Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs*

Yun Zhang and Michael Voss
(yzhang,voss)@eecg.toronto.edu
Edward S. Rogers Sr. Department of Electrical
and Computer Engineering, University of Toronto
10 King's College Road, Toronto, Ontario, Canada M5S 3G4
phone: 416-946-8031 fax: 416-971-2326

Abstract

Hyperthreaded (HT) and simultaneous multithreaded (SMT) processors are now available in commodity workstations and servers. This technology is designed to increase throughput by executing multiple concurrent threads on a single physical processor. These multiple threads share the processor's functional units and on-chip memory hierarchy in an attempt to make better use of idle resources. Most OpenMP applications have been written assuming an Symmetric Multiprocessor (SMP), not an SMT, model. Threads executing on the same physical processor have interactions on data locality and resource sharing that do not occur on traditional SMPs. This work focuses on tuning the behavior of OpenMP applications executing on SMPs with SMT processors. We propose two adaptive loop schedulers that determine effective hierarchical schedulers for individual parallel loops. We compare the performance of our two proposed schedulers against several standard schedulers and the per-region adaptive scheduler proposed by Zhang et al. using the SPEC and NAS OpenMP benchmark suites. We show that both of our proposed schedulers outperform all other schedulers on average, and increase speedup on average by over 25% when all thread contexts are used.

1. Introduction

Hyperthreaded (HT) [5] and simultaneous multithreaded (SMT) [10] processors are now available in commodity systems[1]. SMTs allow multiple threads to execute concurrently on a single physical processor. In this work, we focus on tuning the behavior of OpenMP applications executing on Hyperthreaded SMPs. OpenMP and SMT processors represent emerging entry-points for new parallel programmers. It is therefore important for the success of both technologies that they work well together.

Unfortunately, understanding and controlling the performance of OpenMP applications on SMT processors is non-trivial. To understand their combined performance, three application characteristics must be considered: (1) inter-thread data locality, (2) instruction mix and (3) SMT-related load imbalance.

The choice of threads to be co-located on a processor is important to exploit inter-thread data locality. Data that has been recently accessed by a thread can be accessed more quickly by a co-located thread (a thread executing on the same physical processor) than by a non-co-located thread. Likewise, the instruction mix of each thread affects combined performance, since they compete for functional units. Finally, most OpenMP applications have been written assuming an SMP model. Given the unique interaction of data locality and instruction mix on program performance, codes executing on a Hyperthreaded SMP may show load imbalances for loops that are well balanced on a true SMP.

To address these issues, Zhang et al. [13] proposed a self-tuning OpenMP loop scheduler. We refer to the scheduler from [13] as a Region-based adaptive scheduler (RBS). RBS is designed to react to behavior caused by the three important application characteristics outlined above, automatically selecting the number of threads and scheduling policy that should be used for each parallel region.

However, RBS was designed under the assumption that parallel regions contain a single parallel loop. In [13], RBS was evaluated on the SPEC OpenMP benchmarks, where this assumption often holds. RBS was shown to outperform a wide range of traditional single-level non-adaptive schedulers. In this paper, we show that RBS does not perform well on benchmarks that break this assumption. To address this issue, we propose two alternative adaptive sched-

* This work was supported in part by the Canadian National Science and Engineering Research Council, the Canada Foundation for Innovation, the Ontario Innovation Trust, the Connaught Foundation and the University of Toronto.

1 We shall use SMT and HT interchangeably in this paper

ulers: (1) a Loop-based Scheduler (LBS) that makes decisions at the granularity of individual parallel loops and (2) a Hardware-Counter Directed Scheduler (HCS) that uses hardware counters to sample performance of parallel loops and quickly decide on a good scheduler using a decision tree that is created off-line. We show that both the LBS and HCS outperform a number of single-level non-adaptive schedulers as well as RBS for a range of benchmarks.

2. Related Work

The OpenMP API The OpenMP API has become the industry standard for loop-level shared-memory parallel programming [2, 9, 8]. In this work, we extend the Omni OpenMP research compiler [7]. The OpenMP API supports general parallel regions, parallel sections and parallel loops. Figure 1 shows an example of a parallel OpenMP loop in C.

```
#pragma omp parallel
#pragma for shared(a,b) private(i,j) \
            schedule(runtime)
for (i = 0; i < 100; i++) {
 for (j = 0; j < 100; j++) {
   a[i][j] = a[i][j] + b[i][j];
 }
}
```

Figure 1. An example of a parallel loop in C. The `schedule(runtime)` clause specifies that the scheduler will be selected by the user at runtime through an environment variable.

Adaptive Loop Scheduling Properly selecting the scheduling policy for parallel loops can result in large performance gains. A number of groups have investigated runtime techniques for selecting loop schedules to improve performance. In [1], an adaptive scheduler is designed for the IBM XLF OpenMP compiler that derives the best scheduling policy for each parallel loop at runtime. In [1], the target system is an SMP and no decisions are made to reduce the number of threads used by the loops.

In [3], a system is proposed that adaptively adjusts the number of threads assigned to applications to increase the throughput of a multiprogram workload on an SMP. In contrast to [3], our work focuses on the speed of applications on a dedicated system. In [12], parallel loops are dynamically serialized to avoid overheads that cannot be amortized by parallel execution. Unlike our work, the work in [12] finds loops that have sufficient work to amortize parallelization overheads. The work presented here proposes a general self-tuning loop scheduler.

Zhang et al. [13] propose an adaptive hierarchical scheduler that targets SMPs builts from SMT nodes. Their scheduler, referred to as Region-based Scheduler (RBS) in this paper, makes decisions at the granularity of parallel regions. This paper extends the work in [13]. Our evaluation is also performed on both the SPEC and NAS OpenMP benchmarks suites, whereas in [13] only the SPEC benchmarks were evaluated.

3. Standard and Novel Loop Schedulers

In OpenMP, a parallel loop can be specified using an `omp for` construct in a C/C++ program and an `omp do` directive in a Fortran program. Figure 1 shows an example of a parallel loop written in C and Fortran using the OpenMP API. While the example shows only a single loop in the parallel region, there may be more than one parallel loop per parallel region. The OpenMP compiler converts the parallel regions into thread-based code with calls to the OpenMP runtime library to perform synchronization and scheduling of the parallel loops, as shown in Figure 2.

```
_ompc_runtime_sched_init (_p_i_0, _p_i_1,
                          _p_i_2);
while (_ompc_runtime_sched_next (&_p_i_0,
                                 &_p_i_1)) {
 for (i = _p_i_0;
      i < _p_i_1;
      i += _p_i_2) {
   for (j = 0; j < (100; j++) {
     a[i][j] = a[i][j] + b[i][j];
   }
 }
}
```

Figure 2. A loop with a schedule(runtime) pragma as transformed by the Omni compiler.

The region shown in Figure 2 has been outlined into a subroutine by the Omni compiler. At the location in the code where the region is to be executed, a call to the Omni scheduler (`_ompc_do_parallel`) is made with a pointer to the outlined function as an argument. The runtime system creates a team of threads to execute the region and passes each of them a copy of the function pointer. As each thread executes a parallel loop in the outlined code, they make a call to `_ompc_runtime_sched_init` which initializes the user-selected scheduler for that loop. In each iteration of the while loop, the call to `_ompc_runtime_sched_next` returns a chunk of iterations for the thread to perform.

Each thread continues to execute the while loop until `_ompc_runtime_sched_next` returns zero.

3.1. Single-level Schedulers

In this paper, we evaluate several different adaptive scheduling methods and their effect on the performance of applications executed on a Hyperthreaded SMP. Table 1 describes the traditional single-level schedulers we compared our techniques to, including the standard OpenMP loop schedulers (static, dynamic and guided), as well as two advanced schedulers from the literature (affinity and trapezoidal self-scheduling). These schedulers were designed to target symmetric multiprocessors. As described in Table 1, each scheduler was designed to improve the performance of particular classes of loops, i.e. loops with poor load balance or specific data locality patterns.

As will be demonstrated in Section 4, none of the loop schedulers described in Table 1 effectively react to OpenMP applications executing on SMPs with SMT nodes. In the next sections, we describe three self-tuning algorithms that are designed to exploit the unique features of SMPs built from SMT nodes.

3.2. RBS Adaptive Hierarchical Scheduler

Zhang et al. [13] proposed an advanced self-tuning scheduler tailored specifically for this domain, which we shall refer to as the Region-based Scheduler (RBS). We will first provide an overview of the RBS and then present our extensions of this approach.

As shown in Table 2, most execution time in the SPEC and NAS benchmarks is spent in parallel loops that are executed more than 40 times. RBS leverages this behavior, by using the runtime history of a loop to better select a scheduler for it. A pseudo-code description of RBS is found in Figure 3 and is described in detail in this section.

RBS uses a two-level hierarchical scheduler as shown in Figure 4. Two-level scheduling is based on the observation that not all processors in a Hyperthreaded SMP are equal: the virtual processors and their physical siblings share the same cache and functional resources. The hierarchical scheduler exploits this aspect by grouping the processors into nodes (e.g., on our 2-way Hyperthreaded 4-processor system, we have 4 nodes of 2 processors each). The scheduler assigns iterations to nodes by using what we will hereafter refer to as "the upper algorithm", and then at each node the iterations will be further distributed between the two siblings using a "lower algorithm" [2].

Since it is difficult to make an *a priori* decision about which scheduler to use for any given region, RBS is also adaptive. It samples the performance of a number of scheduling methods at runtime to determine which scheduler performs the best for each region. RBS samples all of the schedulers described in Table 1. It begins by searching for a best upper-level algorithm. Only 1 thread is used per physical processor during this Upper-level search phase. At each invocation of each parallel region, a different scheduling method is used for all of the parallel loops within the region and the execution time is collected. After all scheduling methods have been sampled for a given region, the algorithm with the smallest execution time is selected as best.

Benchmark	< 10 times	$10 \leq$ times ≤ 40	> 40 times
ammp	0%	0%	84.20%
apsi	0%	0%	82.55%
art	100%	0%	0%
equake	0.05%	0%	98.23%
mgrid	0%	0.11%	95.95%
swim	0.09%	0%	99.25%
wupwise	0.12%	0%	99.49%
bt.W	0%	0%	100%
cg.A	0.92%	3.5%	92.57%
ep.A	100%	0%	0%
mg.A	12.73%	12.87%	71.91%
sp.W	1.02%	0%	92.71%

Table 2. Number of calls to parallel loops and their percentage of total execution time

At each invocation, the complexity of the region, as calculated from the loop bounds of the contained parallel loops, is also passed to the runtime system. If the work varies between invocations, the sampling will be inaccurate and the system bails out, using affinity scheduling with 4 threads for all subsequent invocations of the region.

After the Upper-level search phase is complete, RBS has selected an upper-level algorithm that has the best performance for that region. This choice will be fixed as the upper algorithm. The scheduler will then enter a Lower-level search phase (see Figure 3). During this phase it will use 2 threads per SMT processor.

It begins by sampling the per-thread execution times for this region when using the static scheduler as the lower algorithm for all parallel loops in the region. If per-thread timings indicate that no load imbalance is seen between sibling threads across all loops, there is no need to examine other schedulers, and static will be asserted as the best lower-level choice. If during the Lower-level search phase, per-thread execution times exhibited by the static scheduler show an imbalance, other load balancing scheduling algorithms must be sampled.

The execution time obtained by the best performing two-level algorithm (using 2 threads per node) will be compared

2 While our system provides only 2 threads per physical processor, this scheme can easily be generalized for nodes with T threads.

Algorithm	Description
Static	The static OpenMP scheduler divides the iterations of a loop among the threads by handing out chunks of N iterations in a round-robin fashion. The size of the chunks can be statically set by the user by explicitly specifying an `n` in the `schedule` directive. If no chunk size is provided, the iterations are divided into P evenly sized contiguous chunks, where P is the number of processors. Since the schedule can be statically determined, this method has the least runtime overhead.
Dynamic	The dynamic scheduler divides the iterations among the threads by handing out chunks of `n` iterations on a first-come, first-served basis. If no chunk size is specified, a single iteration is provided. In this paper, we will assume that dynamic scheduling always uses a chunk size of 1. Dynamic schedulers are used for computations that have a load imbalance if distributed statically.
Guided	The guided scheduler works in a fashion similar to the dynamic scheduler, except that the size of the assigned chunks decreases exponentially (to `n`, if one is specified, and to 1 otherwise). Each time a new chunk is assigned, its size is approximately the number of remaining (i.e., unassigned) iterations divided by the number of threads. The chunk sizes in guided scheduling begin large and slowly decrease in size, resulting in fewer synchronizations than with dynamic scheduling, while still providing load balancing.
Affinity	The affinity-based scheduler [4] addresses the problem of the significant communication overhead incurred by addressing non-local data on shared-memory multiprocessors. It uses work queues for each processor, to which it statically distributes the iterations of the loop. Each processor retrieves only a small fraction of these iterations at a time; when a processor's queue is empty, it removes a fraction of the iterations from the most loaded processor's queue. Load imbalance is addressed by dynamic reassignment of iterations from the most loaded processors to the idle ones. This scheduling algorithm is particularly efficient when the same data is used repeatedly by the same processor during different invocations of a parallel loop.
Trapezoidal	Trapezoid self-scheduling[11], like both dynamic and guided scheduling, is designed to distribute work more evenly to threads by doing runtime load balancing. TSS provides a linearly decreasing number of iterations per request. It is argued in [11] that this linear function is faster to compute than the exponential function used by guided, and that the decreasing chunk size will still yield substantially fewer synchronizations than dynamic scheduling.

```
BEGIN Upper-level search phase
  deactivate lower-level sched
  activate only 1 thread per SMT node
  FOR all sched s
    IF workload varies THEN bailout ENDIF
    sample performance using sched s
  ENDFOR
  SET upper-level to best performing sched
  SET T1 to execution time with best sched
END Upper-level search phase

BEGIN Lower-level search phase
  activate 2 threads per SMT node
  activate the lower-level sched
  SET lower-level sched to static
  sample 2-level best-upper-alg/static sched
  IF per-thread times show imbalance across
  siblings
     FOR all sched s
        sample performance of best-upper-alg/s
        if per-thread times are now balanced,
        end loop
     ENDFOR
  ENDIF
  SET T2 to execution time with best 2-level
  SET lower-level sched to best 2-level sched
  IF T1 < T2
    deactivate lower-level sched
    activate only 1 thread per SMT node
  ENDIF
END Lower-level search phase
```

Figure 3. The adaptive hierarchical scheduling algorithm used by RBS and LBS.

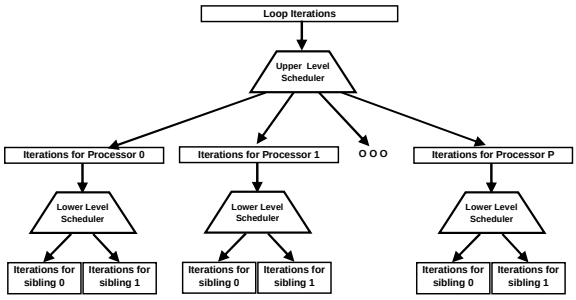


Figure 4. The structure of the hierarchical scheduler used by RBS, LBS and HCS.

against the execution time of the best upper-only execution time (using 1 thread per node). If the 1-thread-per-SMT version performs better, the second thread on each SMT will be disabled for subsequent executions of this region, and the previously determined best upper-algorithm will be used to schedule iterations across the physical processors only. Otherwise, the best two-level scheduler will be used for all subsequent executions.

Since the default loop scheduler used by an OpenMP runtime library is implementation dependent [9, 8], loops that do not include a `schedule` pragma cannot assume that any particular thread will execute any of its iterations. However, all threads must (and will) execute code not found within work-sharing constructs (such as parallel loops). Therefore 8 threads are used to execute all code that falls outside of parallel loops but within the parallel region, ensuring correct execution.

To reduce the complexity of the required book keeping, our implementation allows only 1 parallel loop to be active at a time. This requires the addition of a barrier at the beginning of `_ompc_runtime_sched_init`. In OpenMP, there are implicit barriers at the end of parallel loops that can be removed by the use of a `NOWAIT` pragma. Due to the added barrier in our implementation of both RBS, two consecutive parallel loops will always be separated by a barrier even if a `NOWAIT` pragma is specified by the user. However, since the barrier occurs in `_ompc_runtime_sched_init`, it is a subsequent paral-

lel loop that includes the barrier. Therefore a NOWAIT loop will continue to execute code after the parallel loop until a subsequent parallel loop is encountered.

In [13], RBS was evaluated across a subset of the SPEC OpenMP Benchmarks and was shown to outperform all other single-level schedulers. It should be noted that in Table 3, the parallel regions in the SPEC benchmarks often have only 1 parallel loop per region. Only wupwise has a region with more than 1 parallel loop. Therefore the assumption made by RBS that the scheduler can be set per region is acceptable for these benchmarks. However, it is clear for other applications, such as those in the NAS OpenMP benchmark suite, this assumption does not hold.

3.3. LBS Adaptive Hierarchical Scheduler

As shown in Table 3, there are often multiple parallel loops per region. To address this, we propose a Loop-based (LBS) implementation of Figure 3. The algorithm remains unchanged, except that instead of making decisions at the granularity of a parallel region, decisions are made at the granularity of each parallel loop.

To reduce the complexity of the required book keeping, our implementation of LBS contains the same additional barrier used by RBS in _ompc_runtime_sched_init. However, dropping the 1-loop-per-region assumption complicates the design of the system and adds the need for additional synchronizations. In RBS, timestamps are taken by each thread only at the beginning and end of each parallel region. In LBS, timestamps must be collected at each call to _ompc_runtime_sched_init and at the last call by each thread to _ompc_runtime_sched_next for the loop being sampled. Therefore, during the initial sampling phase, an additional barrier is enforced at the end of each parallel loop so that accurate timings can be made. This additional end-of-loop barrier can be removed for NOWAIT loops after a final decision on the configuration for this loop has been made. The barrier in _ompc_runtime_sched_init however remains.

LBS allows decisions to be made at the loop level; however it requires more instrumentation (timestamps at each loop) as well as forces additional barriers between parallel loops in the same region even if NOWAIT has been specified [3].

3.4. HCS Scheduler

Our Hardware-Counter Directed Scheduler (HCS) short-circuits the sampling phases of Figure 3 by choosing the runtime scheduler according to the characteristics of a loop, such as its cache miss rate, number of floating point operations, load imbalance etc. HCS measures metrics during a single invocation of a loop and uses a decision tree, created off-line, to immediately select the appropriate hierarchical scheduler. The decision tree is created automatically in an off-line step that profiles benchmarks using the various schedulers and feeds the profile data to clustering software to generate the selection rules. This step creates a single tree that is used across all benchmarks. Figure 5 shows the modified algorithm used by HCS.

```
BEGIN HCS scheduling

  activate only 1 thread per SMT node
  execute the loop in the first warm-up run

  activate 2 threads per SMT node
  SET upper-level scheduler to Static
  SET lowerer-level scheduler to Static

  collect number of micro-operations
          number of load/store instructions
          number of floating point operations
          number of cache misses
          in this run

  apply decision tree to make a decision for
  subsequent executions
```

Figure 5. The modified adaptive loop scheduler algorithm used by the HCS scheduler.

To generate the decision tree used in Figure 5, training data is collected from a subset of the benchmarks in Table 3. The data collected for each parallel loop includes: the number of micro-operations executed, the number of load-store operations, the number of floating point operations, the number of cache misses and the load imbalance (the difference between the fastest thread and slowest thread). The training phase executes each loop with all possible scheduling configurations, noting which configuration shows the best execution time.

The data collected during the training phase is used by the C4.5 [6] classification software to automatically generate a decision tree. Each tree node corresponds to a simple question based on the collected metrics, with its children corresponding to alternative answers. Leaf nodes nodes correspond to a specific scheduling decision. The decision tree generated by C4.5 is currently integrated by hand into the implementation of our Hardware-Counter Scheduler. This could easily be done automatically and deployed as an install-time empirical optimization system.

At runtime, HCS begins by executing each loop with 1 thread per SMT processor, using a static scheduling policy to warm up the instruction and data caches. During the second invocation of the loop, it uses 2 threads per node and

3 We are currently investigating methods for removing these extra barriers in both RBS and LBS

Name	Source	Lines	Regions with 1 loop	Regions with > 1 loop	Parallel Loops	Modified Loops	Time on 1 CPU (sec)
ammp	SpecOMP	14688	5	0	10	9	364
apsi	SpecOMP	7744	24	0	24	24	378
art	SpecOMP	1917	3	0	3	2	214
equake	SpecOMP	1622	11	0	11	10	231
mgrid	SpecOMP	683	11	0	11	11	740
swim	SpecOMP	462	8	0	8	8	514
wupwise	SpecOMP	2506	6	2	10	10	1189
bt	NAS2004	3731	6	4	14	14	1907
cg	NAS2004	1106	4	5	18	18	14.98
ep	NAS2004	291	3	0	3	3	80.77
mg	NAS2004	1446	8	3	16	13	21.41
sp	NAS2004	3203	9	4	29	29	50.84

samples the same characteristics collected during the training phase (i.e. number of cache misses, etc...). This data is used to traverse the decision tree and predict the scheduler best suited to the loop. At each subsequent execution of the loop, this predicted best scheduler is used. As with the RBS and LBS approaches, the final scheduler can use 4 or 8 threads, and any hierarchical combination of the schedulers described in Table 1.

The HCS approach uses the hierarchical scheduler used by RBS and LBS, but attempts to remove the overheads incurred by the sampling of sub-optimal variants. It short circuits the sampling of the RBS and LBS techniques by using a decision tree to quickly determine the best applicable scheduler after only a single execution of the region.

4. Experimental Evaluation

4.1. Methodology

We evaluate the performance of our scheduling methods using a 4-processor Hyperthreaded Xeon server. The server has four 2.8 GHz Hyperthreaded Xeon processors and a 16 GB main memory. Each processor has a 512 KB L2 data cache and a 1 MB L3 data cache. The system runs Redhat Linux 7.3 with a slightly modified version of the 2.4.18-smp kernel [4]. In all of our experiments, we use explicit binding to ensure that threads are evenly distributed among the physical processors. We investigate the performance of the 13 benchmark programs from the SpecOMP2001 and NAS2004 benchmark suite shows in Table 3. [5]

To allow us to perform our experiments, runtime scheduling directives were added to all of the major parallel loops in these benchmarks. Loops that explicitly specified schedulers using a `schedule` clause were not modified. As shown in Table 3, the large majority of loops in these programs have no schedule clause in the original code.

4.2. Benchmark Scaling on a Hyperthreaded SMP

We first compiled our benchmark suite with the Omni research compiler (version 1.4a). Figure 6 shows the speedup of each of the original unmodified applications when executed on 1 through 8 threads. It is important to note that when only the physical processors are used, performance increases with the number of threads. When 1 or more of the sibling threads are active, performance no longer scales. For many benchmarks, using only the physical processors leads to better performance. The average speedup on 4 processors is 2.78. On 8 virtual processors, the Omni compiler shows an average speedup of 2.31 across the benchmarks. It is clear that using the extra thread on each processor is rarely of benefit.

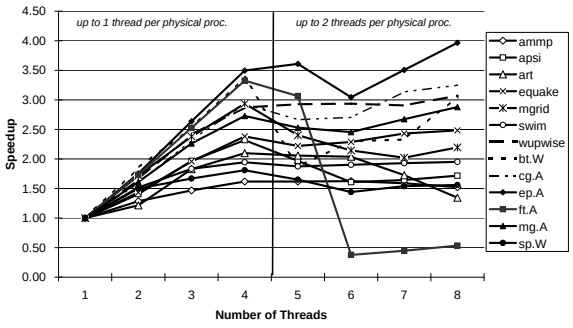


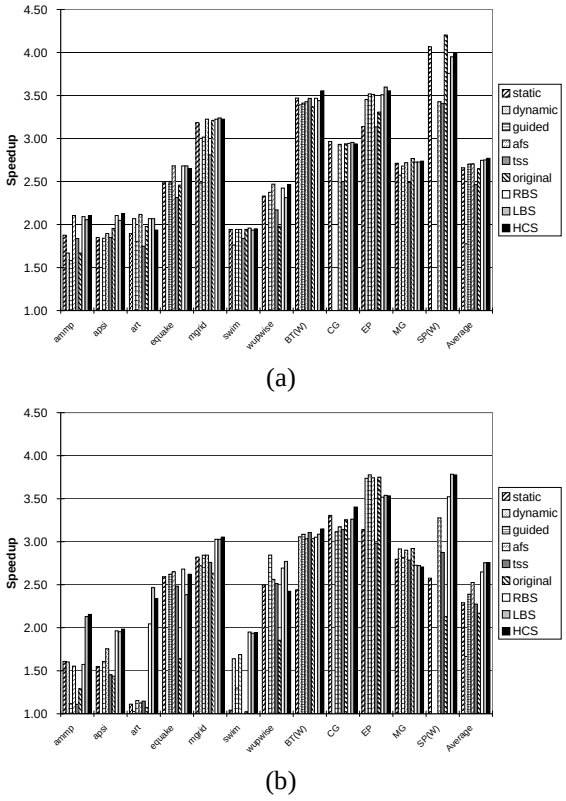
Figure 6. The speedup on 1 through 8 processors using the Omni research compiler with the original parallel applications.

4 We added a system call to allow threads to be bound to processors

5 For SPEC, we evaluated only the C and Fortran 77 benchmarks since Omni does not currently support Fortran 90, therefore gafort, galgel and fma3d are not included. In addition there is a known bug in the Omni runtime library which precludes the execution of applu. For NAS, ft, is, lu and ua likewise did not validate with Omni and were therefore not included in our study.

4.3. Single-Level Non-Adaptive Schedulers

The speedups for the original applications are poor as shown in Figure 6. To investigate the effect of more advanced schedulers on this performance, we executed the benchmarks with 4 and 8 threads using several single-level non-adaptive scheduling algorithms: static, dynamic, guided, affinity (afs) and trapezoidal self-scheduling (tss). Since affinity and tss are not available in standard OpenMP compilers, we modified Omni to support these algorithms. Figure 7 shows the results of these scheduling methods (as well as RBS, LBS and HCS) applied to these benchmarks when executed on 4 and 8 processors.



(a)

(b)

Figure 7. The speedup of applications using different schedulers when (a) only the 4 physical processors are used and (b) when all 8 virtual processors are used.

As shown in Figure 7(a), static shows a speedup of 2.66 on average across the benchmarks. The dynamic scheduler, performed very poorly on **apsi**, **equake**, **wupwise** and **cg**, resulting in an average speedup of only 1.87. Guided, which still performs runtime load balancing but with a lower overhead than dynamic, had an average speedup of 2.71. The affinity scheduler (afs) shows large improvements in most benchmarks and has an average speedup of 2.74.

When 8 threads are used, as shown in Figure 7(b), significant decreases in speedups can be seen for a number of benchmarks. The average speedups drop by 14% when using the runtime static scheduler. The dynamic scheduler sees an 8% decrease in speedup on average. Similarly, guided sees a loss of 12%, and afs a loss of 7%. Trapezoidal self-scheduling sees an average loss of 7.4%. The method with the highest average performance is afs with an improvement of 6.2% over the next best scheduling method on 8 processors (guided). However, even afs shows a better average performance when using only 4 threads.

4.4. Adaptive Schedulers

We evaluated the adaptive schedulers described in Section 3: the Region-Based scheduler(RBS), the Loop-Based Scheduler(LBS), and the Hardware-Counter Scheduler (HCS). The performance of the adaptive and non-adaptive schedulers is shown in Figure 7. The performance of each scheduler relative to the single-level schedulers is shown in Figures 12 - 14.

RBS: In [13], RBS was evaluated on a subset of the SPEC OMP2001 benchmarks and was shown to outperform all single-level schedulers on both 4 and 8 processors. Contrary to the results in [13], Figure 12 shows that RBS does not perform well on average when applications with more than 1 parallel loop per region are included. Due to space constraints, we show results only for 8 threads. On 4 processors (not shown), it is outperformed on average by static, guided and afs on the NAS benchmarks. On 8 processors, as shown in Figure 12, its relative performance increases since it can decide to use only 4 processors for some regions. However on average for 8 processors, both guided and afs are still faster for benchmarks taken from NAS. When using 4 threads, RBS is outperformed overall by both static and afs. When using 8 threads, RBS outperforms all schedulers overall, although it's performance on the NAS benchmarks is poor. Figure 8(a) shows the improvement of RBS relative to the best single-level scheduler for each benchmark as well as the original unmodified application.

The decisions selected by RBS, shown in Figure 8(b), show that a variety of scheduler configurations are used across the benchmarks. While RBS performs well on the SPEC benchmarks, the one loop per region assumption causes its performance to suffer on the NAS benchmarks. It should be noted that in both `mgrid` and `mg`, the work performed by the parallel loops vary at each invocation and so both RBS and LBS (Figure 13) bail-out on all regions.

LBS: In Figure 7, the Loop-Based Scheduler (LBS) outperforms all other schedulers on average when using either 4

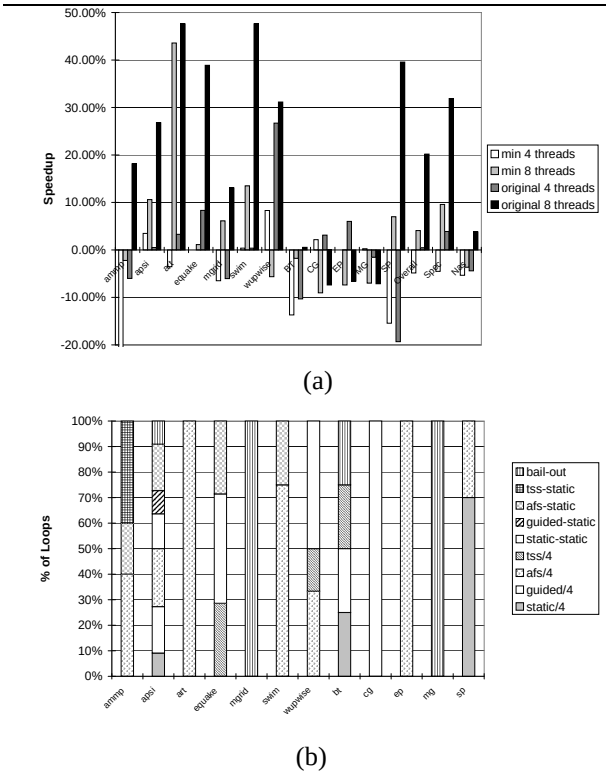


Figure 8. The improvement of RBS over the best single-level schedulers and the original unmodified benchmarks: (a) the percent increase that would be seen by using the RBS and (b) the choices made by RBS for each benchmark. "min on 4" corresponds to the best single-level scheduler for each benchmark when using 4 threads. "min on 8" corresponds to the best single-level scheduler for each benchmark when using 8 threads.

or 8 threads. However as shown in Figure 13, it is outperformed by guided on NAS when using 8 threads (this is also true for 4 threads). Unlike RBS, the LBS selects the best single-level scheduler to use for each parallel loop when using 4 threads. On 8 threads, the LBS uses the full algorithm presented in Figure 3. The average improvement gained by using the LBS over RBS on 8 threads is almost 16%. Figure 9(a) shows the improvement of LBS relative to the best single-level scheduler for each benchmark as well as the original unmodified application. The decisions selected by LBS are shown in Figure 8(b). When each loop is considered independently, the selected schedulers differs considerable from RBS. For example, in ammp afs-static and tss-static is selected for 60% of the regions using RBS, while LBS selects guided/4 and guided-static for 60% of the parallel loops.

Figure 10 shows an example where LBS benfits from the ability to select multiple schedulers for a single region. In this region, RBS chooses static-static 8 threads, resulting in a total execution time of 1.64 seconds. LBS chooses static-static 8 threads for the first loop, and tss using 4 threads for the second and third loop, resulting in a total execution time of 1.40 seconds, a reduction of 15%.

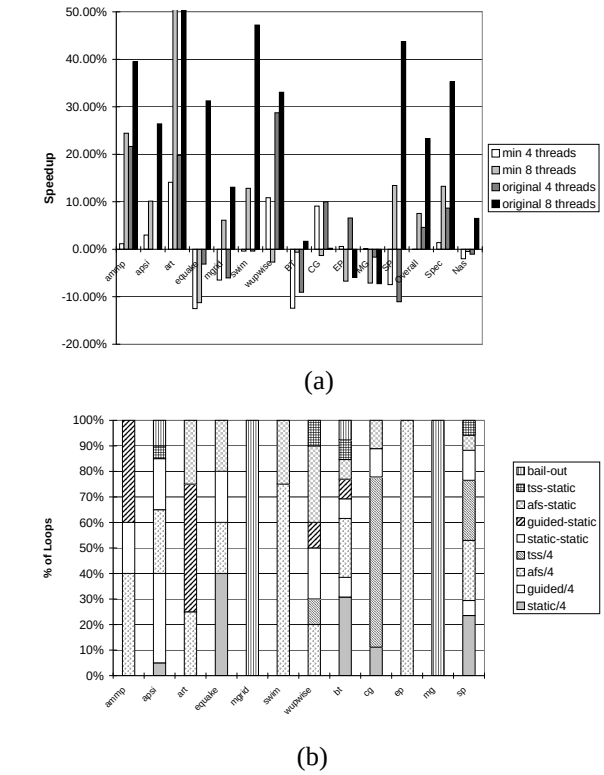


Figure 9. The improvement of LBS over the best single-level schedulers and the original unmodified applications: (a) the percent increase in performance that would be seen by using LBS and (b) the choices made by the LBS for each benchmark. "min on 4" corresponds to the best single-level scheduler for each benchmark when using 4 threads. "min on 8" corresponds to the best single-level scheduler for each benchmark when using 8 threads.

HCS: HCS likewise has better performance then RBS and all single-level schedulers overall, as shown in Figure 14. Using the decision tree to short circuit the sampling phases of RBS and LBS, HCS arrives at accurate decisions with lower overhead. It does not need to sample sub-optimal variants, but instead quickly begins using a efficient sched-

```
!$omp parallel default(shared) private(i,j,k)
!$omp do schedule(runtime)
      do j=1,lastrow-firstrow+1
         do k=rowstr(j),rowstr(j+1)-1
            colidx(k) = colidx(k) - firstcol + 1
         enddo
      enddo
!$omp end do nowait
!$omp do schedule(runtime)
      do i = 1, na+1
         x(i) = 1.0D0
      enddo
!$omp end do nowait
!$omp do schedule(runtime)
      do j=1, lastcol-firstcol+1
         q(j) = 0.0d0
         z(j) = 0.0d0
         r(j) = 0.0d0
         p(j) = 0.0d0
      enddo
!$omp end do nowait
!$omp end parallel
```

Figure 10. The loops in a region from CG.

uler. The average improvement gained by using the HCS over other single-level loop schedulers are above 9%. Figure 11(a) shows the improvement of HCS relative to the best single-level scheduler for each benchmark as well as the original unmodified application. The decisions selected by HCS are shown in Figure 8(b). Again, HCS makes different choices as compared to both RBS and LBS. In many cases, several schedulers have comparable performance, HCS selects a good scheduler with low overhead, offering performance that is comparable to LBS on average.

Both LBS and HCS show better performance than any single-level scheduler on 4 or 8 threads. In addition, on average they both outperform the best performing schedulers for a fixed number of threads. Both LBS and HCS using 8 threads show an average improvement of 4% over the original applications when executed using 4 threads, and 27% over the original applications when executing on 8 threads. RBS when using 8 threads shows a loss of 10% over the original application on 4 threads, and a gain of only 9% over the original application executed on 8 threads. The best average speedup is achieved by LBS executing with 8 threads.

5. Conclusions

Simultaneous multithreaded (SMT) and Hyperthreaded (HT) processors allow multiple threads to execute concurrently on a single physical processor. The unique features of SMT processors make it difficult to determine when to use these extra threads. Ideally, a user could view the threads on an SMT as virtual processors, and execute parallel applications assuming that these virtually processors are all equal. In Section 4, we show that it is sometimes better to execute OpenMP applications using only a single thread per physical processor. Using the additional virtual processors often results in worse performance.

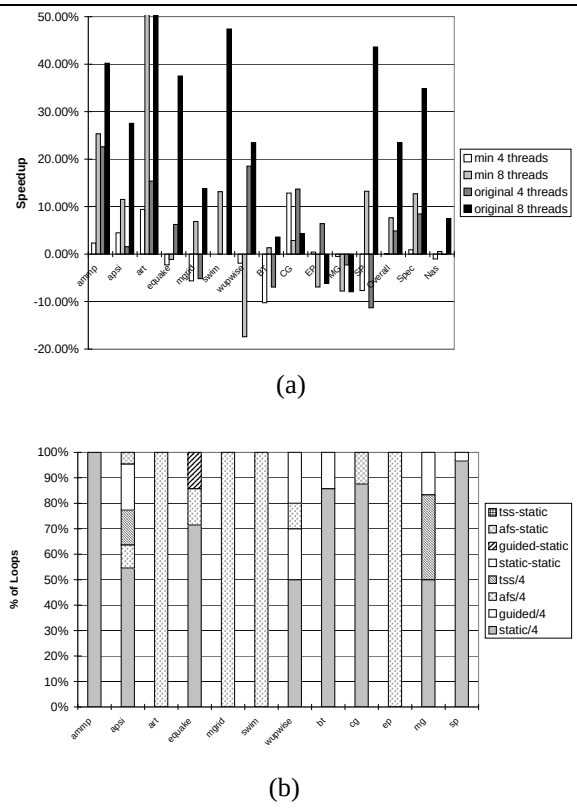
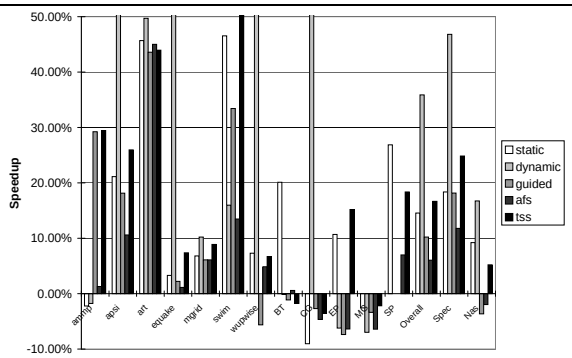


Figure 11. The improvement of HCS over the perfect single-level schedulers: (a) the percent increase in performance that would be seen by using the HCS and (b) the choices made by the HCS for each benchmark. "Perfect on 4" corresponds to the best single-level scheduler for each benchmark when using 4 threads. "Perfect on 8" corresponds to the best single-level scheduler for each benchmark when using 8 threads.

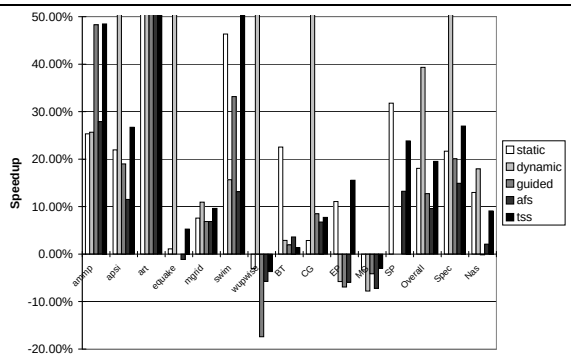
In Section 3, we propose extensions of the Region-based Scheduler (RBS) described in [13]. Our schedulers exploit the two-level structure of SMPs built from SMT processors. Our Loop-based Scheduler (LBS) uses a sampling phase to select the best hierarchical scheduler for each parallel loop. Our Hardware-counter Directed Scheduler (HCS) uses hardware counters to sample loop behavior, short circuiting the sampling required by both RBS and LBS. Both LBS and HCS are shown to outperform a number of single-level non-adaptive schedulers as well as RBS on benchmarks from the SPEC and NAS OpenMP benchmarks suites.

Given the availability of SMT processors in commodity



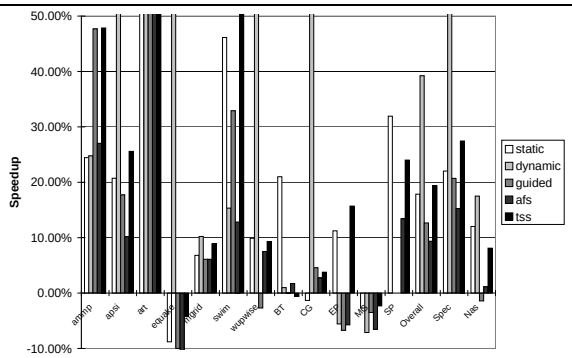
RBS 8 Threads

Figure 12. The improvement of RBS over the single-level schedulers using 8 threads.



HCS 8 Threads

Figure 14. The improvement of HCS over the single-level schedulers using 8 threads.



LBS 8 Threads

Figure 13. The improvement of LBS over the single-level schedulers using 8 threads.

systems, loop schedulers that can handle the unique features of these systems will become increasingly important. Our schedulers through their hierarchical structure, and ability to dynamically select the best scheduling algorithm and number of threads for each parallel loop, frees users from the complex task of selecting best schedules for this emerging architecture.

Traditional schedulers, whether static or runtime, are not aware of the unique characteristics of SMT architectures. Our novel adaptive scheduler addresses these features, enabling an effective coupling of the OpenMP API with these increasingly important SMT-based parallel systems.

References

[1] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in OpenMP? *International Workshop on OpenMP Applications and Tools*, pages 147–159, June 2003.

[2] O. A. R. Board. OpenMP Website. http://www.openmp.org, Oct. 2003.

[3] M. W. Hall and M. Martonosi. Adaptive Parallelism in Compiler-Parallelized Code. In *Proc. of the 2nd SUIF Compiler Workshop*, Aug. 1997.

[4] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. Technical Report TR410, 1992.

[5] 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*, 06(01), Feb. 2002.

[6] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1 edition, 1993.

[7] RWCP. The Omni OpenMP Compiler. http://phase.hpcc.jp/Omni/, 2004.

[8] The OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*, 2.0 edition, Nov. 2000.

[9] The OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, 2.0 edition, Mar. 2002.

[10] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[11] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.

[12] M. J. Voss and R. Eigenmann. Reducing Parallel Overheads Through Dynamic Serialization. In *IPPS: 13th International Parallel Processing Symposium*, pages 88–92, San Juan, Puerto Rico, Apr. 1999.

[13] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *Proceedings of the International Conference on Parallel and Distributed Systems*, San Francisco, CA, 2004.