Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors *

Ali El-Moursy⋆, Rajeev Garg⋆, David H. Albonesi† and Sandhya Dwarkadas⋆

⋆Departments of Electrical and Computer Engineering and of Computer Science, University of Rochester
†Computer Systems Laboratory, Cornell University
elmours@ece.rochester.edu, {garg,sandhya}@cs.rochester.edu, albonesi@csl.cornell.edu

Abstract

The industry is rapidly moving towards the adoption of Chip Multi-Processors (CMPs) of Simultaneous Multi-Threaded (SMT) cores for general purpose systems. The most prominent use of such processors, at least in the near term, will be as job servers running multiple independent threads on the different contexts of the various SMT cores. In such an environment, the co-scheduling of phases from different threads plays a significant role in the overall throughput. Less throughput is achieved when phases from different threads that conflict for particular hardware resources are scheduled together, compared with the situation where compatible phases *are co-scheduled on the same SMT core. Achieving the latter requires precise per-phase hardware statistics that the scheduler can use to rapidly identify possible incompatibilities among phases of different threads, thereby avoiding the potentially high performance cost of inter-thread contention.*

In this paper, we devise phase co-scheduling policies for a dual-core CMP of dual-threaded SMT processors. We explore a number of approaches and find that the use of ready and in-flight instruction metrics permits effective co-scheduling of compatible phases among the four contexts. This approach significantly outperforms the worst static grouping of threads, and very closely matches the best static grouping, even outperforming it by as much as 7%.

1 Introduction

The microprocessor industry is focused on the development of Chip Multi-Processors (CMPs) of Simultaneous Multi-Threaded (SMT) cores for general purpose systems. IBM's Power5 [14], Intel's Montecito [13], and Sun's Niagara [9] are three examples of this type of microprocessor. While these initial multi-core efforts contain a limited number of processors and contexts, in the future, microprocessors are expected to support upwards of 100 simultaneous threads.

While much research effort is being expended on the development of multi-threaded applications and their efficient execution on multi-core chips, the most prominent near-term use of such processors will be as job servers running independent threads on the different contexts of the various SMT cores. With some parallel applications, offline characterization of thread phases may be viable for determining how to share context resources among the various threads of the application. In a job server environment, this is a much more difficult proposition. The co-scheduling of phases from different threads plays a significant role in the overall throughput. Consider the dual-core CMP consisting of dual-threaded SMT cores shown in Figure 1. Similar to Intel's Hyper-Threaded Xeon processor [6], each of the two threads that share an SMT core can occupy no more than half of the entries of each shared queue. Furthermore, like Xeon, the core is designed for good single thread performance, and therefore resources are not over-provisioned so as not to impact single thread pipeline speed. For instance, there are 128 integer and 128 floating point registers and an L1 Dcache size of 8KB (similar to Xeon).

In such processors, contention for resources can be severe depending on what threads are scheduled together on the same SMT core. Figure 2 shows the performance of the three possible options (relative to the first option) for scheduling the four threads in each of the ten evaluated workloads (discussed in detail in Section 3) on the two dual-thread processor cores. Due to widely varing resource contention among the three different thread groupings, the performance difference between the best and the worst cases is significant, as much as 25%.

This result motivates the need for thread phase co-scheduling algorithms in which thread phase-to-processor assignments are based in part on the *compatibility* of different thread phases in terms of how well they are able to share processor resources. While the scheduler may be im-

*This research was supported in part by NSF grants CSR/AES-0509270, CNS-0411127, CCF-0325931, ITR/CCF-0219848, ECS-0225413, INT-0117667, and EIA-0080124; by two IBM Faculty Partnership Awards; and by equipment and/or financial grants from Compaq, IBM, and Intel.

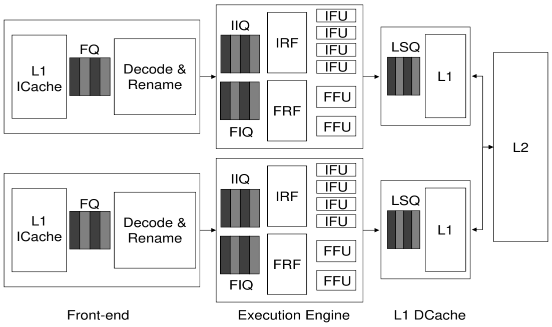

Figure 1. Dual core CMP of partitioned dual-threaded SMT processors (FQ: Fetch Queue; IIQ: Integer Issue Queue; IRF: Integer Register File; IFU: Integer Functional Unit; FIQ: Floating Point Issue Queue; FRF: Floating Point Register File; FFU: Floating Point Functional Unit; LSQ: Load/Store Queue).

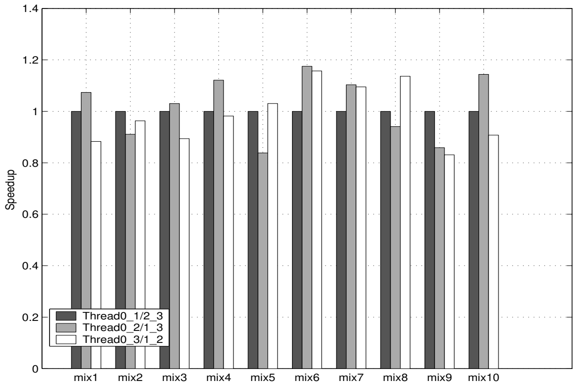

Figure 2. Overall performance for ten four-thread workloads. For each workload, the three bars correspond to the three possible ways that the four threads can share the two processors.

plemented at a fine grain in hardware, or coarser grain in software (we discuss these tradeoffs in Section 6.2), hardware performance counters can be used to gather the appropriate information required to assess compatibility among thread phases.

In this paper, we explore various means for accumulating this hardware information, and its use by thread co-scheduling policies, for a dual-core CMP of dual-threaded SMT processors as shown in Figure 1. We discover that register file and L1 data cache usage or contention are inconsistent metrics for making effective co-scheduling decisions. Rather, we find that more general information, namely the number of ready instructions, and the number of in-flight instructions, provides a very strong indication of thread phase compatibility, and provides a consistent result no matter how the threads are grouped when the measurement is made. Our co-scheduling policy at a fine-grain phase granularity closely matches the best static grouping of threads, and even exceeds it by as much as 7%. Perhaps more importantly, the approach avoids the significant performance degradation of a poor static grouping of incompatible threads. Furthermore, we find that the approach works well at a much coarser time interval, permitting the decision function to be assumed by the operating system.

The rest of this paper is organized as follows. Related work is discussed in the next section, followed by our experimental methodology in Section 3. In Section 4, we present an evaluation of resource contention in SMT processors, followed by a description of various candidate thread co-scheduling policies in Section 5. Results are presented in Section 6, and we conclude in Section 7.

2 Related Work

The topic of job scheduling on both uni- and multi-processor systems has been researched for decades. However, much less attention has been paid to the subject of job scheduling on an SMT machine. We focus the discussion in this section on related work in scheduling for SMT machines.

The first body of work to address scheduling in SMT machines was by Snavely et al. [18, 19, 20]. Their efforts focused on a job scheduler called *SOS* (Sample, Optimize, Symbios) for monolithic (not partitioned, as in our microarchitecture) SMT machines. The approach uses exploration by running each thread combination for a certain interval of time. During the sampling period, SOS gathers hardware statistics such as functional unit and queue conflicts. The subsequent optimization phase uses this information to determine the symbiosis among the threads in order to make the best scheduling decision.

Two efforts explore avoiding the high cost of exploration in the SOS approach by monitoring resource activity to make on-the-fly SMT scheduling decisions without exploration. The thread scheduling work by Parekh et al. [15] is the most similar to our own. They examine a variety of metrics for determining the best way to schedule threads on an SMT processor, and experimentally determine that an IPC-based thread-sensitive scheduling algorithm that schedules the highest IPC threads together is the best performing scheduler.

Settle et al. [8, 17] use an L2 cache *activity vector* to enhance the Linux scheduler for SMT machines. As we discuss in more detail in Section 5.1.1, this approach dynamically tracks the usage of the L2 cache for each thread on a set by set basis and creates per-thread cache activity vectors that indicate the most heavily utilized sets. The logical ANDing of two activity vectors creates a *cache conflict* *vector* that is used by the Linux scheduler to determine how best to co-schedule threads on an SMT machine. Chandra et al. [3] also address L2 cache contention, but on a dual-core CMP organization using single-threaded processors. They propose the analytical *Inductive Probability* (Prob) model to produce a very accurate prediction of the number of extra L2 cache misses due to inter-thread contention. Fedorova et al. [5] propose a multi-threaded data locality conflict model to estimate the L2 cache misses for co-scheduled threads. Their approach requires examining all possible combinations of threads to make a scheduling decision for a multi-core multi-threaded machine.

Exploration-based approaches are viable only with a limited number of thread combinations, and therefore, are inappropriate for use with the large number of SMT cores and threads expected in the future. Moreover, although cache conflicts can certainly be a major source of thread phase compatibility, threads that can co-exist cache-wise may be incompatible due to serious conflicts for other resources. In a CMP of SMT processors with partitioned queues as explored in this paper (Figure 1), the additional primary sources of inter-thread contention are the functional units and the register files. Unfortunately, it is very difficult to reconcile these different sources of conflict in order to achieve consistently close to the optimal grouping across a varied set of workloads. Rather, what is needed is an online scheduling approach that employs simple metrics to discern inter-thread phase compatibility independent of the cause of contention. The development and evaluation of such policies is the subject of this paper.

Finally, we mention the work by Cazorla et al. [2] on SMT hardware resource allocation. This effort attempts to allocate SMT hardware resources to threads in a way that maximizes performance. Although their objective is different than ours (maximize performance given thread phases that are running together, versus determining which phases to co-schedule), there are elements in the two approaches that are similar, such as identifying sensitive and insensitive threads (called "fast" and "slow" threads in [2]).

3 Methodology

Before exploring the different causes of resource contention in the next section, we discuss the methodology used to obtain the experimental results.

3.1 Simulation Infrastructure and Machine Configurations

Our simulator is based on Simplescalar-3.0 [1] for the Alpha AXP instruction set. Like the Alpha 21264 [7], the register update unit (RUU) is decomposed into integer and floating point issue queues and register files, and a reorder buffer (ROB).

SMT support in each core has been added by replicating the fetch control, rename, and ROB per thread. Per-thread fetch, issue, and load/store queues are implemented as discussed in Section 1, similar to Pentium 4's Hyperthreading approach [6]. We model a CMP of two dual-thread SMT cores using our simulator, in a job-scheduling environment, *i.e.*, where the threads are independent applications. The memory hierarchy is modeled in significant detail, including accounting for bus contention between the L1 caches and the unified L2 cache that is shared between the two cores. Table 1 shows the simulation parameters for each SMT core, and the L2 cache and main memory.

Table 1. Simulator parameters for each core and the shared L2 cache and main memory.

Branch predictor	comb of bimodal and gshare
Bimodal predictor entries	2048
Level 1 table entries	1024
Level 2 table entries	4096
BTB entries, associativity	2048, 2-way
Branch mispredict penalty	10 cycles
Fetch policy	ICOUNT.4.32 [21]
Fetch width	32 per core
Fetch queue size	32 per thread
Integer Issue queue size	20 per thread
FP Issue queue size	20 per thread
Load/Store queue size	28 per thread
Issue width	6 per core
Dispatch and commit width	6 per core
Integer Physical Registers	128 per core
FP Physical Registers	128 per core
Reorder Buffer Size	512 per thread
Integer FUs	4 per core
FP FUs	2 per core
L1 ICache	32KB, 2-way per core
L1 ICache Latency	1 cycle
L1 DCache	8KB, 2-way per core
L1 DCache Latency	2 cycles
L2 Cache	2MB, 8-way
L2 Cache latency	20 cycles
TLB (each, I and D)	128 entries, 8KB page size, fully associative, per thread
Memory latency	100 cycles

We model the migration of a thread from one core to the other by flushing the pipeline, invalidating the cache, and switching the context to the other core. We arbitrarily select one of the two threads to be switched as our experiments indicate that selecting either one has roughly the same performance impact.

3.2 Benchmarks and Multi-Threaded Workloads

Table 2 lists the multi-threaded workload mixes of SPEC2000 benchmarks used to evaluate the different scheduling policies. The order of the benchmarks in the table represents their order in sharing the cores for the baseline scenario. The case where the first two benchmarks share one core while the second two share the second core is represented in the results as P_CMP_1. In P_CMP_2 and P_CMP_3, the first benchmark shares a core with the third and fourth benchmarks, respectively. Accordingly, the other

Table 2. Multi-threaded workloads.

Workload Name	Benchmarks Included
mix1	mgrid, equake, vpr, galgel
mix2	gcc, swim, art, lucas
mix3	swim, equake, art, mgrid
mix4	swim, equake, mgrid, galgel
mix5	perlbmk, mgrid, vpr, galgel
mix6	gcc, mcf, swim, lucas
mix7	gcc, mcf, art, swim
mix8	swim, mesa, equake, art
mix9	art, swim, galgel, applu
mix10	mesa, swim, galgel, art

two benchmarks share the other core.

Each individual benchmark was compiled with gcc with the -O4 optimization and run with its reference input set. We report results for both fine-grain (100K cycles) and coarse-grain (100M cycles) interval schemes. For both approaches, we fast-forward each benchmark by 4 billion instructions. When comparing the various fine-grain approaches, we run for one 100K cycle interval before gathering performance data. This permits each of the dynamic scheduling policies to gather statistics for an interval and make an initial pairing. We then run each benchmark for the next 100 million instructions, and therefore report performance results from executing 100 million instructions from each thread (400 million instructions total) in the workload mix. In comparing the fine and the coarse-grain schemes, after fast-forwarding, we first run for one coarse-grain interval (100M cycles) before we begin measuring performance. We then run each benchmark for 500M instructions (2B instructions in all).

3.3 Metrics

As a performance metric, we chose the geometric mean of the relative IPC ratings of the four threads:

$$\sqrt[4]{\prod_{4} \frac{IPC_{new}}{IPC_{old}}}.$$

Since the metric used is a relative measure, the use of a geometric mean avoids the issue of which configuration is placed in the numerator or in the denominator [11]. The geometric mean equally weighs a performance improvement in one thread and an identical performance degradation in another simultaneously executing one. The harmonic mean would overly penalize a performance degradation, while the arithmetic mean would overly reward a performance improvement. The geometric mean avoids skewing the results in either direction when presenting relative performance.

4 Inter-Thread Resource Contention in SMT Processors

Assuming an SMT processor in which the major queues (fetch queue, issue queues, and load-store queue in our processor) are equally partitioned among the two threads (or, equivalently, shared queues in which each thread may occupy no more than one-half of the entries), the major machine resources for which thread phases may contend are the functional units, register files, and cache hierarchy. This contention for resources can potentially degrade performance, but it is unclear how this impacts the optimal co-scheduling of thread phases.

In this section, we perform a series of experiments in which we individually remove the sources of contention and observe the impact on the variation in performance with the three different thread groupings. This provides an indication as to how well directly measuring these sources of contention can be used to guide thread co-scheduling decisions. Figures 3 and 4 present the breakdown for two representative workload mixes. The first set of three bars in each figure presents the performance on our baseline.

4.1 Register File Contention

Register files and issue queues are relatively long latency resources. Due to the true dependencies in the instruction stream, these resources may be occupied for very long periods of time. The contention among the threads for these resources is very important and critical to the overall performance. Raasch et al. [16] demonstrated that providing each thread with an equal share of the machine queues as is done in Intel P4 Hyperthreaded processors [10, 12] performs better than sharing the queue resources. Register files, on the other hand, should be shared to permit a given thread to hide the effective cache miss latency. The second set of three bars in Figures 3 and 4 demonstrate this impact by doubling the register file size and equally partitioning it among the two threads sharing a CMP, thereby eliminating any register file contention. Much of the difference between the performance of the different thread pairings is eliminated, indicating that the contention for register file resources with some thread groupings may be significant, although insignificant with other groupings.

4.2 Functional Unit Contention

In order to evaluate the impact of functional unit (FU) contention on thread compatibility, we performed experiments in which we also (in addition to doubling and partitioning the register file) doubled the number of FUs and assigned half to each of the two threads. This in effect gives each thread a full set of FUs for itself, removing any contention. The results are indicated by the third set of three bars, which are virtually identical to the prior set with only register contention removed. Since FUs are low latency resources (in terms of the number of cycles that they are occupied by a thread) in comparison to the registers and L1 Dcache blocks, contention for these resources has a low performance impact, showing little change in relative performance.

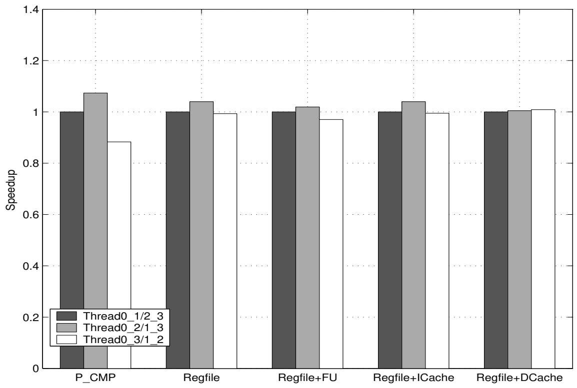

Figure 3. Performance variation breakdown for mix1.

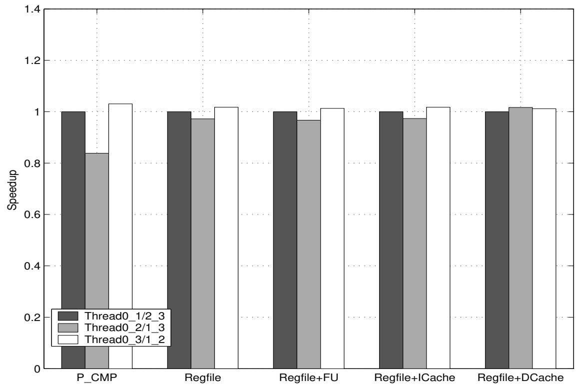

Figure 4. Performance variation breakdown for mix5.

4.3 L1 ICache Contention

The fourth set of three bars eliminates contention for L1 ICache resources as well by artificially duplicating the ICache. Due to the very small L1 ICache footprints and low miss rates of our applications, the ICache has a negligible impact on the performance variation with different thread groupings.

4.4 L1 DCache Contention

For the fifth set of three bars, the entire L1 DCache is replicated for each thread. As can be seen, any remaining differences in performance among the three pairings is eliminated. Like the register file, the L1 DCache is a long latency resource and the contention for this resource is very important. However, an additional factor that may make it less important than register files is that the contention is highly dependent on the degree to which the threads simultaneously use the same cache sets. In contrast, register files are fully associative resources that could be used by any instruction from any thread, which makes them a more significant source of contention.

In summary, functional units are used for short periods and therefore FU contention is a relatively small factor in terms of thread co-scheduling. Due to the small ICache footprints and miss rates of our benchmarks, the Icache plays no role in performance variation with our workload mixes. The register files are occupied for longer periods and therefore contention is a much more important factor. Contention in the L1 DCache is also critical but the contention may not be as frequent as with the register file.

5 Thread Phase Co-Scheduling Policies

The last section identified contention for registers and L1 Dcache sets as the two major causes of performance variation according to how threads were co-scheduled for our workloads. In this section, we introduce several candidate thread co-scheduling policies, some that directly address these sources of contention, and others that are based on an alternative approach of *thread sensitivity* to contention for these resources. In each of these approaches, per-thread statistics are gathered for each interval, and a decision made at the end of an interval on whether threads should migrate between processors, or the current grouping should be used.

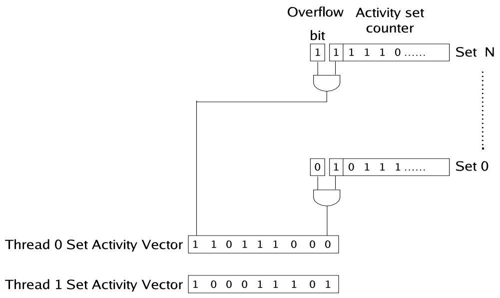

Figure 5. Cache set activity vector generation.

5.1 Policies Based on Resource Contention

5.1.1 Data Cache Conflict Scheduling (DCCS)

This policy uses L1 Dcache *conflict vectors* to co-schedule threads, similar to what is proposed in [8, 17]. As is shown in Figure 5, a counter per thread per L1 Dcache set is used to count the number of accesses during an interval period. At the end of each interval, these counters are examined, and each counter value is translated into a bit in a thread cache set *activity vector*. To identify whether the set is heavily used by a thread, the most significant bit and the overflow bit are logically ANDed to reduce each set counter to a bit in the activity vector. Each pair of thread activity vectors are logically ANDed and the produced vector per thread pair is called the *pair conflict vector*. This vector shows the cache sets that have the most contention for these two threads. With four threads and two dual-thread cores, there are six possible conflict vectors. As is shown in Figure 6,

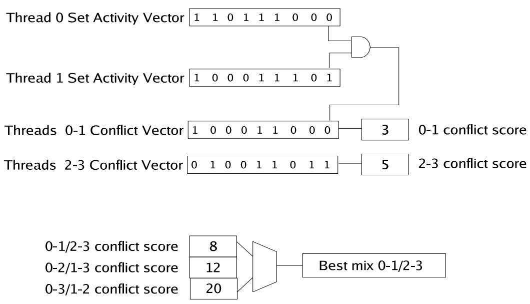

Figure 6. Combining activity vectors to form conflict scores and to determine the best pairing of the four threads for the next interval.

the number of ones in each conflict vector is determined, and these are further combined to score the three possible ways to pair the four threads on the two cores. The smallest score represents the thread grouping with the fewest cache conflicts, and this grouping is used in the next interval.

5.1.2 Register File Utilization Scheduling (RFUS)

In this policy, register file utilization is measured for each thread on an interval-to-interval basis. A per-thread register usage counter is maintained, which is incremented at rename (when a physical register is assigned to an instruction) and is decremented at commit time (when the corresponding logical register is overwritten, indicating that the previously assigned physical register is no longer in use). This counter is read every cycle and accumulated in a per-interval counter for each thread. The per-interval counters are read at the end of an interval and then reset. The values are used to schedule threads for the next interval. We found that the best policy was to co-schedule the thread with the lowest usage count with the median of the other three threads.

5.1.3 Register File Conflict Scheduling (RFCS)

Rather than register file usage, this policy is based on register file conflicts due to high register file occupation with particular thread pairings. Whenever an instruction is unable to dispatch due to the unavailability of a free register, the *conflict counter* for that thread is incremented. The counters are read at the end of an interval and then reset. As with RFUS, the best results were obtained by co-scheduling the thread with the lowest conflict count with the median of the other three threads.

5.2 An Alternative Approach: Measuring Thread Sensitivity

An alternative to directly measuring inter-thread-phase contention within the L1 Dcache and/or register file is to determine the *sensitivity* of thread phases to the need for these resources. The philosophy behind this approach is that the critical issue in resource contention between thread phases is the performance sensitivity to the availability of critical shared resources. In other words, knowledge regarding the load on the resource is less important than knowing the degree to which each thread's performance is impacted by a reduction in resource availability. This stands to reason since high ILP threads tend to use and recycle resources rapidly (and therefore their performance is more sensitive to resource availability), whereas those that, for instance, are more memory bound, are already at a low IPC point, and are impacted less by loss of resources; the resources of the machine are highly utilized but not in an efficient way.

We explore two policies that use thread phase sensitivity to resource availability: one that uses IPC as a measure of sensitivity and the other that uses ready instruction and inflight instruction metrics.

5.2.1 IPC-based Scheduling (IPCS)

This approach is inspired by the work of Parekh et al. [15] in which IPC information is used to guide thread scheduling. Per-thread hardware counters are used to measure IPC on an interval by interval basis, and used to schedule threads for the next interval. They observe that the best performance is achieved by scheduling threads with the highest IPC together. The rationale is that scheduling threads together that can use these resources effectively is the best approach, as it prevents slow moving threads that occupy many resources due to long latency instructions (primarily cache misses) from disturbing fast threads. In effect, IPC is used to determine thread sensitivity to resource contention.

We chose the simpler IPC_MIX approach from [15] since it only requires one overall performance counter (compared to G_IPC, which separately tracks integer and floating point instruction IPCs) and achieves close to the same performance.

5.2.2 Ready Inflight Ratio Scheduling (RIRS)

One drawback of the IPC-based approach is that the IPC that is measured for a given thread is highly dependent on which thread it is paired with. (This drawback also holds for the two policies based on register file contention.) This is shown in Figures 7 and 8, which plot the IPCs of the individual benchmarks for the different pairings. In almost all workloads, the highest performing benchmarks change dramatically depending on how the threads are paired. This can lead to significantly different scheduling decisions depending on how the benchmarks are paired when the IPC measurements are obtained.

Based on these results, we seek a metric that, like IPC, provides a measure of thread sensitivity, yet whose relative

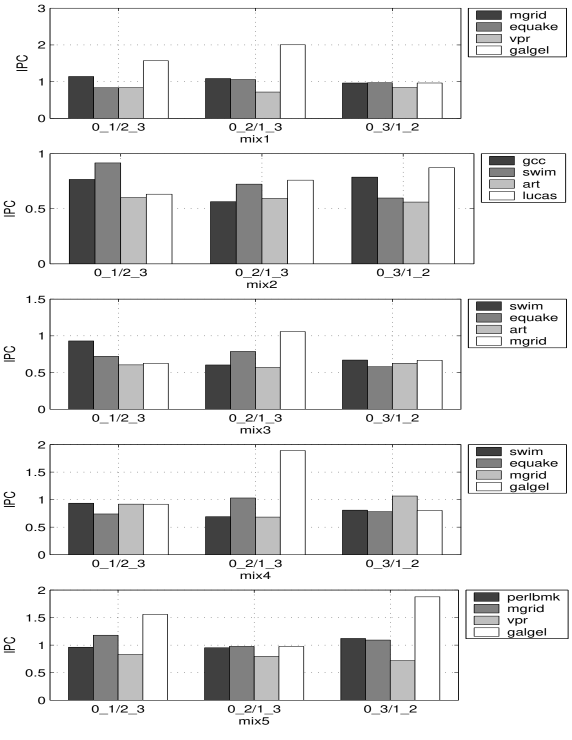

Figure 7. IPC metric variation for each benchmark with different groupings (mixes 1-5).

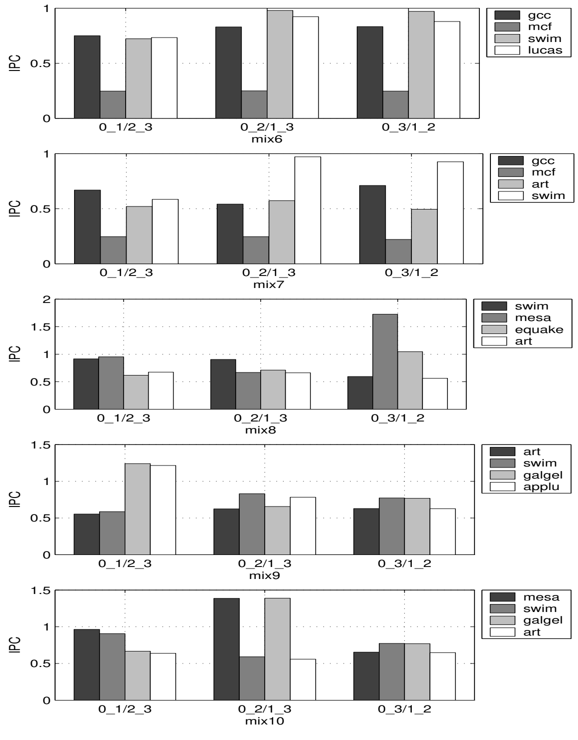

Figure 8. IPC metric variation for each benchmark with different groupings (mixes 6-10).

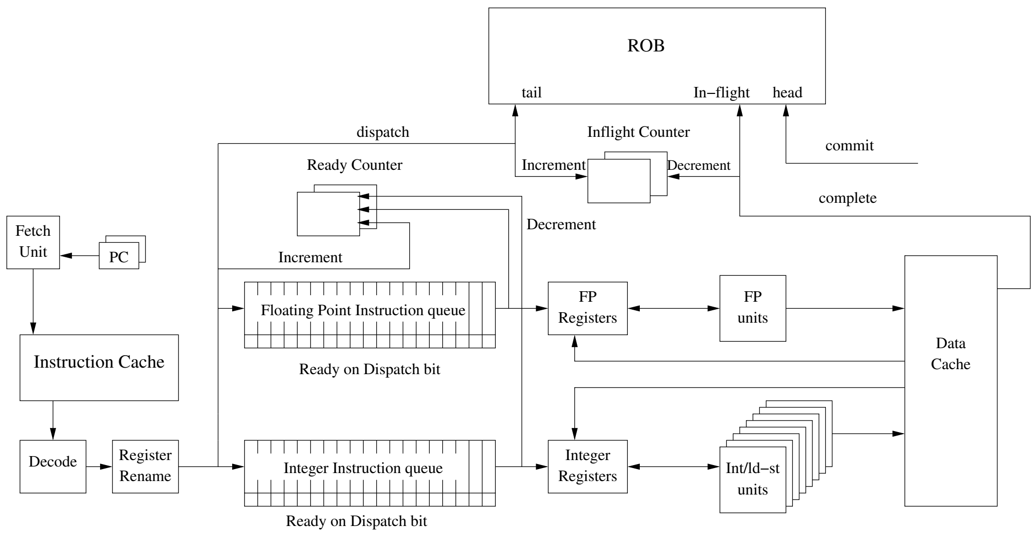

Figure 9. Counters used in the RIRS scheme.

rankings among the threads in a workload remains consistent independent of the degree of contention with another thread. We have found that the ratio between the number of ready instructions in the issue queues, and the number of in-flight instructions from the issue to the writeback stages (which we call the *Ready to In-flight Ratio*, or RIR) serves as an effective sensitivity measure that exhibits consistent results compared with IPC across different thread pairings. Intuitively, a high ratio of ready to in-flight instructions means that the thread is able to make good progress with whatever resources it has available, and that depriving it of those resources will non-trivially degrade its performance. Thus, a high value for this ratio indicates high sensitivity for resources. If the thread is paired with another for which it

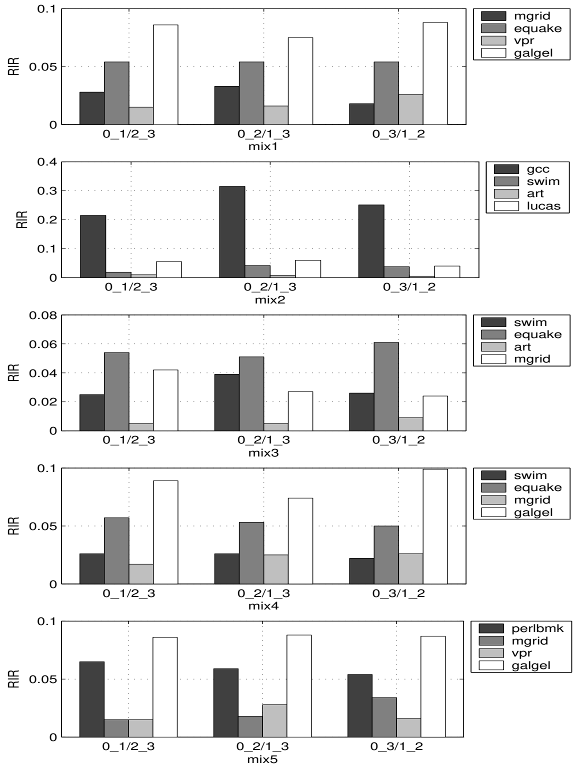

Figure 10. RIR metric variation for each benchmark with different groupings (mixes 1-5).

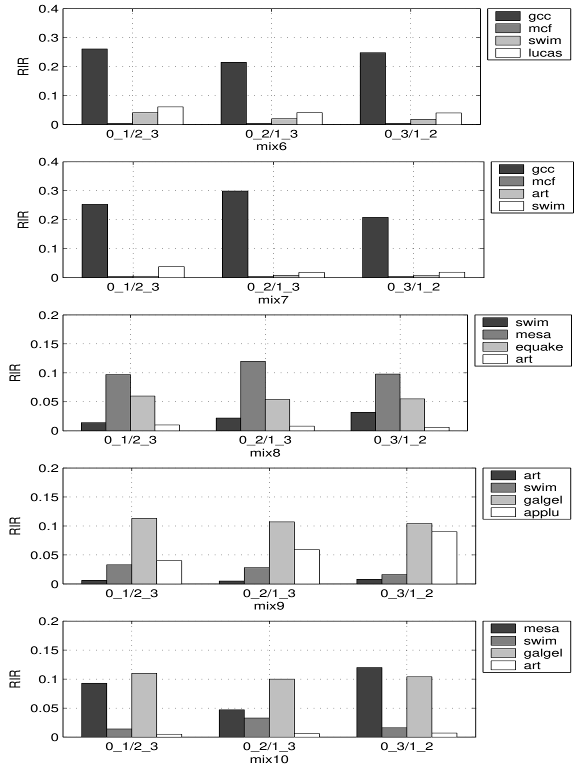

Figure 11. RIR metric variation for each benchmark with different groupings (mixes 6-10).

contends for resources, then although the number of ready instructions and the number of in-flight instructions may change, the RIR metric should stay much more constant than the IPC (which may drop severely).

The integration of the RIR counters within an SMT pipeline is shown in Figure 9. One of the issues with the approach is that direct measurement of the number of ready instructions in the queues on a cycle-by-cycle basis is complex. To avoid this, we use the approximate approach of [4]. Here, the *Ready Counter* is incremented whenever an instruction is dispatched into the queue with both of its source operands available. In this case, the *Ready on Dispatch* bit is set in the queue for this instruction. An instruction that issues with its Ready on Dispatch bit set decrements the counter. The *In-flight Counter* measures the number of instructions from the issue queue to the write back stage. Once the RIR ratios are computed within an interval, we schedule the two threads with the highest RIR ratio together. We found that, like the IPC_MIX policy, this approach outperformed alternatives such as pairing the threads with the highest and lowest RIR ratios.

Evidence of the relative invariability of the RIR metric is shown for the ten workload mixes in Figures 10 and 11. Compared to the IPC (Figures 7 and 8), RIR provides much better results in terms of consistency across the different applications with different pairings. For nine of the ten workloads, the applications with the top two highest RIR values do not change with the application grouping.

6 Results

In this section, we first compare the results of the different scheduling policies using a fine-grain interval (100K cycles). We work at this fine grain to provide maximum opportunity for the dynamic schemes to improve the performance of the static groupings. However, this granularity requires low level hardware control. Therefore, in Section 6.2, we examine the use of a coarser grain interval on the order of an OS time slice (100M cycles). At this granularity, it becomes feasible for the OS to read the hardware counters at the end of every interval and make co-scheduling decisions based on the policies outlined in the prior section.

6.1 Fine-Grain Thread Co-Scheduling Policies

Figure 12 presents the performance of the various dynamic co-scheduling schemes (RFUS [register file utilization scheduling], RFCS [register file conflict scheduling], DCCS [data cache conflict scheduling], IPCS [IPC-based scheduling], and RIRS [ready inflight ratio scheduling]) relative to the best of the three static groupings of threads for

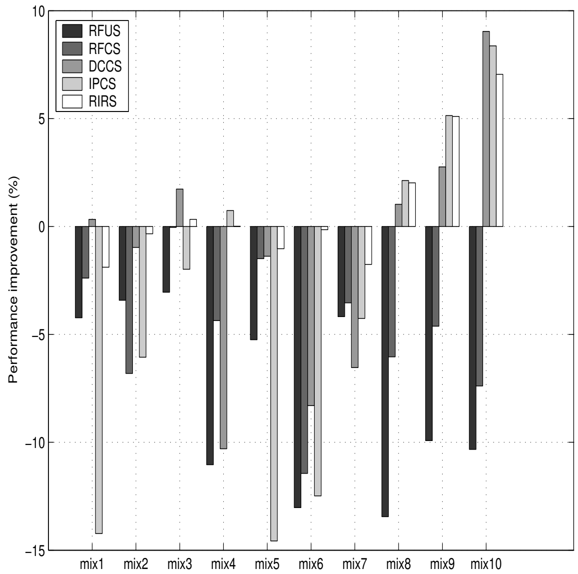

Figure 12. Performance of the different scheduling policies relative to the best static thread grouping.

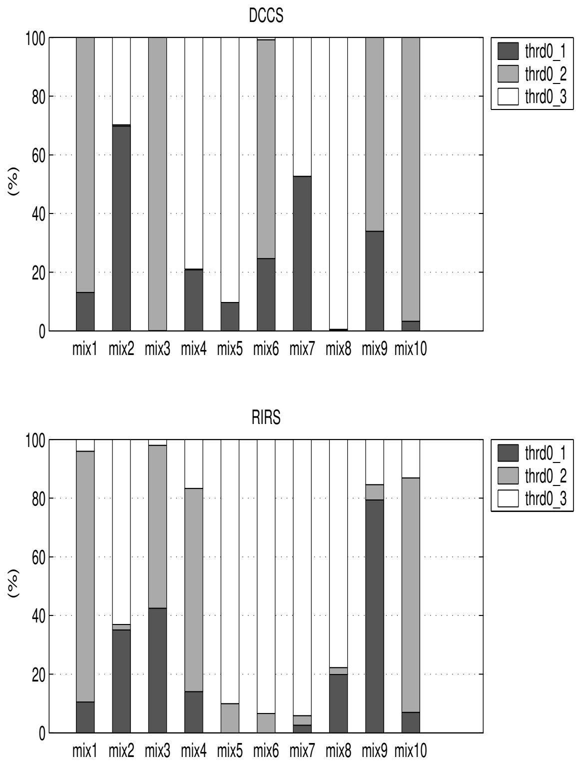

Figure 13. Percentage of time thread 0 shares a core with the other three threads for DCCS and RIRS.

each individual workload mix. This baseline assumes that the best grouping is known *a priori* and is used throughout the execution of the four threads. The results show the percent difference in geometric mean performance compared to this best static baseline. Of the dynamic schemes, the RIRS scheme that uses the RIR metric performs most consistently across the ten workloads. While the maximum performance loss using the RIR metric is about 2%, it reaches about 10% for DCCS, 13% for RFUS, 11% for RFCS, and 14% for IPCS. In several cases, RIRS exceeds the performance of the best static scheme (by as much as 7%), demonstrating the benefit of performing dynamic phase-level regrouping through thread migration.

Figure 13 shows the percentage of time thread 0 shares the same core with each of the remaining three threads. While the optimal grouping varies somewhat on a phase-to-phase basis, the overall results nonetheless shed some light on the differences in the approaches. For example, Figure 2 demonstrated that for mix4, it is best overall for threads 0 and 2 to share a processor. Indeed, Figure 13 demonstrates that RIRS uses this thread grouping for the majority of the execution of mix4. With DCCS however, this thread grouping is never selected, and the result is a large degradation in performance.

There are a few instances for which RIRS sometimes picks a suboptimal choice of thread groupings, for example, with mix3. The cache conflicts metric used by DCCS shows more stability and DCCS always schedules threads 0 and 2 together, which results in the best performance. The RIR metric, on the other hand, incorporates both dependence and resource contention (cache, register, and functional unit conflicts) information into its metric, and therefore shows more variation. It switches between pairings more often because the performance of the pairings do not differ significantly, resulting in a slight degradation in performance relative to DCCS because cache conflicts are the slightly dominant effect. However, the difference in performance between the two schemes in any of these cases is $\leq 2\%$.

6.2 Fine Grain Versus Coarse Grain Intervals

A granularity of 100K cycles requires hardware level control. However, such a hardware mechanism may interfere with OS-level scheduling decisions. Therefore, it would be desirable if these techniques could also work well at the coarser granularity of an operating system time slice (10-30 ms). In this section, we compare the effectiveness of the RIRS scheme at fine grain (100K cycles) and coarse grain (100M cycles) intervals.

Table 3 shows the performance of the coarse-grain scheme relative to the fine-grain approach for each workload. While there are small differences from workload to workload, overall the coarse and fine-grain schemes achieve very similar performance. Thus, co-scheduling decisions

Table 3. Performance of the coarse-grain RIRS scheme relative to the fine-grain approach.

Workload	Performance improvement(%)
mix1	-0.02
mix2	-0.73
mix3	0.54
mix4	2.80
mix5	1.14
mix6	-0.57
mix7	-0.86
mix8	-1.13
mix9	2.11
mix10	-1.44

can be made at the OS scheduler level given access to the RIR hardware counters.

7 Conclusions

The advent of CMPs of SMT processors for server environments places a premium on determining the assignment of threads to processors that will maximize overall throughput. Achieving this goal requires online mechanisms that gather per-thread statistics that both capture the essential thread characteristics necessary for making scheduling decisions, and that remain relatively invariant no matter how the threads are paired when the statistics are gathered.

In this paper, we examined a number of potential mechanisms, and determined that per-thread measurements of the ready to in-flight instruction ratio (RIR) permits effective co-scheduling decisions to be made. This approach achieves consistently competitive performance with the best *a priori* static scheme, and even achieves up to a 7% performance improvement due to its ability to make dynamic runtime adjustments. We also found that the scheme is effective at a coarser granularity that would permit the use of the RIR metric within the operating system scheduler.

We plan to explore schemes for parallel applications and more aggressive CMP-SMT configurations in the future.

References

[1] D. Burger and T. Austin. The Simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[2] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernández. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 171–182, December 2005.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, February 2005.

[4] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 31–40, February 2003.

[5] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *Proceedings of USENIX 2005 Annual Technical Conference*, pages 395–398, June 2005.

[6] Intel Corporation. IA-32 Intel Architecture Optimization: Reference Manual. *http://www.intel.com/design/pentium4/manuals*, 2004.

[7] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[8] J. L. Kihm and D. A. Connors. Implementation of Fine-Grained Cache Monitoring for Improved SMT Scheduling. In *Proceedings of the 22nd IEEE International Conference on Computer Design*, pages 326–331, October 2004.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March 2005.

[10] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[11] K. Luo, J. Gummaraju, and M. Franklin. Balancing Thoughput and Fairness in SMT Processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, January 2001.

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[13] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March 2005.

[14] R. Merritt. IBM Weaves Multithreading into Power5. *EE Times*, 2003.

[15] S. Parekh, S. Eggers, and H. Levy. Thread-Sensitive Scheduling for SMT Processors. Technical report, University of Washington, 2000.

[16] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference of Parallel Architectures and Compilation Techniques*, pages 15–26, September 2003.

[17] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–73, October 2004.

[18] A. Snavely and D. M. Tullsen. Explorations in Symbiosis on Two Multithreaded Architectures. In *Proceedings of the Workshop on Multithreaded Execution, Architecture, and Compilation*, January 1999.

[19] A. Snavely and D. M. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Architecture. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, November 2000.

[20] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.

[21] D. Tullsen, S. Eggers, H. Levy, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. 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, May 1996.