by the probability of that state and summing across all states:

$$IPC_mt = \sum_{i=0}^{N} P(i) * R_IPC(i) \quad (7).$$

3.2.3. Deriving R_IPC

The model for *R_IPC* is derived using linear regression analysis. The resulting equation expresses the dependence of *R_IPC(i)* on *ideal_IPC_mt* and *i.* We use SPEC CPU2000 integer benchmarks for derivation and arrive at an equation with a good fit (R-squared of 0.9):

$$R_IPC(i) = -0.69 + 0.2 * i + 0.94 * ideal_IPC_mt \quad (8).$$

Although we derived our equation offline, a recent study suggests that this could also be done online [31]. SPEC CPU2000 benchmark suite consists of a heterogeneous set of benchmarks, so we expect the equation derived using this benchmark suite to generalize well to integer workloads. A different equation needs to be derived for workloads that perform many floating-point operations.

3.3. Model evaluation

We evaluate our model using integer benchmarks in the SPEC CPU2000 benchmark suite [10]. We do not use *perlbmk*, because it is a multi-process benchmark, and we needed single-process benchmarks. For each benchmark, we estimate the overall IPC for multiple concurrent instances of the benchmark using our model. We obtain the inputs for our model by measurement. Also, for each benchmark we measure the actual IPC by running multiple concurrent instances of the benchmark on our simulator. We compare the estimated and measured IPC. We study the sensitivity of our model to variations in two of the input factors: the L2 cache miss rate and the degree of concurrency.

For the first analysis we alter the L2 cache size of the simulated machine from 48KB to 192KB: this creates variation in the L2 miss rate. Figure 3 shows the data. The difference between the measured and estimated IPC is 4% on average, with a median

	4 threads			3 threads			2 threads		
	Measured	Estimated	Difference	Measured	Estimated	Difference	Measured	Estimated	Difference
bzip	0.50	0.53	6.00%	0.64	0.67	4.69%	0.89	0.74	16.85%
crafty	0.46	0.47	2.17%	0.40	0.49	22.50%	0.34	0.41	20.59%
eon	0.42	0.40	4.76%	0.34	0.35	2.94%	0.26	0.27	3.85%
gap	0.50	0.54	8.00%	0.46	0.49	6.52%	0.43	0.46	6.98%
gcc	0.35	0.35	0.00%	0.31	0.38	22.58%	0.23	0.28	21.74%
gzip	0.82	0.72	12.20%	0.68	0.71	4.41%	0.52	0.55	5.77%
mcf	0.50	0.50	0.00%	0.18	0.20	11.11%	0.15	0.16	6.67%
parser	0.62	0.59	4.84%	0.49	0.56	14.29%	0.35	0.42	20.00%
twolf	0.45	0.47	4.44%	0.38	0.35	7.89%	0.32	0.29	9.38%
vortex	0.46	0.46	0.00%	0.37	0.43	16.22%	0.27	0.31	14.81%
vpr	0.52	0.54	3.85%	0.45	0.45	0.00%	0.35	0.35	0.00%
		Mean	**4.21%**		**Mean**	**10.29%**		**Mean**	**11.51%**
		Median	**4.44%**		**Median**	**7.89%**		**Median**	**9.38%**
		Maximum	**12.20%**		**Maximum**	**22.58%**		**Maximum**	**21.74%**

Table 2. Analysis of model sensitivity to the number of threads running in parallel. The data is obtained with the assumption of unlimited memory bandwidth.

difference of 3%, and a maximum of 12%. The estimates were less accurate for *gzip* (12% difference between measured and estimated IPC) because the *R_IPC* estimate was less accurate for *gzip* than for other benchmarks. *Gzip* is notably more compute-bound than the rest of the benchmarks, and the *R_IPC* equation did not suit it as well as the other benchmarks. The model for *R_IPC* could be made more accurate if parameterized by the workload's instruction mix. For the majority of the benchmarks, however, the simple *R_IPC* equation works well, as we expected.

Next we analyze the model's sensitivity to the number of concurrent threads. We set the cache size to 48KB and vary the number of concurrent benchmark instances: two, three, and four. The input *ideal_IPC_mt* for degrees of concurrency smaller than the maximum is estimated using equations derived in a similar fashion as the equation for *R_IPC* (equation 8).

Table 2 shows the data. The difference between the measured and estimated IPC is 4.21% (mean), 4.44% (median), and 12.20% (largest) for four concurrent benchmarks, 10.29% (mean), 7.89% (median), and 22.58% (largest) for three concurrent benchmarks, and 11.51% (mean), 9.38% (median), 21.74% (largest) for two concurrent benchmarks. The accuracy is comparable to that of the SMT models described in the past: the mean, median and maximum errors reported by Saavedra-Barrera [16] are 7%, 4% and 28% respectively.

Our model is less accurate when the degree of concurrency is smaller than the number of virtual processors, because in this case *ideal_IPC_mt* is estimated using a regression-derived linear equation. A precise model of *ideal_IPC_mt* would account for the architecture of the processor's shared resources and how these resources are used by a particular workload. Ongoing work on modeling SMT performance [7,12,31] suggests alternative avenues for approximating *ideal_IPC_mt* without sacrificing the model's simplicity.

A final enhancement to the model is accounting for the

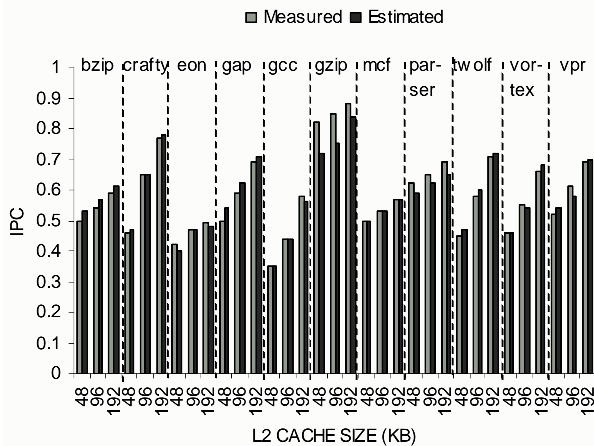

Figure 3. Comparison between measured and estimated IPC.

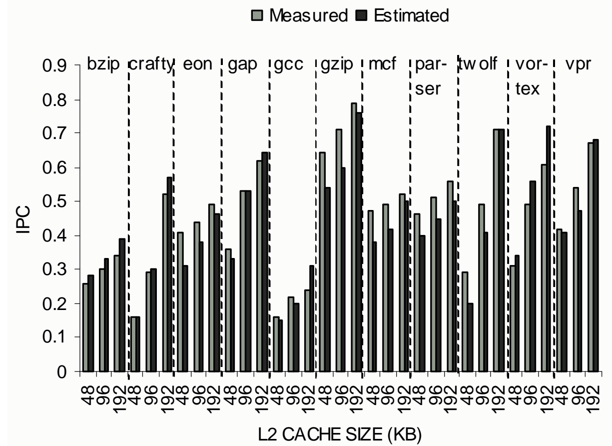

Figure 4. Measured vs. estimated IPC with limited memory bandwidth.

memory-bus delay. The existing models for memory-bus delay [26,27] are typically based on mean-value analysis of closed queuing networks [13,14,34]. We designed a model that is simpler to solve than the existing models. We omit presentation of our memory bus model due to space constraints and encourage the reader to refer to the manuscript providing the details [33]. We account for the memory-bus delay in our model by adding the estimated delay to *L2_MCOST* (equation (4)). Figure 4 shows how the IPC estimated using our model that accounts for the memory-bus delay compares with the IPC measured on a simulated machine configured to have limited memory bandwidth. The differences between the measured and estimated values are 11% (mean), 10% (median), and 31% (largest).

4. THE SCHEDULER

We implemented a user-level scheduler prototype that uses our model to determine the optimal number of concurrency to use on an SMT processor. The scheduler starts the multi-process or multithreaded job specified by the user, running a thread on each virtual processor, collects the machine statistics used to generate inputs for the model (*preparation phase)*, uses the model to estimate the IPC for each degree of concurrency smaller than the maximum, and finally forces the degree of concurrency corresponding to the highest predicted IPC by disabling one or more virtual processors (*optimization phase*). If the IPC obtained with the highest degree of concurrency (measured during the preparation phase) is higher than any of the estimated IPC for smaller degrees of concurrency the scheduler keeps all virtual processors busy.

4.1 Obtaining input parameters for the model

To estimate the IPC for a degree of concurrency N, where $N < V$, the scheduler must furnish the following inputs to the model: (1) the ideal IPC for N threads, (2) the L2 cache miss rate when N threads share the cache. All statistics necessary to generate the inputs are obtained using standard hardware counters.

4.1.1 Obtaining the ideal IPC

The scheduler first computes the ideal IPC for $N = V$, *ideal_IPC(V)*. Then it estimates *ideal_IPC(N)* using *ideal_IPC(V)* and a regression-derived linear equation. Recall the specification of our IPC model:

$$IPC = F(N, ideal_IPC(N), L2\ miss\ rate).$$

The actual IPC and the L2 miss rate for $N = V$ are obtained during the preparation phase from the hardware counters. *Ideal_IPC(V)* is then estimated by applying the inverse of the model:

$$ideal_IPC(V) = F'(V, IPC, L2\ miss\ rate)$$

Once *ideal_IPC(V)* is known, the ideal IPC for $N < V$ is estimated using the same process (described in Section 3.3) as we use to estimate *ideal_IPC_mt* for degrees of concurrency smaller than the maximum. Recall that we use linear equations derived in a similar fashion as the equation for *R_IPC* (equation (8)).

4.1.2 Obtaining the L2 miss rate

There are two known techniques to estimate the L2 miss rate when $N < V$ threads run in parallel. They are based on (1) the *stack-distance* model and (2) the *reuse-distance* model. The stack-distance model requires a *stack-distance profile* of the program. A stack-distance profile captures the temporal reuse behavior of an application; it can be obtained statically by the compiler [18], or at runtime if the machine has appropriate hardware counters [19]. Multiple stack-distance profiles are combined to estimate the miss rate for multiple threads running in parallel [20].

A *reuse-distance* model [32] requires a reuse-distance histogram, which, similarly to the stack-distance profile, captures the temporal reuse behavior of the program. Reuse-distance histogram can be obtained online with some performance overhead.

More work is needed to determine which is the better method for estimating the L2 cache miss rate by the scheduler. In our prototype we used the method based on the stack-distance model.

4.2 Performance results

Figure 5 shows the IPC achieved with our non-work-conserving (*NWC*) scheduler during the optimization phase and compares it to the IPC achieved with the default scheduler (*default*) and to the optimal IPC (*optimal*) determined via simulation. Each set of bars is labeled *<benchmark - L2 cache size>*, indicating that four instances of a *benchmark* run on a machine configured with *L2 cache size*.

On the left side of the graph are the cases where better performance can be achieved with non-work-conserving scheduling (CAN IMPROVE). These are the cases when the optimal IPC is greater than the IPC with the default scheduler. The NWC scheduler improves performance in each case, from 3% to 56%, often achieving the IPC as high as the optimal.

On the right side of the graph are the cases where no performance improvement can result from non-work-conserving scheduling (CANNOT IMPROVE): the IPC with the default scheduler is already as high as the optimal. In these cases the NWC scheduler correctly decides to keep all virtual processors busy, achieving the IPC equal to the optimal. Note that each memory-intensive benchmark that appears in the "CAN IMPROVE" section of the graph also appears in the "CANNOT IMPROVE" section with a larger cache size, where performance can no longer be improved by lowering the degree of concurrency.

These results demonstrate feasibility of implementing a non-work-conserving scheduler that uses an online analytical model to determine the optimal degree of concurrency on SMT processors. The key to obtaining these results is our simple model that works with inputs obtainable during compilation and at runtime.

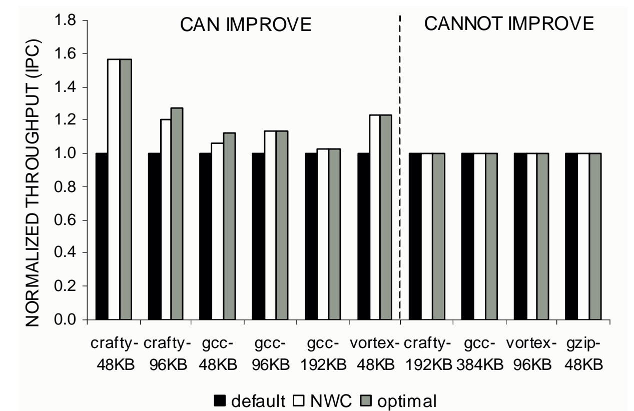

Figure 5. Normalized IPC with the default scheduler NWC scheduler, and the optimal.

5. RELATED WORK

Detailed IPC models for SMT processors proposed in the past [16,17] use Markov processes to model transitions between the busy and blocked states. These models are not trivial to solve; closed form solutions are approximated. Our model uses basic probability theory and is simpler, but no less accurate. Additionally, our model works with inputs that are obtainable at runtime or from a compiler.

Matick et al. described an IPC model for a single-threaded processor [11]. We use similar principles for our single-threaded IPC model.

Jung and et al. proposed a compiler technique to determine the optimal number of threads to execute a parallelizable loop on an SMT processor [12]. Their IPC model is derived using linear regression and is parameterized by the workload instruction mix. We would like to use the modeling techniques presented in this work to develop more precise models for *R_IPC* and *ideal_IPC_mt* (recall section 3.3).

Moseley et al. [31] presented IPC models for SMT processors that are used at runtime but are less accurate than ours. The authors model all sources of resource contention with simple models derived using linear regression and recursive partitioning. In contrast, we model the most significant source of contention using an accurate model, resorting to regression-derived models only for less important sources of contention. This could explain why we achieve higher accuracy. Although Moseley's models are less accurate and were not designed for a non-work conserving scheduler, their advantage is that they are designed to work across different architectures. We envision using the results of this work when applying our model to other architectures.

6. CONCLUSIONS

We presented a user-level prototype for a non-work-conserving scheduler for SMT processors. The scheduler uses a new analytical model designed specifically for the use inside an operating system scheduler. We succeeded in creating a simple model without sacrificing accuracy by precisely modeling the sources of resource contention that have a high impact on the IPC and approximating contention for other resources using simple methods.

We demonstrated that our scheduling prototype improves performance whenever possible, and does not make it worse when improvement cannot be achieved using the non-work-conserving policy. In the future we would like to apply our model to other SMT architectures and improve its accuracy when this can be done without sacrificing simplicity.

7. REFERENCES

[1] D. Tullsen, S. Eggers, H. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism, *ISCA'95.*

[2] R. Alverson et al., The Tera Computer System, *Proc. 1990 Intl. Conf. on Supercomputing*.

[3] A. Agarwal, B-H. Lim, D. Kranz, J. Kubiatowicz, APRIL: A Processor Architecture for Multiprocessing, *ISCA'90*

[4] J. Laudon, A. Gupta, M. Horowitz, Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations, *ASPLOS VI*, October 1994.

[5] Daniel Nussbaum, Alexandra Fedorova and Christopher Small, "An overview of the Sam CMT simulator kit", *Technical Report TR-2004-133*, Sun Microsystems Research Labs, March 2004.

[6] P. Kongetira. A 32-way Multithreaded SPARC(R) Processor. http://www.hotchips.org/archives/hc16/, *HOTCHIPS 16,* 2004.

[7] A. Snavely, D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Machine. In *Proc. of ASPLOS IX*, pp. 234-244, 2000.

[8] S. Hily, A. Seznec. Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading. *MTEAC'98*.

[9] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum. Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design, *USENIX 2005*

[10] SPEC CPU2000 Web site: http://www.spec.org

[11] R. E. Matick, T. J. Heller, and M. Ignatowski, Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory, *IBM Journal Of Research And Development, Vol. 45 No. 6,* November 2001.

[12] C. Jung, D. Lim, J. Lee, S. Han, Adaptive Execution Techniques for SMT Multiprocessor Architectures, in *Proc. Of PPoPP*, 2005.

[13] R. Onvural, Survey of Closed Queuing Networks With Blocking, *ACM Computing Surveys,* v. 22, issue 2, pp. 83-121, 1990

[14] L. Kleinrock, Queuing Systems Vol I, *Wiley*, 1975.

[15] P. J. Denning. 1968. Thrashing: Its Causes and Prevention. In *Proc. of AFIPS, 1968 Fall Joint Computer Conference*, vol. 33, pp. 915-922.

[16] R. Saavedra-Barrera, D. Culler and T. von Eicken, Analysis of Multithreaded Architectures for Parallel Computing, *SPAA 1990.*

[17] P. K. Dubey, A. Krishna and M. Squillante, Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures, *MASCOTS 1995.*

[18] C. Cascaval, L. DeRose, D.A. Padua, and D. Reed, Compile-Time Based Performance Prediction, *12th Intl. Workshop on Languages and Compilers for Parallel Computing,* 1999.

[19] G.E. Suh, S. Devadas, and L. Rudolph, A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning, In *Proc. Of Intl. High Performance Computer Architecture*, 2002.

[20] D. Chandra, F. Guo, S. Kim, and Y. Solihin, Predicting Inter-Thread Cache Contention on a Multi-Processor Architecture, In *Proc. Of 11th Int'l. Symposium on High-Performance Computer Architecture (HPCA-11),* 2005.

[21] E. Rosti, E. Smirni, G. Serazzi, L. Dowdy, Analysis of Non-Work-Conserving Processor Partitioning Policies, In *Proc. Of the Workshop on Job Scheduling Strategies for Parallel Processing,* 1995.

[22] M. Funk, Simultaneous Multi-threading (SMT) on eServer iSeries Power5™ Processors, *http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/pdf/SMT.pdf*

[23] Deborah T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture. *Intel Technical Journal*, pp. 4-15, February 2002.

[24] D. Patterson and K. Yelick, *Final Report 2002-2003 for MICRO Project #02-060*, University of California, Berkeley, CA 2003.

[25] E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, and K. C. Sevcik, Performance gains from leaving idle processors in multiprocessor systems, In *Proc. Of ICPP 1995*, Vol. III, pp. 203-210, 1995.

[26] Daniel J. Sorin, et al., Analytic Evaluation of Shared-memory Systems with ILP Processors. In *Proc. Of ISCA 1998*, pp. 380-391.

[27] D. Willick and D. Eager, An Analytical Model of Multistage Interconnection Networks, In *Proc. of 1990 ACM SIGMETRICS*, pp. 192-199.
[28] mpiBLAST: Open-Source Parallel BLAST *http://mpiblast.lanl.gov*
[29] D.A. Bader et al., BioSPLASH: A sample workload for bioinformatics and computational biology for optimizing next-generation performance computer systems, *Technical Report, University of New Mexico*, May 2005.
[30] Y. Chen et al., Compute-Intensive, Highly Parallel Applications and Uses, *Intel Technology Journal,* v.9(2), 2005.
[31] T. Moseley, J. Kihm, D. Connors and D. Grunwald, Methods for Modeling Resource Contention on Simultaneous Multithreading Processors, In *Proceedings of the 2005 International Conference on Computer Design (ICCD)*, 2005.
[32] E. Berg and E. Hagersten, StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis, *International Symposium on Performance Analysis of Systems And Software (ISPASS-2004)*, 2004.
[33] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. “Modeling the Effects of Memory Hierarchy Performance On Throughput of Multithreaded Processors”, *Technical Report TR-15-05, Harvard University*, 2005.
[34] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM*, 27(2):313-322, April 1980.
[35] K. Krewell, Cell Moves into the Limelight, *Microprocessor Report,* February 14, 2005.