Architectural Support for Enhanced SMT Job Scheduling

Alex Settle Joshua Kihm Andrew Janiszewski
Dan Connors
University of Colorado at Boulder
Department of Electrical and Computer Engineering 425 UCB, Boulder, Colorado
{settle,kihm,janiszew,dconnors}@colorado.edu

Abstract

By converting thread-level parallelism to instruction level parallelism, Simultaneous Multithreaded (SMT) processors are emerging as effective ways to utilize the resources of modern superscalar architectures. However, the full potential of SMT has not yet been reached as most modern operating systems use existing single-thread or multiprocessor algorithms to schedule threads, neglecting contention for resources between threads. To date, even the best SMT scheduling algorithms simply try to group threads for co-residency based on each thread's expected resource utilization but do not take into account variance in thread behavior. As such, we introduce architectural support that enables new thread scheduling algorithms to group threads for co-residency based on fine-grain memory system activity information. The proposed memory monitoring framework centers on the concept of a cache activity vector, which exposes runtime cache resource information to the operating system to improve job scheduling. Using this scheduling technique, we experimentally evaluate the overall performance improvement of workloads on an SMT machine compared against the most recent Linux job scheduler. This work is first motivated with experiments in a simulated environment, then validated on a Hyperthreading-enabled Intel Pentium-4 Xeon microprocessor running a modified version of the latest Linux Kernel.

1. Introduction

Simultaneous Multithreading (SMT) [9, 22, 23, 3] has emerged as a leading architecture model to achieve high performance in scientific and commercial workloads. By improving the utilization of architecture resources, SMT architectures maximize on-chip parallelism by converting thread-level parallelism (TLP) to instruction-level parallelism (ILP) [14]. During periods when individual threads might otherwise stall and disable the efficiency of a large devoted single-threaded processor, SMT improves execution throughput by maintaining on-chip parallelism. Although an SMT machine can compensate for periods of low ILP due to cache miss delays by exploiting TLP between applications, there are severe limitations to scaling future SMT designs by simply maximizing the number of active threads. Likewise, current operating system scheduling techniques and traditional hardware approaches cannot sufficiently improve the interaction of threads in the presence of the widening gap between processor and memory latencies. For instance, although current SMT techniques such as Intel's Hyperthreading [6] have shown improvements in throughput, studies [4] have shown that contention for cache resources in SMT designs can have equally negative effects on application performance.

To date most SMT thread management strategies have been static and software-oriented, greatly limiting the expected performance and application space of multithreaded architectures. Three general domains of thread optimization have been proposed: operating system thread scheduling [19, 15], dedicated architecture techniques [21, 1], and cooperative compilation techniques [10]. These techniques have largely been directed by using prior knowledge of program execution behavior. For instance, thread symbiosis [19], which is the characteristic that threads may excel together in a simultaneous multithreading environment, has largely been evaluated using informed scheduling techniques based on prior knowledge of thread interaction. Even with profile-based scheduling decisions, such approaches have a limited ability to adapt to the wide variations in engineering and commercial workloads and the corresponding impact on modern memory systems.

In order to increase the cache system effectiveness for simultaneously multithreaded architectures, we investigate methods of *run-time guided thread management,* in which thread 'symbiosis' is accurately estimated by exposing novel architecture performance monitoring features to the operating system job scheduler.

Our scheme seeks to aid thread scheduling by gathering run-time cache use and miss patterns for each active hardware thread. We introduce the concept of an *activity vector*, which is a collection of event counters for a set of contiguous cache blocks. In addition, this paper presents techniques for predicting thread 'symbiosis' used for co-scheduling applications. Overall, we show that the use of activity vectors for thread scheduling in real simultaneously multithreaded systems improves the performance by roughly 5% over current scheduling techniques. The improvements are due to increased cache hit rates which result from a corresponding reduction in inter-thread cache contention.

The remainder of this paper is organized as follows. Section 2 provides an empirical analysis of thread conflicts and cache contention in simultaneous multithreading machines and the motivation behind our proposed approach. Related work is discussed in Section 3. Next, Section 4 presents an overview of the architectural support for enhancing thread symbiosis in simultaneous multithreaded architectures. The effectiveness of the proposed approach in improving performance and exploiting cache resource efficiency in an SMT architecture is presented in Section 5. Finally, the paper is summarized in Section 6.

2. Motivation

2.1. SMT contention on memory resources

Unlike intra-thread conflicts, inter-thread conflicts in SMT are typically non-repeatable and thus difficult to predict [10]. As such, eliminating inter-thread conflict penalties requires fundamentally different techniques than those that work on single-thread applications. Figure 1 shows the second level cache miss rates for pairs of SPEC 2000 benchmarks that were run on a simulated SMT architecture. In addition to the net miss rate, the cause of the cache misses are reported for each job pair. The job pair 164.gzip_164.gzip for example, has a 34% miss rate where 25% of these misses are due to inter-thread conflicts. On average, inter-thread conflict misses account for more than 30% of the overall misses.

2.2. Multithreading cache behavior

To understand the inefficiencies of current cache hierarchies in relation to SMT architectures, it is necessary to examine the run-time behavior of simultaneous threads. Often, program behavior varies over time, meaning that an application may oscillate between phases of high and low memory system demand. A high-level approach to co-scheduling jobs would be to pair memory intensive jobs with those that

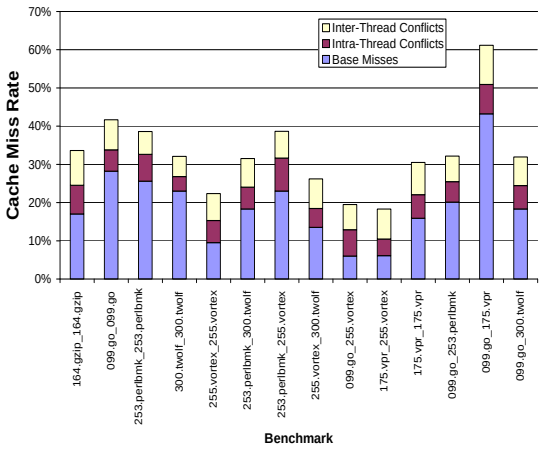


Figure 1. SMT cache contention.

have low memory demands. This approach however offers no solution for job mixes that have similar memory system demands. We propose a fine-grain approach to analyzing the cache activity of each running job, which is used to guide the operating system job scheduling software for any type of application mix. In this model, the cache is viewed as a series of contiguous groups of sets called *Super Sets*. The number of accesses to each Super Set is recorded in hardware in order to identify the cache regions which are used most frequently by a given application. The operating system scheduler uses this information to select job pairs that have as few as possible potentially conflicting cache regions.

The tracking of cache region activity is illustrated in Figure 2, through a set of heat maps which shows the temporal data cache activity of four SPEC 2000 benchmarks. The x-axis of each graph is the time measured in samples of five million clock cycles. Along the y-axis is cache position, based upon the super set number. The color of each super set indicates cache activity, dark shades represents very low usage and light shades very high. These maps demonstrate that program behavior in the cache changes not only temporally, but also spatially with some regions hosting the majority of overall cache activity. For instance, *164.gzip* and *253.perlbmk* both have very active bands (groups of super sets) with very high access activity relative to the average super set activity at a given time interval. Similar behavior is demonstrated when tracking the number of incident misses occurring within each cache super set. Together both the active super set accesses and misses indicate a high probability of inter-thread conflicts, since the individual threads have high resource demands at the associated super set location. As such, we believe the ability to track and predict which cache regions will be active by each thread is impor-

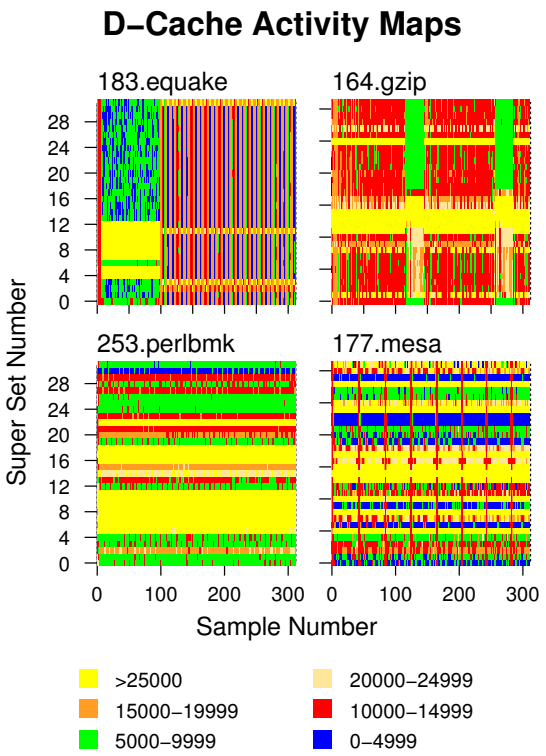
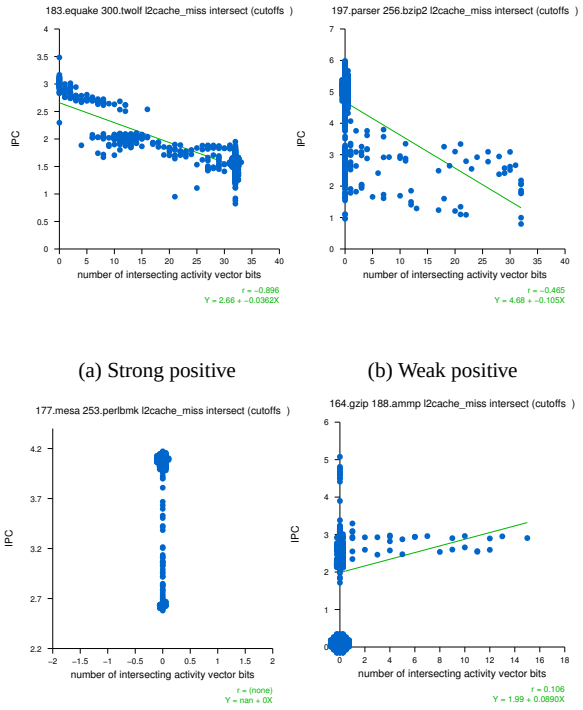


Figure 2. Activity maps of the level-1 data cache for 4 SPEC2000 benchmarks.



(a) Strong positive (b) Weak positive

(c) No discernible variance (d) Weak negative

Figure 3. Four types of correlation plot.

tant for minimizing inter-thread conflicts, and thus improving SMT throughput performance.

2.3. Activity vectors as performance indicators

Recognizing and predicting fine grained cache behavior is only worthwhile if it correlates with performance. The SMT simulator described in Section 4.1 was used to test this correlation as a means for motivating the operating system scheduling techniques proposed in this paper. Starting with nine SPEC 2000 benchmark applications, all forty-five possible pair-wise combinations were run in the absence of operating system level scheduling or thread switching policies. These workloads were simulated for one billion instructions and both the IPC and the activity vectors for each job were recorded at intervals of 5 million cycles. Figure 3 shows the results of these simulations where IPC was plotted against a number of intersecting bits (both executing threads with high activity in a given super set) in each of the caches and for miss and activity. For each test, a line was fit to the data. The pairings exhibited four general behaviors, as shown in Figure 3. The simulations demonstrate the correlation between achieving higher IPC when finding fewer high activity intersections. We make the distinction between weak and strong positive correlation (with 17 and 16 members in each category respectively, and the remainder with weak negative or zero correlation) by assessing when the correlation coefficient, r^2, crosses above 0.5. It can be concluded from this data that as interference between vectors increases, performance, measured by IPC, decreases.

Activity Vector	Average Slope	Average Correlation (r^2)
D-cache use	0.109	0.326
D-cache miss	-0.023	0.142
I-cache use	0.085	0.395
I-cache miss	0.007	0.177
L2-cache use	-0.022	0.143
L2-cache miss	-0.151	0.331

Table 1. Correlation of activity vector intersect to IPC.

The slope (IPC v. active intersection of supersets) and correlation factor of Figure 3 were averaged over all 45

tests and the results are shown in Table 1. In this table, a large negative slope is desirable. That is, the more intersecting bits in the activity vectors, the lower the resulting IPC performance. The table reveals that the L2 cache has the steepest slope, which is expected, because the overlap in the activity vectors would lead to an increase in second level cache misses. Thus, a reduction in the number of second level inter-thread conflict misses should limit the overall off-chip memory latency. This relation between second level conflict misses and performance is supported in the literature by [5].

2.4. Cache activity predictability

For a processor to leverage run-time cache activity information, it needs a mechanism for recording past cache activity so that it can predict future activity. A number of prediction methods were analyzed in order to provide insight into the potential success rate of a cache activity based job scheduler. Table 2.4 presents the results obtained by using the activity from the previous time interval to predict the activity of the next interval. The results for the instruction, data, and second level caches indicate that by simply predicting that the activity for the next sample will be the same as the previous sample, an accuracy between 82% and 95% can be achieved. Although a more sophisticated prediction algorithm could have been used for this study, such as phase based prediction schemes proposed in [17, 18], the accuracy of a single phase model is quite good considering the lack of algorithm complexity. The predictability of fine-grained cache behavior is further explored in [8].

Activity Vector	Use Predictability	Miss Predictability
D-cache	82.3%	93.6%
I-cache	94.9%	90.3%
L2-cache	93.8%	94.6%

Table 2. Predictability of activity vector bits, 100 million cycle samples.

3. Related work

There have been numerous studies focused on improving the throughput performance of simultaneous multithreading processors. The techniques for solving this class of problems is covered by a range of disciplines, including: microarchitecture level solutions, operating system thread scheduling, and compiler level optimizations. The SMT architecture was introduced by Tullsen et al. in [23] as a means for improving the utilization of wide issue super-scalar processors. By allowing the processor to execute instructions from independent software contexts at the same cycle, periods of low instruction level parallelism (ILP) were offset by the thread level parallelism (TLP) provided by the other contexts. Although this thread model has proved to improve the throughput performance of multi-job workloads, performance bottlenecks related to the shared hardware resources on these processors have placed limitations on their efficiency. In particular, the shared cache hierarchy, and the instruction fetch and issue logic have emerged as components that require special attention in an SMT environment.

In [21], Tullsen et al. studied instruction fetch and issue techniques at the microarchitectural level that provided the best processor utilization for a set of jobs on an SMT processor. The ‘ican’ issue policy was introduced as a method for assigning fetch and issue priorities to the hardware contexts resident on the machine. Lo et al. examined additional techniques for maximizing instruction throughput in [13]. Hily and Seznec demonstrated in [4] that the performance of multithreaded systems was limited by the cache hierarchy, in particular, the increase in bus contention introduced by the additional memory requests in an SMT machine model. This observation inspired later studies that focused on improving the latency tolerance of SMT architectures,

In response to the importance of resource contention between threads associated with the cache hierarchy, there have been a number of operating system level solutions to solving this problem. Jack Lo et al. demonstrated in [11] that database applications which suffered from inter-thread conflict misses could be improved significantly by using the operating system to change the way virtual memory pages were mapped to their physical counterparts. This approach made it less likely that data belonging to different thread contexts would map to the same cache sets, thus reducing the amount of inter-thread conflict misses. Symbiotic job scheduling [19] was introduced by Snavely and Tullsen as a method to improve performance by making the operating system co-schedule threads that were likely to ‘behave’ well with one another. While this technique exposed the potential headroom for using the operating system to advance SMT performance, it did not reflect closely enough the runtime environment of a real operating system job scheduler. Parekh, Eggers, and Levy proposed exposing microarchitectural level performance information to the operating system job scheduler in [15]. This work showed IPC improvements on the order of 10%, in a simulated operating system environment.

Studies of the impact of compiler optimizations on SMT performance [12] show specific cases where an SMT processor can benefit from changing the compiler optimization strategy. In particular, they showed that cyclic iteration scheduling is more appropriate for an SMT processor because of its ability to reduce the TLB footprint. In addi-

tion, the work showed that software speculative execution can be detrimental to SMT systems, because it decreases useful instruction throughput. According to [12], although the latency-hiding benefits of software speculation may be needed less on an SMT, the additional instruction overhead introduced by incorrect speculation might severely degrade performance.

Kumar et al [10] examine instruction cache aware compilation for SMT using weighted call-graph layout techniques to balance the placement of procedures across the shared instruction cache. This work presented optimization techniques that can be applied to multiple programs compiled at the same time in anticipation of being co-scheduled on an SMT processor. The best compilation technique results in an average 11% performance improvement in the instruction cache performance when attempting to eliminate inter-thread cache conflicts. In the case where the compiler has access to only one of the co-active threads, improvements were limited. Overall, this illustrates the potential of compilation techniques to aid SMT effectiveness, but indicates compilers do not have influence over run-time behavior.

In [20], the overall cache demands of threads is predicted using fine grained cache monitoring. By tracking which cache lines and and sets are most recently used and how often they are used, they can track overall cache demand for each thread. Threads are allocated a different number of sets within the cache based on predicted hit rates, and scheduled based on a greedy static assignment. Our work differs in that we allow adjustment to accommodate different program phases, and do not explicitly forbid inter-thread conflict by partitioning the cache. Rather, we attempt to schedule around predicted areas of conflict.

4. Experimental approach

4.1. Simulator details

Resource	Configuration	Associativity	Latency
Branch Predictor	counter	-	-
Issue Width	8way	-	-
SMT contexts	2	-	-
D-cache	32K	4way	1 cycle
I-cache	32K	4way	1 cycle
L2-cache	512K	8way	12 cycle
Memory	-	-	300 cycle

Table 3. Xsim configuration.

Before implementing activity based scheduling on a real system, the concept was validated on an SMT simulator called Xsim. Xsim is a cycle accurate simulator derived from IMPACT LSIM, originally developed at the University of Illinois [2]. It was modified to support SMT in addition to the activity vectors and operating system scheduling algorithms described in this paper. In addition to being highly configurable, it runs executable code generated by the IMPACT C compiler. The processor model configuration used in this study is shown in Table 3. The following sub-sections detail the experiments and the associated Xsim features which provide the testing infrastructure.

4.2. Hardware support: activity vectors

Activity vectors were modeled in Xsim as bit vectors to approximate the registers required in an actual hardware implementation. Figure 4 illustrates the hardware structures required for activity vector support. Each hardware context has an associated group of registers or *Activity Counters*. The number of counters is equal to the number of cache super sets, which are described in Section 2.2. The low order bits of the cache set component of a memory address are used to index the activity counter associated with each cache super set. In addition to the activity counters, each hardware context has two activity vector registers that are visible to the operating system software. Each bit in the activity vectors correspond to a super set index number. The activity counters are used to record both the number of accesses and misses which occur for a given super set. When the count exceeds a threshold value, a bit in the appropriate use or miss activity vector is set. In Figure 4 T_0 is the miss vector associated with hardware context 0. To measure the overlap in cache activity between two hardware threads, a logical AND of the vectors for each context is performed, the resultant vector identifies the cache super sets which both threads consistently address. This hardware is further explored in [7].

4.3. Hardware assisted job scheduling algorithm

The scheduling algorithm included in Xsim is based loosely on the Linux scoring system. Each thread in the run queue is given a score based on a number of factors, and the threads with the lowest scores are scheduled first. The major difference between this implementation and Linux lies in the determination of the scores. In Linux, the score is set only by a user and OS defined priority, and incremented as the thread ages. Under this mechanism older threads end up with a higher score, and hence a lower priority, which decreases the response time of shorter jobs. Our model differs by adding more factors to the scoring process. Some examples of these elements include the number of interfering bits in the activity vectors and information from performance counters such as cache misses and number of operations retired. For fairness, the number of samples that the

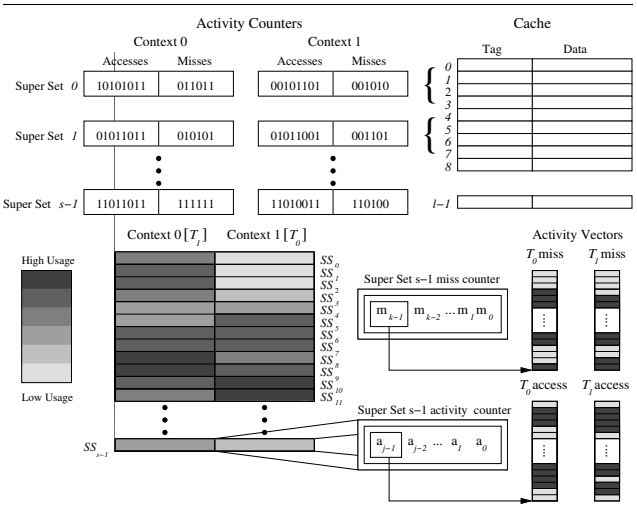


Figure 4. Illustrating the basic construction of activity vectors.

thread has previously been resident in the processor is also included. Additionally, each element is assigned a weight by which it is scaled before contributing to the final score. The overall scheduling score is simply the sum of the products of the various factor scores and their weights, as shown in Figure 4.3. Flexibility of the model is achieved by simply adjusting the weights.

$$
\begin{aligned}
S &= \sum(w_i * s_i) \\
S &= \text{Switching Score} \\
w_i &= \text{factor weight} \\
s_i &= \text{factor score}
\end{aligned}
$$

(Example Factors: L2 activity vector intersects, total flops)

Figure 5. Calculation of the scheduling score.

The thread scheduler has three phases. First, it removes all completed threads. Next, if no threads have completed, it determines which running thread has the greatest number of samples resident. This thread will be switched out unless its switching score is lower than or equal to the scores of all of its potential replacements. After determining which thread or threads are candidates for removal, each potential eviction as well as all pending threads are given a switching score. This score is based upon the number of samples the threads have already run, various hardware performance counters, and the activity vector interference. Two steps are needed to determine this interference. First, the activity vector is predicted for each active thread, then a logical AND of this vector is performed against the predicted activity vectors of each job in the run queue. This is illustrated in Figure 6. The figure is simplified in that the switching score is determined solely by the predicted interference in one activity vector rather than the sum of several factors. In the figure, it has been determined that thread zero will retain one of the active thread contexts. As such, each of the other three threads is compared against thread zero's predicted activity vector. Since thread two has the fewest conflicting activity vector bits, it is chosen to be scheduled for the next scheduling sample. If the lowest scoring non-resident thread has a lower switching score than the thread marked as the candidate for removal, the two threads are switched.

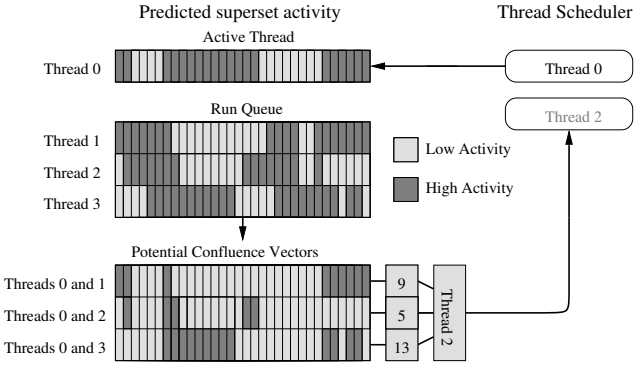


Figure 6. Illustration of the simplified scheduling algorithm.

Three main simplifications are made in our scheduler. First, no accounting is made for scheduling overhead and OS behavior as was studied in [16]. The next is that there are no interrupts in the system and all threads are kicked off at the same time. This abstracts away a good deal of the complexity of the problem as the number of threads stays constant (except when a thread finishes) and scheduling decisions only have to be made on sample boundaries. This may be necessary, however, as it allows for highly repeatable experiments. The final simplification is that thread priority can not be assigned by a user. All simulated jobs are set to have equal priority on the system.

4.3.1. Linux scheduler implementation The Linux scheduler from kernel version 2.6.0 was modified to support activity vector based scheduling and was run on an Intel Pentium 4 Xeon processor with hyperthreading enabled. This processor has two contexts, or virtual CPUs, that share the cache hierarchy, data registers, and func-

tional units. The default Linux scheduler runs a symmetric multiprocessing (SMP) scheduling algorithm on each virtual processor to select the next job to run. This algorithm was adjusted to include activity vector information in the job selection criteria. Rather than treat the activity vector information as a score in the scheduling priority algorithm, the default scheduler is first used to select the best job to run. Then, it queries the activity vector scheduler to search for a job that is expected to have fewer resource conflicts with the job running on the other virtual processor. If the activity scheduler can not identify a better job, then the default job is allowed to run. Since the Pentium processor used in these experiments can not be modified, activity vector support was modeled in software. The details of which are described below.

In order to expose cache activity information to the operating system each benchmark application is profiled, during which time the activity vectors are generated. The Intel Pentium 4 Xeon processor does not provide a mechanism for collecting cache miss statistics on a per line granularity. Thus, in order to approximate the actual memory behavior of a given job, each benchmark application is run under the Valgrind memory simulator. Valgrind provides a model of the actual cache hierarchy of the host machine along with a record of runtime program information, such as the instruction count. This simulation stage is used to record all memory requests and misses associated with a given cache super set. The access and miss counts for each super set are recorded in software activity counters every program phase interval of 50 million instructions. As described in Section 4.2, a bit is set in the activity vector if the corresponding counter exceeds a threshold value. The resulting activity vectors for each cache level are written to a text file in ASCII format. These vectors are later passed to the operating system kernel memory space, where they are referenced by the job scheduling algorithm.

4.3.2. Algorithm details Algorithm 1 is a pseudo code representation of the activity based scheduling module and its interface to the kernel scheduling software. The scheduling module is a tool that provides a user level interface to the kernel scheduler, which can be used for both collection of statistics, and the activation of the scheduling algorithms discussed in this paper. The module was designed so that it could be recompiled and run without rebuilding the Linux kernel. To achieve this goal, the module exports 2 functions to the Linux schedule() function. At runtime, the scheduler queries these functions in order to select which job to activate at each schedule invocation. The first function th_sample_cache_vector(), shown in Algorithm 2, is used to sample hardware performance counters and record their associated data for each of the jobs running on the system. The second function, th_schedule_cache_vector(), shown in Algorithm 3, is used to select the next job to run based upon analysis of the cache activity vectors.

Algorithm 1 Modified linux schedule() function.

```
Require: current_active_task
Ensure: next_task_ready
  if need_resched then
    log_task_runtime()
    th_sample_cache_vector(task)
    update_context_switch_count()
  end if
  if pick_next_task then
    if runqueues_not_balanced then
      load_balance()
    end if
  end if
  next_task = select_highest_priority_task()
  next_task = th_schedule_cache_vector(next_task,
      other_cpu_runq, task_priority_list)
  recalc_task_prio(next, timestamp)
  if switch_tasks then
    unmodified linux source
    ...
  end if
```

Algorithm 2 Performance monitoring function.

```
Require: current_running_task
Ensure: Global perf counter tables updated
  if logging_l2cache_perf then
    sample_perf_counter(L2CACHE, cpu_id)
    log_counter(bmark_name)
  end if
  sample_perf_counter(icount, cpu_id)
  store_icount(task_id)
```

The schedule() function of Algorithm 1 has two important code regions that were chosen as entry points for the activity based scheduling module. The first section is labeled 'need_resched', here the available tasks either get removed from the run_queue if they have completed, or their scheduling priority gets updated. The th_sample_cache_vector() function was inserted into this section to monitor the state of the cache system performance counter registers associated with the presently active tasks on each of the logical CPUs. The performance counter information was stored internally in the kernel memory space for use in both statistical reporting, and as a reference point for the scheduler optimization. The second important region in the function schedule() is named 'pick_next_task'. Here, as the name suggests, the kernel selects the next job to activate on the CPU that the scheduler is currently running. The function th_schedule_cache_vector() was introduced here to

select the best job to run based upon the activity vector information that was passed to kernel memory by the user.

Algorithm 3 Vector scheduling algorithm.

Require: ready_task, runq_other_cpu,
  active_task_other_cpu, task_priority_array
Ensure: next_active_job
  initialize_job_weights()
  get_vectors($ready\_task$, $other\_task$, $icount$)
  for all Cache_Levels **do**
    res_vec = AND($vector\_ready$, $vector\_other$)
    $overlap\_bits$ = count_result_bits(res_vec)
    update_weight_function($overlap\_bits$)
  end for
  for all tasks_in_run_queue **do**
    compare_vectors($task$, $task\_other\_cpu$)
    if $weight < best\_weight$ **then**
      $best\_task = last\_job\_in\_queue$
    end if
  end for

4.3.3. Kernel module details Upon completion of the profiling stage for each of the benchmarks, the user level profiling tool is used to copy the contents of the vector files into the kernel memory. This tool is a device driver that enables a user to control when the scheduler should activate the th_sample_cache_vector() and th_schedule_cache_vector() functions. When the user interface is invoked, the default Linux scheduler is active. The user is prompted for the file names of the activity vector files, then each is read into kernel memory for later use in the scheduling algorithm. Once the vector files have been read in successfully, the user selects the desired scheduling algorithm, then starts all of the benchmark applications in the workload set. The total execution time for the workload set is recorded, at which point, the user can command the kernel to return to the default scheduling mode.

4.3.4. Linux scheduler opportunity Figure 7 illustrates the scheduling opportunities available to the Linux schedule for a range of workloads. Each time the scheduler is invoked on one of the logical CPUs it has a pool of ready jobs to choose from. The scheduler was instrumented in order to support runtime analysis of the job pool. The data in Figure 7 show the ratio of the number of times the scheduler was invoked to the number of times a benchmark could have been selected in order to reduce cache resource conflicts. Note that in order for the activity based scheduler to be invoked, the job running on the other virtual processor must belong to the experimental workload. On average, more than 45% of schedule invocations trigger the activity based scheduler. The remainder of schedule invocations are to handle operating system tasks that are independent of the workload set. This indicates that there is considerable headroom for the selection of better jobs for co-scheduling.

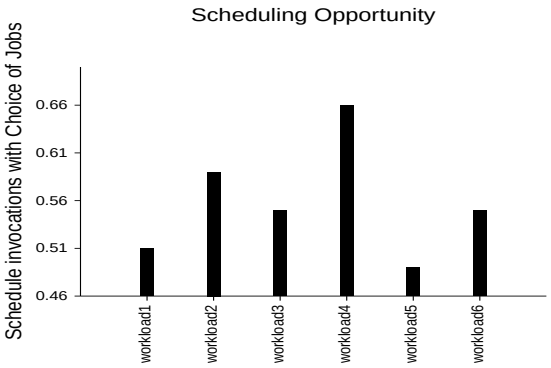


Figure 7. Default linux activity aware scheduling opportunity.

5. Results

5.1. Simulation scheduling results

The first experiments in scheduling were performed using the simulator outlined in Section 4.1 and the results were compared against the round-robin priority scheme. The results are encouraging given that the experiments were performed on a simulator. Increases of up to 7% in overall IPC were observed in some workloads where L2 activity vector based scheduling was used, as shown in table 5.1. In addition, the inter-thread cache interference was reduced by an average of 10% across the simulated workloads and as much as 54% for the workload labled Job_1.

These results are promising due to the limited program coverage achieved by the simulator. Since simulation times are considerably long, the simulator, as described in Section 4.1, was configured to run only one billion instructions. It is expected that running the jobs to completion would bring an improvement in system performance results since the job scheduler would have had greater opportunity to build more accurate activity vectors.

Using intelligent simulation sampling in the future, such as the work proposed by [24], should allow us to simulate longer execution times and therefore more scheduling decisions. The final limitation to our experiments, having only 4 threads to choose from, was a choice made also because of simulation time constraints. With a larger number of threads available, the chances of finding a complementary thread are higher. However, it also means more potential combi-

nations which need be tested to obtain an accurate profile. As such, we chose to limit the number of threads in a workload to four.

$Job_{\#}$	Benchmark Configuration	% ITKO Reduction	% IPC Gain
Job_1	gzip.gzip.mcf.equake	54.0%	3.6%
Job_2	gzip.gzip.ammp.twolf	10.5%	4.5%
Job_3	gzip.mesa.mcf.equake	39.5%	3.0%
Job_4	gzip.mesa.equake.equake	47.0%	2.4%
Job_5	gzip.parser.perlbmk.twolf	-9.2%	1.4%
Job_6	mesa.mesa.parser.twolf	10.3%	4.8%
Job_7	mesa.mcf.perlbmk.bzip2	-4.1%	3.7%
Job_8	mesa.ammp.perlbmk.twolf	-6.3%	3.0%
Job_9	mesa.parser.parser.twolf	-0.7%	4.0%
Job_{10}	mcf.mcf.bzip2.bzip2	-3.3%	5.4%
Job_{11}	mcf.equake.perlbmk.twolf	1.7%	3.0%
Job_{12}	mcf.perlbmk.perlbmk.bzip2	13.0%	12.1%
Job_{13}	ammp.ammp.parser.parser	-2.7%	3.0%
Job_{14}	ammp.twolf.twolf.twolf	1.9%	6.1%
Job_{15}	parser.parser.perlbmk.bzip2	1.3%	5.7%
Mean		10.2%	4.4%

Table 4. Inter-thread kickout reduction and IPC improvement between base and activity vectors.

5.2. Linux hyperthreading results

Workload	Benchmark Configuration
WL_0	gzip.vpr.gcc.mesa.art.mcf.equake.crafty
WL_1	parser.gap.vortex.bzip2.vpr.mesa.crafty.mcf
WL_2	mesa.twolf.vortex.gzip.gcc.art.crafty.vpr
WL_3	gzip.twolf.vpr.bzip2.gcc.gap.mesa.parser
WL_4	equake.crafty.mcf.parser.art.gap.mesa.vortex
WL_5	twolf.bzip2.vortex.gap.parser.crafty.equake.mcf

Table 5. Linux workloads.

The results reported in this section were collected from a set of workloads each consisting of 8 benchmarks from the SPEC 2000 suite. The benchmarks were all run to completion using the training input set. They were started at the same time and set to run in the background so that the OS could manage their scheduling priorities. The benchmarks used in each workload are shown in Table 5 as a reference for the data reported in this section. In addition to these jobs, the operating system scheduler also manages the system level jobs that run in the background on any Linux system.

The activity based Linux scheduler was configured to collect runtime scheduling statistics based upon the hardware performance counters. The cache activity information was used to to evaluate how well the job scheduler performed at minimizing potential resource conflicts. Figure 8 shows the percentage of scheduling periods in which the default scheduler chose jobs that were classified as either 'good', 'bad', or 'moderate', relative to the resulting resource contention. These classifications were derived by comparing the cache activity of each job in the pool with the activity of the job running on the other virtual CPU. Rather than using this information to change the schedule it was instead used to assess the effectiveness of the default scheduler. From Figure 8, one can see that for each of the 6 workloads, the default scheduler makes a poor scheduling decision for 30%-40% of the scheduler invocations. This shows that there is a significant room for improvement for scheduling tuned to reduce resource contention. In addition, as the number of jobs in the pool increases, the opportunity for the activity based scheduler to find a good match for co-scheduling should also increase.

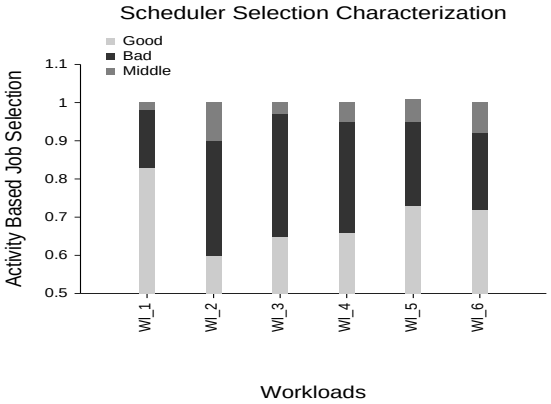


Figure 8. Activity aware scheduling decisions.

Figure 9 shows the breakdown of the job pool size for a given scheduler invocation. Each time the Linux scheduler was determined to have made a poor decision, the job search space was evaluated to identify the opportunity for finding a better choice. For roughly 85% of the 'poor' choices, there was at least one better job to choose from. The chances of finding 2 better jobs drops significantly to roughly 10%. These results suggest that systems with larger job mixes should expose more scheduling options to the operating system. In addition to providing a larger set of jobs to choose from, a broader set of applications would increase the chances of finding benchmark pairs that are good candidates for co-scheduling.

After analyzing the default scheduler, the activity based scheduler was enabled and the resulting performance for the set of workloads was measured and reported in Figure 10. For each workload, the L2 cache misses associated with the workload were recorded along with the IPC. Four out of the

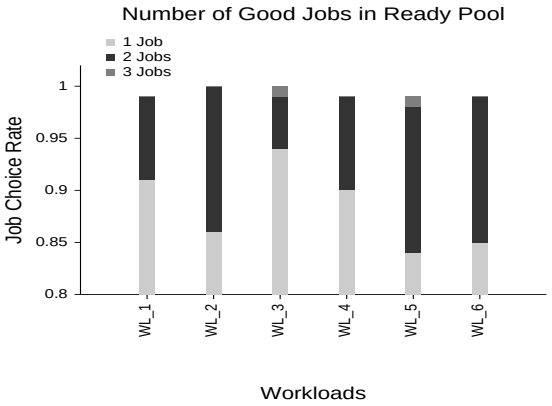


Figure 9. Job selection pool.

six workloads show an improvement in L2 cache misses of more than 4%, with a similar improvement in IPC. The net performance gain is dependent upon the job mix, which is expected since some of the workloads may contain a number of jobs with similar cache activities. The results are encouraging since this technique was implemented on real hardware with a commonly used operating system. As a final comparison for scheduling heuristics, this approach was compared to a performance counter driven schedule. On the Xeon processor, the resulting workload completion times mapped closely to the activity base approach.

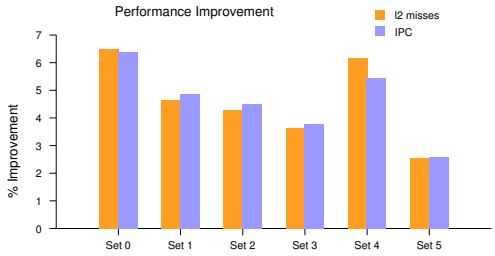


Figure 10. Performance improvement using activity vector scheduling.

In addition to the overall performance improvement, individual finish times for each application in the job set were evaluated. Figure 11 shows finish times for the applications, classified by the earliest finish time relative to base scheduling 'Early', the median 'Median', and the longest relative finish time 'Late.' Overall, both the early and median scores across the jobs indicate that in addition to a performance improvement many of the individual jobs have an earlier completion time as shown by the Early and Median bars. Though the worst case delay in completing individual applications is on average 12%, there is a slight gain overall due to the jobs that finish early.

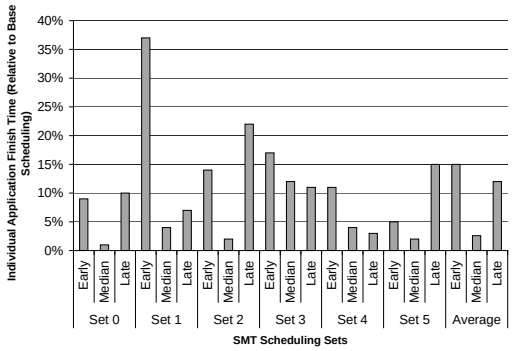


Figure 11. Measured individual (early, median, and late) finish times relative to base scheduling.

6. Summary

SMT is a promising solution to many of the general performance issues in modern computing. It offers a way to circumvent limitations to instruction level parallelism such as control and data flow dependencies and long latency memory operations. However, the trade off is the introduction of performance penalties related to the competition between threads for shared hardware resources. In this paper, we introduced a novel runtime technique for reducing the penalty associated with inter-thread resource conflicts in the cache system. By using hardware *Activity Vectors* to monitor the access patterns of the caches on a *Super Sets* granularity, we have provided the operating system a mechanism for using microarchitectural feedback to improve job scheduling. In a simulated environment, activity vector scheduling reduced inter-thread cache conflicts by an average of 10% and as much as 54% for a set of SPEC 2000 benchmark workloads. When implemented on a Pentium 4 Xeon processor with hyperthreading, improvements in both IPC and second level cache misses ranged from between 2% and 6%. Following on these promising results, further refinement of both the activity vector implementation and the scheduling algorithm should lead to even better performance improvements.

7. Acknowledgments

The authors would like to thank Joshua Stone for assistance in gathering the Valgrind memory data for the hardware experiments, Tipp Moseley and Dirk Grunwald for assistance with the Linux scheduler software, and the anonymous reviewers for their helpful insights and comments.

References

[1] T. Aamodt, P. Marcuello, P. Chow, P. Hammarlund, and H. Wang. Prescient instruction prefetch. In *Proc. of the 6th Workshop on Multithreaded Execution, Architecture and Compilation*, pages 2–10, November 2002.

[2] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[3] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–26, / 1997.

[4] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, 1997.

[5] S. Hily and A. Seznec. Standard memory hierarchy does not fit simultaneous multithreading. In *in Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4)*, 1998.

[6] Intel Corporation. Special issue on intel hyperthreading in pentium-4 processors. *Intel Technology Journal*, 1(1), January 2002.

[7] J. Kihm and D. Connors. Implementation of fine-grained cache monitoring for improved smt scheduling. In *Proceedings of The 22nd International Conference on Computer Design*, 2004.

[8] J. Kihm, A. Janiszewski, and D. Connors. Predictable fine-grained cache behavior for enhanced simultaneous multithreading (smt) scheduling. In *Proceedings of International Conference on Computing, Communications and Control Technologies*, 2004.

[9] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[10] R. Kumar and D. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[11] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, pages 39–50, 1998.

[12] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 114–124, December 1997.

[13] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[14] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Ilp versus tlp on smt. *Supercomputing*, November 1999.

[15] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors. Technical report, University of Wahington, Seattle, WA, May 2000.

[16] J. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.

[17] T. Sherwood and B. Calder. The time varying behavior of programs. Technical Report UCSD-CS99-630, University of California at San Diego, 1999.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[19] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[20] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, pages 117–, 2002.

[21] 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 *Proceedings of the 23rd annual International Symposium on Computer Architecture*, pages 191–202, 1996.

[22] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 54–58, 2000.

[23] D. M. Tulsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] M. VanBeisbrouk, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.