An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs

Yun Zhang, Mihai Burcea, Victor Cheng, Ron Ho and Michael Voss
Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, ON, M5S 3G4, Canada

Abstract

Hyperthreaded(HT) and simultaneous multi-threaded (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. This work focuses on tuning the behavior of OpenMP applications executing on SMPs with SMT processors. We propose a self-tuning OpenMP loop scheduler designed to react to behavior caused by inter-thread data locality, instruction mix and SMT-related load imbalance. This adaptive loop scheduler automatically selects the number of threads that should be used for each parallel loop and a good scheduling policy for the iterations. It is shown that this scheduler outperforms all other OpenMP schedulers, and because it can dynamically select the number of threads to use for each region, it even outperforms the best combination of runtime schedulers for any fixed number of threads.

1 Introduction

Hyperthreaded (HT) [6] and simultaneous multi-threaded (SMT) [11] processors are now available in commodity systems[1]. SMTs allow multiple threads to execute concurrently on a single physical processor. Intel states that HT yields performance gains as large as 25 - 40% for a less than a 5% increase in die size [6]. The cost is minimal because most resources are shared by the threads on an SMT, including the functional units and the on-chip memory hierarchy.

In this work, we focus on tuning the behavior of OpenMP applications executing on Hyperthreaded SMPs. OpenMP has emerged over the last few years as the dominant programming interface for expressing loop-level parallelism for shared-memory systems. It is designed to allow incremental parallelization of sequential applications using straightforward parallel pragmas. OpenMP is supported by a growing number of hardware and software vendors, and has a number of benchmark suites available for performance evaluation [1, 4]. 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. When an SMP contains SMT processors, the shared caches in each physical processor cause the overall system to behave like a non-uniform memory access (NUMA) machine. 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. If co-located threads share non-idle resources, throughput may suffer.

Finally, most OpenMP applications, such as those found in the SPEC OpenMP Benchmark Suite, 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, we propose a self-tuning OpenMP loop scheduler. Our scheduler is designed to react to behavior caused by the three important application characteristics outlined above, automatically selecting the number of threads that should be used for each parallel loop and a good scheduling policy for the iterations.

[1] We shall use SMT and HT interchangeably in this paper

```
#pragma omp parallel 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 `shared` clause lists variables for which all threads should directly access the original copy of the variable. The `private` clause lists variables for which a local, private copy should be made for each thread. The `schedule(runtime)` clause specifies that the scheduler will be selected by the user at runtime through an environment variable.

2 Related Work

The OpenMP API The OpenMP API has become the industry standard for loop-level shared-memory parallel programming [9, 8] and has strong support from vendors, including IBM, Intel and Sun Microsystems. The standard bindings of OpenMP for C/C++ [9] and Fortran [8] are currently implemented in a wide range of commercial and research compilers. 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 both C and Fortran. A detailed description of OpenMP is beyond the scope of this paper.

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 [2], an adaptive scheduler is designed for the IBM XLF OpenMP compiler that derives the best scheduling policy for each parallel loop at runtime. In [2], 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 [13], parallel loops are dynamically serialized to avoid overheads that cannot be amortized by parallel execution. Unlike our work, the work in [13] finds loops that have sufficient work to amortize parallelization overheads. The work presented here proposes a general self-tuning loop scheduler.

```
_ompc_runtime_sched_init (_p_i0, _p_i1, _p_i2);
while (_ompc_runtime_sched_next (&_p_i0,
                                 &_p_i1)) {
 for (i = _p_i0; i < _p_i1; i += _p_i2) {
   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 call to `_ompc_runtime_sched_init` initializes the runtime-selected scheduler and each call to `_ompc_runtime_sched_next` returns a chunk of iterations for the thread to work on.

3 Overview of Scheduling Methods

The OpenMP API supports parallel regions, parallel sections and parallel loops. While a full description of the OpenMP API is beyond the scope of this paper, a brief overview of parallel loops is presented here. 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 using the OpenMP API. The OpenMP compiler converts these loops into thread-based code with calls to the OpenMP runtime library to perform synchronization and scheduling, as shown in Figure 2.

The loop shown in Figure 2 has been outlined into a subroutine by the Omni compiler. At the location in the code where the loop 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 loop and passes each of them a copy of the function pointer. As each thread executes the loop, they make a call to `_ompc_runtime_sched_init` which initializes the user-selected scheduler. 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.

In this paper, we evaluate different scheduling methods and their effect on the performance of applications executed on a Hyperthreaded SMP. Table 1 describes the schedulers we evaluate, including the standard OpenMP loop schedulers (static, dynamic and guided), as well as two advanced schedulers from the literature (affinity and trapezoidal self-scheduling).

Table 1: A description of the runtime schedulers studied.

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. By default, 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.
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 [5] addresses the 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.
Trapezoidal	Trapezoid self-scheduling[12], 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.

3.1 A Self-Tuning Loop Scheduler

As will be demonstrated in Section 4, none of the loop schedulers described in the previous section effectively reacts to OpenMP applications executing on SMPs with SMT nodes. We therefore propose an advanced self-tuning scheduler tailored specifically for this domain. As shown in Table 2, most execution time is spent on the parallel loops that are executed more than 50 times. If we could decide which scheduler we should use for one particular loop in its first several runs, we can gain significant performance improvement [2]. A pseudo-code description of our algorithm is found in Figure 3 and is described in detail in this section.

Table 2: Number of invocations of parallel loops and their percentage of total program time

Benchmark	< 10 times	10-50 times	> 50 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%

[2]Our system targets SMPs. For NUMA machines, changing the scheduling policy of initialization loops may negatively affect first-touch data distribution policies. Such systems are beyond the scope of our current work.

```
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 two-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 sched
        if per-thread times are now balanced, end loop
     ENDFOR
  ENDIF
  SET T2 to execution time with best two-level sched
  SET lower-level sched to best two-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 Self-Tuning Loop Scheduler Algorithm

First, to account for the NUMA-like structure of SMPs built from SMT-nodes, we propose 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 attempts to exploit this aspect by grouping the processors in nodes (e.g., on our 2-way Hyperthreaded 4-processor system, we have 4 nodes of 2 processors each). The scheduler will assign 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" [3].

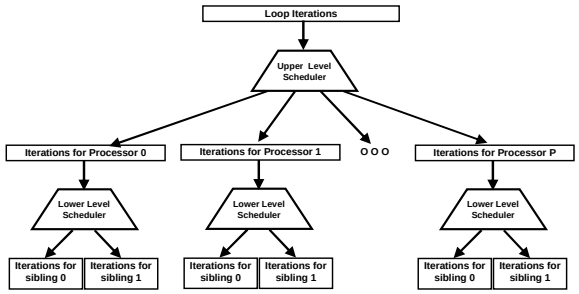


Figure 4: The structure of the hierarchical scheduler.

The additional scheduling overhead from the hierarchical structure should be compensated for by increased data locality among co-located threads. We also expect that the greater flexibility offered by this algorithm will help improve performance of applications by combining an upper-level scheduling algorithm with few synchronizations (and low overhead) with a lower-level algorithm that will at most imply synchronizations between the two siblings.

The various schedulers described in Table 1 each have applications and systems for which they will outperform all other schedulers. It is difficult, however, to make an *a priori* decision about which scheduler to use for any given loop since the choice of best scheduler may depend on characteristics of the system and input data set that is not known at compile-time.

Therefore, our proposed two-level scheduler is also adaptive. It samples the performance of a number of scheduling method at runtime to determine which scheduler performs the best for each loop. We sample all of the schedulers described in Table 1. Our scheduler 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 and the execution time is monitored. After all scheduling methods have been sampled for a given region, the algorithm that gave the smallest execution time is selected as best.

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

As discussed in Section 1, if two co-located threads compete for the same shared resources on an SMT processor, conflicts may degrade overall performance. To react to this situation, our adaptive scheduler determines (a) whether to use 1 or 2 threads per SMT node and (b) if it is decided to use 2 threads, what lower-level algorithm should be employed.

After the Upper-level search phase is complete, our scheduler has selected a upper-level algorithm that has the best performance, which means it can balance work-loads well while minimizing scheduling overhead in the upper level. 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 to execute the loop under consideration.

It begins by sampling the per-thread execution times for this loop when using the static scheduler as the lower algorithm. If per-thread timings indicate that no load imbalance is seen between sibling threads, 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 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 loop, 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.

It should be noted that 8 threads will be used to executed all code that falls outside of parallel loops

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

but within parallel regions, ensuring correct execution. 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).

4 Experimental Evaluation

We evaluate the performance of our scheduling methods using a Hyperthreaded Xeon server with four 2.8 GHz Xeon processors and a 16 GB main memory. Each processor has a 512 KB L2 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 7 benchmark programs described in Table 3.

Table 3: Benchmark Characteristics

Name	Lines	Parallel Loops	Modified Loops	Time on 1 CPU (s)
ammp	14688	10	9	364
apsi	7744	24	24	378
art	1917	3	2	214
equake	1622	11	10	231
mgrid	683	11	11	740
swim	462	8	8	514
wupwise	2506	10	10	1189

4.1 Benchmark Scaling

Figure 5 shows the speedup of each of the original unmodified applications when executed on 1 through 8 threads (compiled with the Omni research compiler version 1.4a). It is important to note that when only the physical processors are used, performance increases with the number of threads. However, for many benchmarks, using only the physical processors leads to better performance. The average speedup is 2.3 on 4 processors and 2.0 on all 8 virtual processors.

4.2 Evaluation of OpenMP Schedulers

To perform our experiments, runtime scheduling directives were added to all of the major parallel loops

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

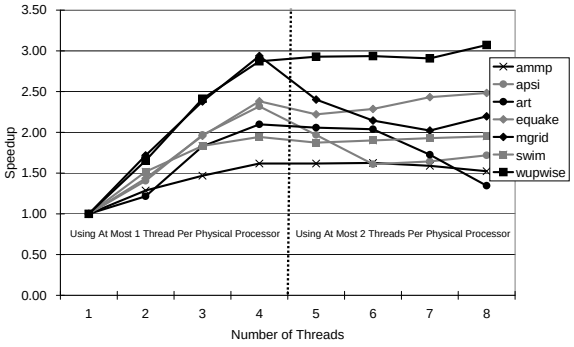


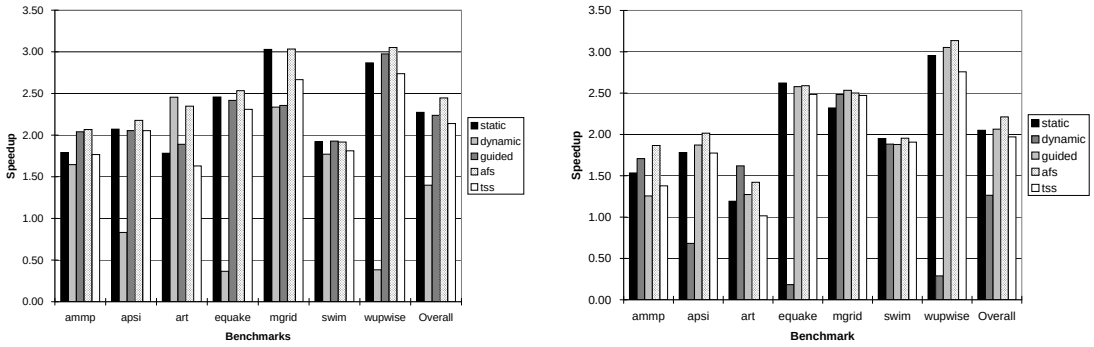
Figure 5: The speedup on 1 through 8 processors using the Omni research compiler with the original parallel applications.

in the benchmark suite (see Table 3). Loops that explicitly specified schedulers were not modified. The Omni compiler uses static scheduling by default for loops without scheduling clauses.

The speedups for the original applications are poor as shown in Figure 5. To investigate the effect of more advanced schedulers on this performance, we executed the benchmarks with 4 and 8 threads using several runtime scheduling algorithms: static, dynamic, guided, affinity and trapezoidal self-scheduling (tss). Since affinity and tss are not available in standard OpenMP compilers, these were added to the Omni research compiler. Figure 6 shows the results of these scheduling methods applied to these benchmarks when executed on 4 and 8 processors.

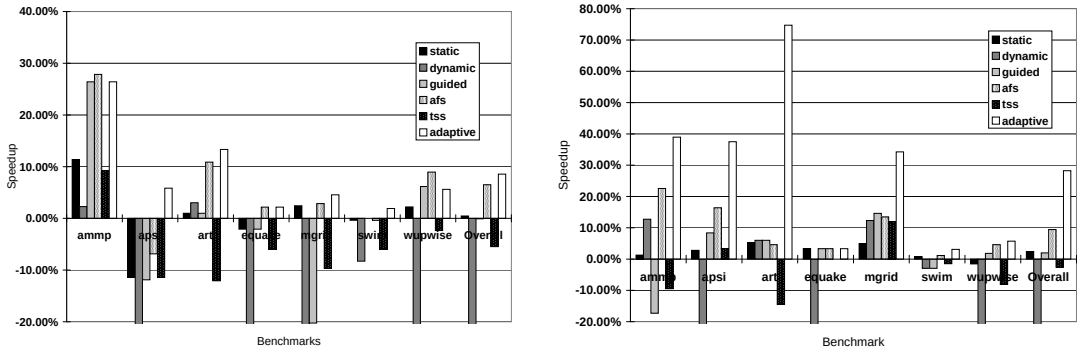
As shown in Figure 6(a), static shows a speedup of 2.27 on average across the benchmarks. The dynamic scheduler, which has the highest overhead, performed very poorly on several applications, resulting in an average speedup of only 1.4. Guided, which still performs runtime load balancing but with a lower overhead than dynamic, had an average speedup of 2.24. Guided shows significant improvements over static on both **ammp** and **wupwise**. The affinity scheduler (afs) shows large improvements in most benchmarks and has an average speedup of 2.45. Affinity also performs load balancing, but uses local work queues to reduce synchronizations and maximize locality.

When 8 threads are used, as shown in Figure 6(b), significant decreases in speedups can be seen for a number of benchmarks. The method with the highest average performance is affinity with an improvement of 15% over the next best scheduling method on 8 processors (guided). However all schedulers, including afs, show a better average performance when using only 4 threads.



(a) (b)

Figure 6: 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.



(a) (b)

Figure 7: The improvement of the runtime schedulers over the original parallel applications when using (a) 4 threads and (b) 8 threads.

4.3 Evaluation of the Adaptive Scheduler

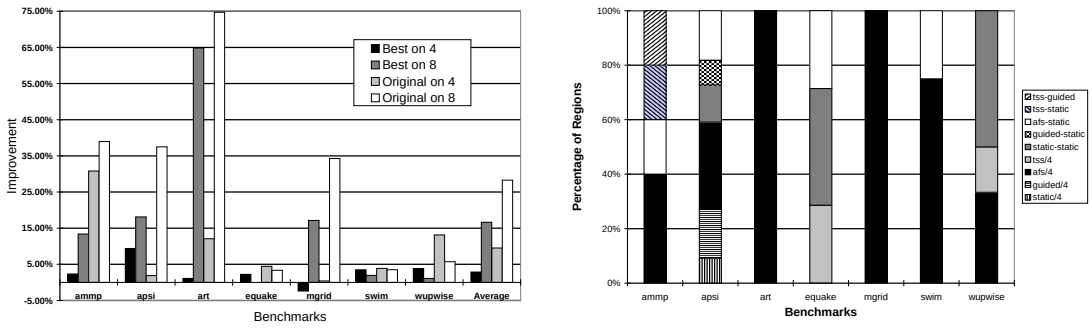
The results in Figure 7 show that the adaptive scheduler on average outperforms all other schedulers when using either 4 or 8 threads. On 4 threads, the adaptive scheduler selects the best single-level scheduler to use for each parallel region, and is therefore able to outperform even the affinity scheduler, which is far better than the other schedulers in Figure 6. On 8 threads, the adaptive scheduler uses the full algorithm presented in Figure 3. The average improvement gained by using the adaptive scheduler over any other scheduler on 8 threads is greater than 20%.

The adaptive scheduler provides significant improvements because (1) it makes independent decisions for each parallel region and (2) on 8 threads it is able to switch between 4 and 8 threads as needed. In Figure 8(a), the improvement of the adaptive scheduler over the best runtime scheduler, and the original application, is shown for each benchmark. The adaptive scheduler provides the best performance for all cases except mgrid, where the best 4-thread runtime scheduler (afs) slightly outperforms it. In mgrid, our adaptive scheduler bails out on several important regions, and therefore cannot amortize its overhead (see Figure 3).

Our adaptive scheduler chooses different schedulers and numbers of threads for parallel regions, as shown in Figure 8(b). Although the affinity scheduler with 4 threads dominates in this decision graph, our adaptive scheduler still chooses other algorithms for some parallel regions, resulting in an increase in performance.

Locality, Imbalance and Instruction Mix

To understand why our adaptive scheduler outperforms all other scheduling methods, we must investigate its effect on inter-thread data locality, load imbalance and instruction mix. Figure 9 shows the effect of schedule choice on data cache misses and load im-



(a) (b)

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

balance. From Figure 9(a), it can be seen that the adaptive scheduler incurs the least increase in data cache misses when moving from 4 to 8 threads. This effect is due to the hierarchical nature of our scheduler (which promotes locality), as well as the choice to use only 4 threads in loops that see significant increases in data cache misses with 8 threads. Figure 9(b) demonstrates that the adaptive scheduler selects algorithms that best balance the load among the threads, yielding significantly improved work distributions.

The effect of instruction mix on the performance of an application executing on a SMT is difficult to measure [10]. However, Table 4 shows evidence that instruction mix plays an important role in determining the number of threads to use per physical processor. In Table 4, both parallel regions see increases in cache misses when 8 threads are used. Region rectmm_32 sees a 5% increase and quake_54 sees a 9% increase in misses. However, rectmm_32 only executes 172 floating-point operations per microsecond (μs), while quake_54 executes 543 floating-point operations per μs. This increased floating-point operation count provides more opportunities for resource sharing on the SMT. Therefore, quake_54 is able to profit from the use of the additional thread context, even at the cost of increased data cache misses.

5 Conclusions

Simultaneous multithreaded (SMT) and Hyper-threaded (HT) processors allow multiple threads to execute concurrently on a single physical processor.

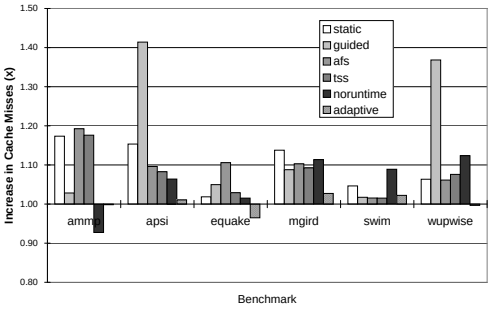
Table 4: Example per-region statistics.

Region	FP Ops	Cache Misses	Time (μs)	Sched
rectmm (reg 32)	690 k	46 k	3856	Static
	690 k	47 k	3619	Guided
	690 k	46 k	3967	TSS
	691 k	**47 k**	**3609**	**AFS**
	691 k	49 k	3997	AFS-Static
quake (reg 54)	7397 k	298 k	16846	Static
	7397 k	299 k	16685	Guided
	7397 k	298 k	17612	TSS
	7399 k	300 k	14979	AFS
	7399 k	**326 k**	**13632**	**AFS-Static**

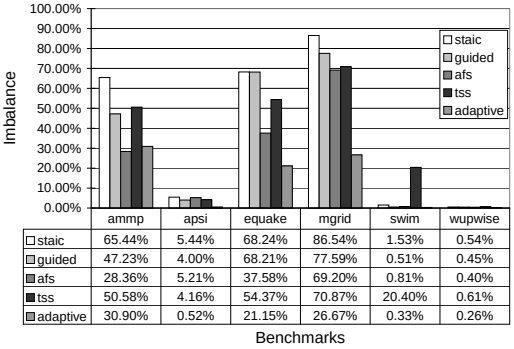
The unique features of SMT processors make it difficult to determine when to use these extra threads. Ideally, a user could view the threads on SMT as virtual processors, and execute parallel applications assuming that these virtually processors are all equal. In Section 4.1, we show that it is sometimes better to execute OpenMP applications using only a single thread per physical processor. Using the additional virtual processors often results in worse performance.

In Section 4.2, we show that using more advanced schedulers may lead to increased performance on SMPs of SMTs. It is shown that an affinity-based runtime scheduler can lead to an almost 10% improvement in the average performance of the benchmarks compared to the original parallel applications.

In Section 3.1, we propose a self-tuning loop scheduler that exploits the two-level structure of SMPs built from SMT processors. It is shown that this scheduler



(a)



	ammp	apsi	equake	mgrid	swim	wupwise
staic	65.44%	5.44%	68.24%	86.54%	1.53%	0.54%
guided	47.23%	4.00%	68.21%	77.59%	0.51%	0.45%
afs	28.36%	5.21%	37.58%	69.20%	0.81%	0.40%
tss	50.58%	4.16%	54.37%	70.87%	20.40%	0.61%
adaptive	30.90%	0.52%	21.15%	26.67%	0.33%	0.26%

(b)

Figure 9: Characterization of the adaptive schedulers effect on (a) data cache misses and (b) load imbalance. In (a), the increase in data cache misses incurred when moving from 4 to 8 threads is shown for each algorithm. In (b), the average load imbalance across all regions is shown on 8 threads for each scheduler. Load imbalance is calculated as the difference between the execution time of the slowest and fastest thread for a given region, normalized to the execution time of the slowest thread.

outperforms all other approaches, and because it can dynamically select the number of threads to use for each region, it even outperforms the best combination of runtime schedulers for any fixed number of threads. With our novel adaptive scheduler, users can safely execute their code with all available threads, and the runtime system will transparently select the appropriate number of threads for the application.

Acknowledgements: 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.

References

[1] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the Workshop on OpenMP Applications and Tools*, pages 1–10, Lafayette, Indiana, July 2001.

[2] 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.

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

[4] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

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

[6] 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), February 2002.

[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, November 2000.

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

[10] N. Tuck and D. Tullsen. Initial Observations of a Simultaneous Multithreading Pentium 4. In *International Conference on Parallel Architectures and Compilations Techniques*, September 2003.

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

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

[13] 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, April 1999.