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

Alexandra Fedorova, Margo Seltzer and Michael D. Smith

***Abstract* – Simultaneous multithreading (SMT) processors run multiple threads simultaneously on a single processing core. Because concurrent threads compete for the processor's shared resources, *non-work-conserving* scheduling, i.e., running fewer threads than the processor allows even if there are threads ready to run, can often improve performance. Nevertheless, conventional operating systems do not use non-work-conserving policies. We present a scheduling algorithm that applies the non-work-conserving policy when it improves performance. The scheduler uses an analytical performance model that, unlike existing models, is sufficiently simple and efficient for use inside the operating system. We built a user-level scheduler prototype demonstrating that our scheduler improves application performance in cases when a non-work-conserving policy is beneficial.**

Index terms – operating systems, processor scheduling, software performance, simultaneous multithreading

I. 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]. Conventional operating systems are *work-conserving*: they schedule a thread on every virtual processor, as long as there are runnable threads in the system. On SMT processors, work-conserving scheduling can result in *thrashing* [12] – a phenomenon where concurrent threads complete less work than they would if one or more virtual processors were left idle. Non-work-conserving scheduling [21,25] can eliminate thrashing by reducing contention for shared resources.

We present a new non-work-conserving scheduling algorithm and evaluate it using a user-level scheduler prototype. At the heart of our scheduler is a new analytical model that determines when it is beneficial to leave virtual processors idle. The model estimates the workload's instructions per cycle (IPC) for a given *degree of concurrency*, i.e., the number of concurrent threads. The scheduler uses this model to determine the degree of concurrency yielding the highest IPC, and then uses only that many virtual processors.

The analytical model is the key contribution of this work. Similar models proposed in the past [16,17] were designed for offline studies of SMT architectures. Their main objective was accuracy. Efficiency, simplicity, and the ability to obtain model inputs at runtime were less important.

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 it performs must impose little performance overhead. Therefore, our model has two critical requirements: a) 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); and b) inputs for the model must be obtained at runtime or at compile time.

Our model's simplicity derives from focusing on those resources for which contention is most likely to cause thrashing. We model these resources precisely, while representing other resources with simple, less precise, models. We identify contention for the L2 cache as the prime cause of performance degradation and thrashing [8,9,31] (also see Figures 1a and 1b). Contention for the L2 cache increases the miss rate, producing more main memory accesses. Memory access times are now 200-300 processor cycles and have been growing at the rate of 50% per year [24]. Therefore, L2 cache contention is likely to continue to cause thrashing for the foreseeable future. Our model precisely expresses the relationship between the L2 cache miss rate and IPC; we approximate contention for other resources using linear regression, significantly simplifying our model.

The challenge in estimating the effect of the L2 cache miss rate on the IPC of an SMT processor is to determine how much of the L2 cache miss latency can be "hidden" by hardware multithreading: the processor overlaps memory access of one thread with computation of another. We use basic probability theory to model this effect simply. Models described in the past used Markov chains and were more complex [16,17].

We assume the SMT architecture of Sun Microsystems' UltraSPARC® T1 [6] processor, 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 [10] to actual measurements obtained using an execution-driven simulator of the 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 for more complex off-line models [16,17].

We implement a user-level prototype of our scheduling algorithm, and demonstrate its ability to predict when it is beneficial to use the non-work-conserving policy. We show

Alexandra Fedorova is at Harvard University, Cambridge, MA 02138 and Sun Microsystems, Burlington, MA, 01803, USA (e-mail: fedorova@eecs.harvard.edu).

Margo Seltzer is with Harvard University, Cambridge, MA 02138, USA (e-mail: margo@eecs.harvard.edu).

Michael D. Smith is with Harvard University, Cambridge, MA 02138, USA (e-mail: smith@eecs.harvard.edu)

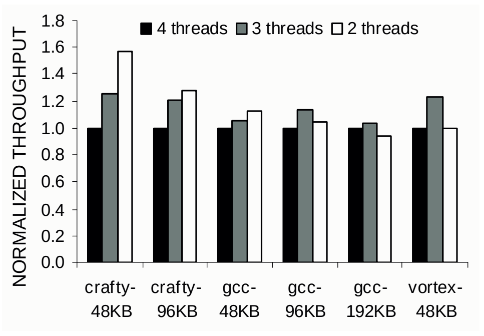

Fig. 1a. Normalized throughput for memory-intensive SPEC CPU2000 benchmarks, on a simulated four-way SMT processor. We use multiple threads to run each benchmark, such that each thread runs its own benchmark instance. There is a set of bars for each benchmark/L2 cache size, and each bar corresponds to the number of concurrent threads used to run the benchmark. 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.

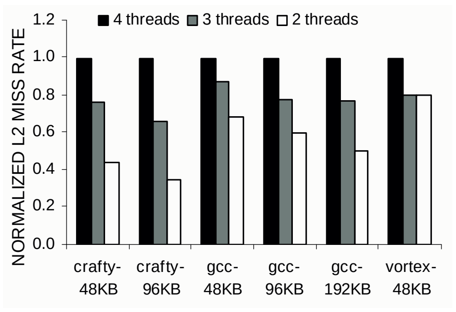

Fig. 1b. Normalized L2 miss rate corresponding to the experiments in Figure 1a.

that applications using our scheduler achieve throughput improvement of 3 to 56% when the non-work-conserving policy is beneficial 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 II, we describe our target SMT architecture and the simulator used for our experiments. In Section III, we present the model and evaluate its accuracy. In Section IV, we present the scheduler prototype and performance results. In Section V, we discuss related work, and in Section VI, we conclude.

II. THE SMT ARCHITECTURE

Modern SMT processors switch threads when 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 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 the 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 multicore architectures.

TABLE 1. SIMULATOR CONFIGURATION.

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.
Memory bandwidth	5.2 GB/s

We run our experiments on an execution-driven full-system simulator [5] of the UltraSPARC T1. Using a simulator allowed us to validate our model for various hardware configurations. Table 1 summarizes the architectural characteristics of the simulated processor. The configuration differs from that of the actual UltraSPARC T1 in the number of cores (we simulate a single core, while the T1 has six or eight), the memory bandwidth, and the clock speed (we could not model the precise clock speed of the T1 due to simulator constraints). We use a smaller L2 cache size than the size typical to modern two- or four-way SMT processors [23]: this allows us to evaluate our model under high L2 cache contention, which we expect on next-generation chip multiprocessors [6,35], where eight threads might be sharing a 512KB cache. Our full-system simulator runs the Solaris™ operating system and all applications for Solaris/SPARC® platforms unmodified.

III. THE MODEL

Our model expresses the relationship between the L2 cache miss rate and the processor's IPC. The IPC is determined by two components: 1) how many cycles the processor is busy and 2) how many cycles the processor stalls handling L2 cache misses. We refer to the first component as *perfect-cache cycles per instruction (CPI)* – the CPI achieved when there are no L2 cache misses. The second component depends on the L2 cache miss rate. We express the IPC of a workload on an SMT processor as the function of its perfect-cache CPI, the L2 cache miss rate, and the number of concurrent threads:

$$IPC = F(N, L2\_MR(N), perf\_cache\_CPI(N)) \qquad (1),$$

where N is the number of concurrent threads, *perf_cache_CPI(N)* is the perfect-cache CPI, and *L2_MR(N)* is the L2 cache miss rate achieved by the N threads.

Our goal is to precisely model the effects of L2 cache contention on IPC. In this section, we describe this model, assuming that *perf_cache_CPI(N)* and *L2_MR(N)* are measured directly. In Section IV, we explain how we derive the values for *perf_cache_CPI(N)* and *L2_MR(N)* at runtime using a combination of simple models and measurements obtained from hardware performance counters.

Our model relies on several architectural parameters that are statically configured for a given system:

- the number of virtual processors,
- DRAM latency,

– L2 cache size,
– memory bus bandwidth.

We begin by describing the performance model for a single-threaded workload (Section III.A). Then we extend our model for multithreaded workloads (Section III.B).

A. Single-threaded model

As a building block for our multithreaded model, we introduce how we model the CPI when a processor runs a single thread. Our single-threaded CPI model is an extension of models proposed in the past [11,15,38]. First, we introduce some definitions:

L2_CPI – for a given L2 miss rate, the number of cycles per instruction that a thread stalls handling L2 cache misses.
L2_RMR – L2 read miss rate – the number of L2 read misses per instruction, including data and instruction misses.
L2_WMR – the number of L2 write misses per instruction.
L2_MCOST – the cost, in cycles, of handling an L2 cache miss, including the memory latency and the memory-bus delay. While the memory latency is fixed, the memory-bus delay depends on contention for the bus. For now we assume that memory-bus delay is zero. We relax this assumption at the end of Section III.

The CPI is comprised of *perf_cache_CPI* and *L2_CPI*:

$$CPI = perf\_cache\_CPI + L2\_CPI \qquad (2)$$

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

$$L2\_CPI = (L2\_RMR + L2\_WMR) * L2\_MCOST \qquad (3)$$

In addition to cache misses, we must account for *write-back transactions*, because they contribute to L2 stall cycles. Processors with write-back caches issue write-back transactions to write dirty cache lines back to memory. A write-back may be triggered on a cache read miss or on a write miss. We define the respective write-back rates as:

L2_WBR_R – the write-backs per instruction triggered by read misses.

L2_WBR_W – the write-backs per instruction triggered by write misses.

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

L2_COMB_RMR – the L2 read miss rate including the write-backs triggered by read misses:

$$L2\_COMB\_RMR = L2\_RMR + L2\_WBR\_R$$

L2_COMB_WMR – the L2 write miss rate including the write-backs triggered by write misses:

$$L2\_COMB\_WMR = L2\_WMR + L2\_WBR\_W.$$

We found that the fraction of write-backs triggered by each type of cache miss can be accurately approximated by the fraction of each type of cache miss. For example, if 60% of the cache misses are read misses, then roughly 60% of the write-backs are triggered by read misses.

Modern processors are equipped with *write buffers* that absorb the effect of writes: a writing thread places a store value into the buffer and proceeds without blocking. The thread stalls only when the write buffer becomes full. Therefore, write cache misses and the corresponding write-backs do not always stall the processor. We model the write-buffer effect using an empirically derived *write stall coefficient (WSC)*. *WSC* expresses the fraction of write misses that stall the processor. We derive this coefficient for a given architecture. The coefficient is also application-specific: the value of the coefficient depends on the workload's write miss rate. Through experimentation with SPEC CPU2000 integer benchmarks, 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 compute the workload-specific value for the *WSC* using the following step function:

$$WSC = \begin{cases} 0.9, & L2\_COMB\_WMR > 0.005, \\ 0, & L2\_COMB\_WMR \le 0.005 \end{cases}$$

To account for the write-buffer effect in our model, we multiply the L2 write miss rate by *WSC*, thus accounting only for those write-miss cycles that stall the processor:

$$L2\_CPI = (L2\_COMB\_RMR + L2\_COMB\_WMR * WSC) * L2\_MCOST \qquad (4)$$

Using the *WSC* coefficient instead of precisely modeling the complexity of the write buffer is a reasonable simplification: for most workloads, the write buffer fully absorbs the writes, so having a detailed write buffer model is not necessary.

B. Multithreaded model

When the processor executes a multithreaded workload, one or more virtual processors stall handling L2 cache misses while others continue executing instructions. The entire processor stalls only when all virtual processors stall. In Section III.A, we showed how to compute the *L2_CPI* for a single threaded execution. In this section we show how to use the *L2_CPI* to estimate the aggregate IPC achieved by a multithreaded workload for a given L2 cache miss rate.

We first introduce a *stall probability* – the probability that an individual virtual processor stalls on an L2 cache miss. Then we combine individual stall probabilities to estimate the processor IPC.

1) Stall probability

Our goal is to model the effect of the L2 cache miss rate on the IPC of an SMT processor. A stall probability captures the effect of the L2 cache miss rate on an individual virtual processor. By combining individual stall probabilities, we estimate the effect of the L2 cache miss rate on the entire processor.

If the processor is running a single thread, the probability that an individual virtual processor stalls (*prob_stall)* is trivially computed using *perf_cache_CPI* and *L2_CPI* for that thread:

$$prob\_stall = \frac{L2\_CPI}{perf\_cache\_CPI + L2\_CPI}.$$

When the processor runs multiple threads on its *V* virtual processors, the meaning of perfect-cache CPI changes, and we must adjust our formula to take this into account. The perfect-cache CPI for the multithreaded workload is the aggregate CPI achieved by *all* the threads when they experience the perfect cache hit rate: *perf_cache_CPI(V)*. (We can estimate this quantity at runtime, as described in Section IV). Our formula for *prob_stall* requires the perfect-cache CPI as perceived by an *individual* virtual processor. We refer to this quantity as *perf_cache_CPI_mt_ind*, and compute it as follows:

$$perf\_cache\_CPI\_mt\_ind = perf\_cache\_CPI(V) * V \quad (5).$$

Recall that our 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. Hence we multiply *perf_cache_CPI(V)* by *V* in order to compute the individual perfect-cache CPI.

A stall probability for an individual virtual processor during a multithreaded execution (*prob_stall_mt*) is:

$$prob\_stall\_mt = \frac{L2\_CPI}{perf\_cache\_CPI\_mt\_ind + L2\_CPI} \quad (6),$$

2) Modeling multithreaded IPC

We now combine individual stall probabilities to estimate the aggregate IPC. Let *V* be the number of virtual processors. An SMT processor can be in one of the following *V+1* states:

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

In State *V*, the processor's CPI equals to *perf_cache_CPI(V).* We say that it runs at *perf_cache_IPC(V)* – the inverse of perfect-cache CPI. In State *0*, the entire processor is stalled, so it runs at the IPC equal to zero. In the remaining states, the processor achieves an IPC less than or equal to *perf_cache_IPC(V)* – we refer to this quantity as *R_IPC(N)*, where *N* corresponds to the number of virtual processors that are *busy.* So, for example, if exactly three virtual processors are busy (*N=3*), the processor is running at *R_IPC(3)*. A probability that a virtual processor is busy, *prob_busy_mt,* is:

$$prob\_busy\_mt = 1 - prob\_stall\_mt$$

We compute probabilities *P(N)* that a processor is in state *N*, *(0 ≤ N ≤ V)* as follows:

$$P(N) = \binom{V}{N} * prob\_busy\_mt^{N} * prob\_stall\_mt^{V-N},$$

where *N* is the number of virtual processors that are busy in state *N*. We then compute the overall processor IPC for the multithreaded execution by multiplying the IPC achieved in each state by the probability of that state and summing across all states:

$$IPC = \sum_{N=0}^{V} P(i) * R\_IPC(N) \quad (7).$$

3) Deriving R_IPC

R_IPC(N) is the IPC achieved by *N* concurrent threads on an SMT processor with *V* virtual processors, when *N* < *V*. We estimate it using a linear model, whose general form is:

$$R\_IPC(N) = F(N, perf\_cache\_IPC(V)) \quad (8),$$

where *N* is the number of concurrent threads, and *perf_cache_IPC(V)* is the perfect-cache IPC achieved by *V* threads. (In Section IV, we explain how we obtain the input value for *perf_cache_IPC(V).*) We derive the equation for *R_IPC* using linear regression applied to the data on *R_IPC* and *perf_cache_IPC* collected from simulation of SPEC CPU2000 benchmarks with a large L2 cache. A recent study suggests that this equation could be derived online using real hardware [31] – this is a more practical approach and we are interested in using it in the future. The equation we derived using the simulation data has a good fit (0.9 R-squared). The SPEC CPU2000 suite consists of benchmarks with a variety of cache access patterns [36], and we expect the resulting equation to generalize well to integer workloads.

C. Model evaluation

We designed our model so that a non-work-conserving scheduler could improve performance by determining the degree of concurrency producing the highest IPC. The model's accuracy determines the effectiveness of performance optimizations. In this section, we evaluate our model's accuracy using the integer benchmarks in the SPEC CPU2000 suite. We derive the *WSC* coefficient and the *R_IPC* equation for our model using a subset of benchmarks, i.e., the *training set.* The training set consists of *crafty, gcc, gzip, parser, vortex* and *vpr*. We report evaluation results for the entire integer suite, except *perlbmk*: this is a multi-process benchmark, and because of how we built our scheduler prototype we needed single-process benchmarks.

For each benchmark, we use our model to estimate the aggregate IPC achieved by multiple concurrent threads, each running its own instance of this benchmark. We obtain the inputs for our model by measurement. We compare the estimated IPC with the actual, which we measure by simulating the benchmark. 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 set the number of concurrent threads to four and alter the L2 cache size of the simulated machine from 48KB to 192KB. This creates variation in the L2 miss rate. Figure 2 shows the results. The difference between the measured and estimated IPC is 4% on average, with a median difference of 3%, and a maximum difference of 12%. The estimates were less accurate for *gzip* (12% discrepency) 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

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.

	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%**

R_IPC equation did not suit it as well as the others. 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 threads executing the benchmark from two to four. The input *perf_cache_IPC(N)* for $N < V$ is estimated using the equation derived in a similar fashion as the equation for *R_IPC*, as explained in Section IV.

Table 2 shows the data. The difference between the measured and estimated IPC is 4.21% (mean), 4.44% (median), and 12.20% (maximum) for four concurrent threads, 10.29% (mean), 7.89% (median), and 22.58% (largest) for three concurrent threads, and 11.51% (mean), 9.38% (median), 21.74% (largest) for two concurrent threads. 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, *perf_cache_IPC(N)* is estimated using a regression-derived linear equation. A precise model of *perf_cache_IPC(N)* would account for architectural details of the processor and the characteristics of the specific workload. Ongoing work on modeling SMT performance [7,12,31] suggests alternative avenues for approximating *perf_cache_IPC(N)* without sacrificing the model's simplicity.

A final enhancement to the model is accounting for the 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 than the existing models. We omit presentation of our memory bus model due to space constraints and encourage the reader to refer to the full manuscript for the

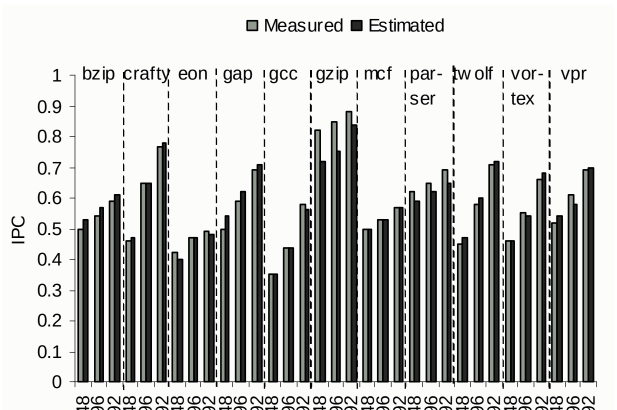

Fig. 2. Comparison between measured and estimated IPC.

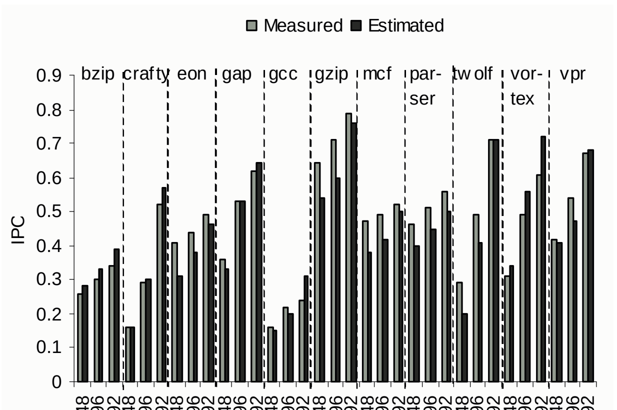

Fig. 3. Measured vs. estimated IPC with limited memory bandwidth.

details [33]. We account for the memory-bus delay in our model by adding the estimated delay to L2_MCOST (4). Figure 3 shows how the IPC estimated using our model, enhanced to account for the memory-bus delay, compares with the IPC measured on a simulated machine configured with memory bandwidth of 5.2GB/s. The differences between the measured and estimated values are 11% (mean), 10% (median), and 31% (largest).

IV. THE SCHEDULER

We designed our model for use by a non-work-conserving scheduler. The scheduler uses our model to determine the degree of concurrency that yields the highest IPC on an SMT processor. To demonstrate the efficacy of our model, we built a user-level scheduler prototype. We now describe how the scheduler uses our model to make scheduling decisions and how it obtains input parameters for the model at runtime.

We configure the scheduler with architecture-specific parameters (listed in the beginning Section III) and the equation for *R_IPC* (Section III.B.3). The scheduler operates in two phases: the *preparation* phase and the *optimization* phase:

Preparation phase: The purpose of this phase is to prepare inputs for our IPC model. The scheduler runs a multi-process or a multithreaded job using all of the *V* virtual contexts. It collects measurements using hardware performance counters and generates the inputs for the model. (We explain how we generate inputs in Sections IV.A and IV.B.) This phase lasts until each thread retires roughly 100 million instructions – we empirically determined that for our benchmarks this length is sufficient to capture long-term cache-locality properties.

Optimization phase: The purpose of this phase is to determine the best degree of concurrency and enforce it. For each degree of concurrency $N < V$, the scheduler estimates the IPC using our model. It then forces the degree of concurrency corresponding to the highest predicted IPC by disabling one or more virtual processors. If the IPC obtained with the highest degree of concurrency (measured during the preparation phase) is higher than any of the estimated IPC for $N < V$, the scheduler keeps all virtual processors busy.

The optimization phase can last as long as the workload does not change its cache miss locality properties. If a workload enters a new locality phase, the optimization phase must be stopped, and the preparation phase must be repeated. We analyzed temporal locality behavior of our benchmarks, and found that most benchmarks have long locality phases and the changes between the phases are gradual. For such benchmarks, it is sufficient to repeat scheduling phases at fixed intervals, such as one billion instructions. For benchmarks with frequently changing access patterns, such as *gcc*, changes in locality phases must be detected dynamically, and the scheduling phases repeated when a change is detected. Previous research has shown how to perform dynamic locality phase prediction on conventional processors [37]. Similar ideas could be used on SMT processors. Our current prototype runs the optimization phase for one billion instructions and then repeats the preparation phase.

A. Obtaining input parameters for the model

To estimate the IPC for a degree of concurrency *N*, where $N < V$, the scheduler provides the following inputs to the model: (1) *perf_cache_IPC(N)* – the perfect-cache IPC for *N* threads, and (2) *L2_MR(N)* – the L2 cache miss rate achieved by *N* concurrent threads. The inputs are obtained using hardware performance counters, compiler-generated input, and simple models, as we describe next.

1) Obtaining the perfect-cache IPC

The perfect-cache IPC achieved by multiple homogeneous threads depends on two factors: how efficiently these threads use the processor, and how many threads are actually running. Therefore, the scheduler computes *perf_cache_IPC(N)* using a two-step process: (1) compute the perfect-cache IPC for the maximum degree of concurrency, *perf_cache_IPC(V)*. This tells us how efficiently the threads use the processor; (2) For a given *N*, estimate *perf_cache_IPC(N)*, using a regression-derived linear equation. We describe these steps in more detail:

(1) We compute *perf_cache_IPC(V)* by applying the inverse of our model. Recall the general form for our model:

$$IPC = F(N, perf\_cache\_CPI(N), L2\_MR(N)).$$

We first compute *perf_cache_CPI(V)* by applying the inverse of the model:

$$perf\_cache\_CPI(V) = F'(V, IPC, L2\_MR(V)).$$

(The actual IPC and the L2 miss rate *L2_MR(V)* are obtained during the preparation phase from the hardware counters.) We then compute *perf_cache_IPC(V)* by taking the inverse of *perf_cache_CPI(V).*

(2) We estimate *perf_cache_IPC(N)* using a regression-derived linear equation, where *N* and *perf_cache_IPC(V)* are supplied as inputs. The equation has the same form and is derived in a similar way as (8) (the equation for *R_IPC*). This equation is also supplied to the scheduler upon initialization.

We evaluated the accuracy of the estimated *perf_cache_IPC(N)* and found that the difference between the actual and the estimated *perf_cache_IPC(N)* is 7.14% (average), 6.17% (median), and 22.92% (max). Furthermore, we found that if separate equations are derived for workloads with high cache miss rates and medium-to-low cache miss rates, the accuracy significantly improves: the difference between the actual and the estimated *perf_cache_IPC(N)* is 2.7% (average), 1.55% (median), and 6.87% (max).

2) Obtaining the L2 miss rate

There are two known techniques to estimate *L2_MR(N)*, the L2 miss rate when $N < V$ threads run in parallel: (1) the *stack-distance* model and (2) the *reuse-distance* model. The stack-distance model requires a *stack-distance profile* of the running program – a short summary of the program's memory access patterns that captures its temporal reuse behavior. The stack-distance profile can be obtained statically using a compiler [18] or at runtime if the machine has appropriate hardware counters [19]. Multiple stack-distance

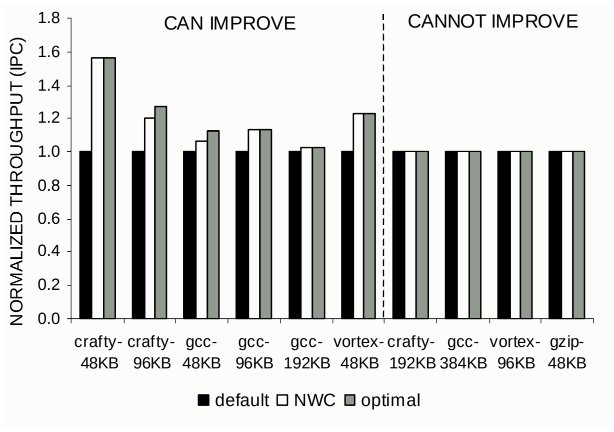

Fig. 4. IPC with the default scheduler, the NWC scheduler, and the optimal.

profiles are combined to accurately estimate miss rates for *N* concurrent threads [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 running program. Reuse-distance histogram can be obtained online, but this process generates performance overhead.

More work is needed to determine which is the better method for online estimation of *L2_MR(N)*. The stack-distance model is said to have poor accuracy for workloads with frequently changing execution patterns [18]. Additionally, it relies on compiler-generated stack distance profiles. A reuse-distance model is more flexible, but generates performance overhead, sometimes as much as 40% [32]. Although accuracy can be sacrificed for performance with the reuse-distance model, more work is needed to determine whether an acceptable trade-off can be achieved.

In our prototype, we estimated *L2_MR(N)* using a method based on the stack-distance model. Although stack-distance profiles are typically furnished by a compiler, we instrumented our simulator to build such profiles. We generated profiles for all benchmarks in advance, and then supplied them to the scheduler.

B. Performance results

In this section, we demonstrate how our non-work-conserving (NWC) scheduler improves application performance by determining the best degree of concurrency. We chose several memory-bound benchmarks (such benchmarks are likely to benefit from the non-work-conserving policy) and one compute-bound benchmark. Our experiment consists of running four concurrent instances of a benchmark on our processor with four virtual contexts and a given L2 cache size. We use several cache sizes. We use reference input sets. We fast-forward the simulation until all the threads have reached their main processing loop, and then perform detailed simulation for 100 million instructions.

The NWC scheduler performs the preparation phase during this execution segment. At the end of the preparation phase, the scheduler uses our model to determine the degree of concurrency yielding the highest IPC. To determine performance with the NWC scheduler, we measure the IPC achieved with the degree of concurrency selected as the best over a period of 100 million instructions.

To determine the performance with the default scheduler, we measure the IPC achieved using the maximum degree of concurrency. We determine the optimal IPC by simulating each benchmark using all possible degrees of concurrency, and selecting the highest IPC.

Figure 4 shows the results. There are three sets of bars for each experiment, showing the IPC achieved with the default scheduler (*default)*, the non-work-conserving scheduler (*NWC)*, and the optimal IPC *(optimal).* On the left side of the graph, we show the cases where better performance can be achieved with non-work-conserving scheduling (CAN IMPROVE). 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, we show 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.

Our results demonstrate feasibility of implementing a non-work-conserving scheduler that uses an online model to determine the best degree of concurrency. Although we have not evaluated the performance overhead generated by the scheduler, we believe it can be made small: most computations involve simple integer arithmetic and reading values for hardware counters, which typically have overheads of only a few cycles. The key to obtaining these results is our simple model that works with inputs obtainable during compilation and at runtime.

V. RELATED WORK

IPC models for SMT processors proposed in the past [16,17] use Markov processes to model an SMT processor’s transitions between busy and stalled states. These models are not trivial to solve; closed form solutions are approximated. Our model uses basic probability theory and is simpler, but not less accurate. Additionally, our model works with inputs that are obtainable at runtime or from a compiler.

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 techniques presented in this work to develop more precise models for *R_IPC* and perfect-cache IPC for degrees of concurrency smaller than the maximum.

Moseley et al. [31] presented IPC models for SMT processors that are also suitable for online use but are less accurate than ours. They 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. Although Moseley's models are less accurate, they work across different architectures. We envision using the results of this work to make our model cross-architectural.

VI. CONCLUSIONS

We presented a non-work-conserving scheduler for SMT processors and evaluated it using a user-level prototype. Our scheduler determines when using a maximum degree of concurrency on an SMT processor produces thrashing and improves performance by lowering the degree of concurrency. Scheduling decisions are made using a new online SMT performance model, which is simple, but as accurate as existing off-line models. We achieved simplicity by precisely modeling contention for the L2 cache, which has a high impact on the IPC, and approximating contention for other resources using simple models. To further enhance fidelity of our scheduler, we plan to work on improving online models for cache miss rates on SMT processors. We also plan to develop methods for online derivation of equations for perfect-cache IPC. In addition to addressing these challenges, we would like to apply our model to other SMT architectures and evaluate if using heterogeneous workloads.

VII. REFERENCES

[1] D. Tullsen, S. Eggers, H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", in *Proc. 1995 Intl. Symposium on Computer Architecture,* pp. 533-544.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portenfield, B. Smith, "The Tera Computer System", in *Proc. 1990 Intl. Conf. on Supercomputing,* pp. 1-6.

[3] A. Agarwal, B-H. Lim, D. Kranz, J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing", in *Proc. 1990 Intl. Symposium on Computer Architecture*, pp. 104-114.

[4] J. Laudon, A. Gupta, M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations", in *Proc. 6th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems*, pp. 308-318, 1994.

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

[6] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor", in *Proc IEEE Micro*, vol. 25, no. 2, pp. 21-29, Mar/Apr, 2005.

[7] A. Snavely, D. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Machine", in *Proc. 9th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems*, pp. 234-244, 2000.

[8] S. Hily, A. Seznec, "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading", in *Proc. of the Workshop on Multithreaded Execution Architecture and Compilation,* January 1998.

[9] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum "Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design", in *Proc. USENIX Annual Technical Conf.,* pp. 395-398, 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. 10th Symposium on Principles and Practice of Parallel Programming,* pp. 236-246, 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] H. Wasserman, O. Lubeck, Y. Luo, F. Bassetti, "Performance Evaluation of the SGI Origin2000: A Memory-Centric Evaluation of LANL ASCI Applications", in Proc. 1997 ACM/IEEE Conf. On Supercomputing, pp. 1-11.

[16] R. Saavedra-Barrera, D. Culler and T. von Eicken, "Analysis of Multithreaded Architectures for Parallel Computing", in Proc. 2nd Annual Symposium on Parallel Algorithms and Architectures, 1999.

[17] P. K. Dubey, A. Krishna and M. Squillante, "Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures", in Proc. 3rd Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 110-122, 1995.

[18] C. Cascaval, L. DeRose, D.A. Padua, and D. Reed, "Compile-Time Based Performance Prediction", in *Proc. 12th Intl. Workshop on Languages and Compilers for Parallel Computing,* pp. 365-379, 1999.

[19] G.E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning", in *Proc. 8th Intl. Symposium on High Performance Computer Architecture*, pp. 117-128, 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*, pp. 340-351, 2005.

[21] E. Rosti, E. Smirni, G. Serazzi, L. Dowdy, "Analysis of Non-Work-Conserving Processor Partitioning Policies", in *Proc. Workshop on Job Scheduling Strategies for Parallel Processing,* pp. 165-181, 1995.

[22] M. Funk, SMT on eServer iSeries Power5™ Processors, 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, University of California, Berkeley, CA, Final Report 2002-2003 for MICRO Project #02-060, 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. 1995 Intl. Conf. on Parallel Processing,* Vol. III, pp. 203-210.

[26] Daniel J. Sorin, V. Pai, S. Adve, M. Vernon, D. Wood, "Analytic Evaluation of Shared-memory Systems with ILP Processors", in *Proc. 1998 International Symposium on Computer Architecture*, 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, *mpiblast.lanl.gov*

[29] D.A. Bader V. Sachdeva, V. Agarwal, G. Goel, A.N. Singh, "BioSPLASH: A Sample Workload For Bioinformatics And Computational Biology For Optimizing Next-Generation Performance Computer Systems", University of New Mexico, Tech. Rep., 2005.

[30] Y. Chen, Q. Diao, C. Dulong, C. Lai, W. Hu, E. Li, T. Wang, Y. Zhang, "Performance Scalability of Data Mining Workloads in Bioinformatics", *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 *Proc. 2005 International Conf. on Computer Design,* pp. 373-380.

[32] E. Berg, E. Hagersten, "StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis", in *Proc. 2004 Intl. Symposium on Performance Analysis of Systems and Software*, pp. 20-27.

[33] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. "Modeling the Effects of Memory Hierarchy Performance On Throughput of Multithreaded Processors", Harvard University, Cambridge, MA, Tech. Rep. TR-15-05, 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.

[36] Benjamin Lee, "An Architectural Assessment of SPEC CPU Benchmark Relevance", Harvard University, Cambridge, MA, Tech. Rep. TR-02-06, 2006.

[37] X. Shen, Y. Zhong and C. Ding, "Locality Phase Prediction", in. *Proc. 11th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems,* pp. 165-176, 2004.

[38] Y. Solihin, V. Lam, and J. Torrellas, "Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors", in *Proc. 1999 ACM/IEEE Conf. on Supercomputing,* pp. 17-30.