ILP versus TLP on SMT

Nicholas Mitchell, Larry Carter, Jeanne Ferrante, Dean Tullsen[*]

University of California, San Diego

E-mail: {mitchell,carter,ferrante,tullsen}@cs.ucsd.edu

[*]Tullsen supported by DEC external grant US-0040-97 and UC MICRO grant 97-018. Others supported by NSF grant CCR-9808946 and DARPA grant DABT63-97-C-0028.

Abstract

By sharing processor resources among threads at a very fine granularity, a simultaneous multithreading *processor (SMT) renders thread-level parallelism (TLP) and instruction-level parallelism (ILP)* operationally equivalent. *Under what circumstances are they* performance equivalent*?*

In this paper, we show that operational equivalence does not imply performance equivalence. Rather, for some codes they perform equally well, for others ILP outperforms TLP, and for yet others, the opposite is true. In this paper, we define the performance characteristics that divide codes into one of these three circumstances. We present evidence from three codes to support the factors involved in the model.

Keywords: simultaneous multithreading, tiling, performance prediction, TLP, ILP

1 Introduction

The last few years has seen the emergence of a number of multithreaded processors [1, 2, 18]. A multithreaded processor aims to increase processor utilization by sharing resources at a finer granularity than a conventional processor. In this paper we study a *simultaneous multithreading processor* (SMT) [18]. SMT multiplexes resources between threads within a single cycle; threads share instruction fetch, register pool, execution units, cache, and translation lookaside buffer (TLB) at the finest resolution possible.

By sharing resources at a fine granularity, SMT ideally renders ILP and TLP *operationally equivalent*; both introduce equally many independent instructions into the processor's pipelines [7]. Does operational equivalence imply *performance equivalence*? That is:

> Given an incoming bandwidth of independent instructions, will SMT perform equally well, whether the independence comes from ILP or from TLP?

In this paper, we argue that this is not always the case. Rather, given ILP and TLP are operationally equivalent, implementing parallelism as ILP or TLP may produce the same, lesser, or greater performance. Thus, a choice of implementing available parallelism as ILP versus TLP depends on features of the application and data. For example, matrix multiply has good register and cache reuse. Consequently, the benefit of multithreading a matrix multiplication depends on its level of tuning. With a high level of tuning, to expose register and cache locality, threads *harm* performance. A naive matrix multiply has little locality and hence benefits from TLP. Using one thread, naive matrix multiply takes 10.60 cycles per iteration of the inner loop, and 5.45 cycles per iteration using eight threads. On the other hand, highly tuned matrix multiply performs best with two threads, at 1.36 cycles per iteration, and slows down to 1.98 cycles per iteration using eight threads. Therefore, depending on which implementation of matrix multiply we are given, a parallelizing compiler would use greater or fewer threads.

To get the most out of SMT, a programmer or compiler needs to *predict* which of the three situations will occur. To this end, we present a performance model for SMT. Using this model, a programmer or compiler can implement available parallelism to make best use of SMT.

We present our argument for the non-equivalence of ILP and TLP in three sections. Section 2 describes the simultaneous multithreading architecture. Section 3 gives experimental evidence for the complexities of performance on SMT. Then, Section 4 provides a model for understanding the the data from Section 3. Finally, we present related work in Section 5 and our conclusions in Section 6.

2 Simultaneous Multithreading

Despite the ability to issue many instructions per cycle, superscalar processors sustain low bandwidth without sufficient instruction-level parallelism. Lack of ILP might be due to memory latencies (from cache misses) or pipeline latencies (from long latency instructions, sequentialized computations, or short basic blocks). Hardware multithreaded processors tackle low issue bandwidth by adding another target for parallelism, thread-level parallelism.[1] That is, instructions from other threads become an alternate source to hide the latencies seen by a single thread. They accomplish this by keeping the state of multiple threads, or programs, on the processor at once.

With enough parallelism, TLP can fill issue slots unused for lack of ILP. Fine-grain multithreaded processors [2] interleave thread executions, issuing instructions from, potentially, a different thread each cycle. This can hide virtually all sources of latency, but does not allow the processor to use TLP to supplement a lack of available parallelism within a single cycle on a superscalar processor.

A *simultaneous multithreading processor* [18] issues and possibly fetches instructions from multiple threads in each cycle. This enables the processor to use all available parallelism to fully utilize the execution resources of the machine. Through this increased competition, SMT decreases wasted issue slots and increases flexibility [17].

[1] See [18] for a summary of multithreaded processors.

dcache level	capacity (kbytes)	line size (bytes)	assoc.	banks
L1	32	64	1	8
L2	256	64	4	8
L3	8192	64	1	1

(a) Cache parameters. The L1 and L2 caches are interleaved across multiple banks.

Feature	value
instr. fetch width	8/cycle, 2 threads
active list size	256 per thread
rename registers	100
int, FP queue size	32
integer units	6
floating point units	3

(b) Processor parameters. Four of the six integer execution units can perform loads.

Table 1: Parameters of the SMT simulator.

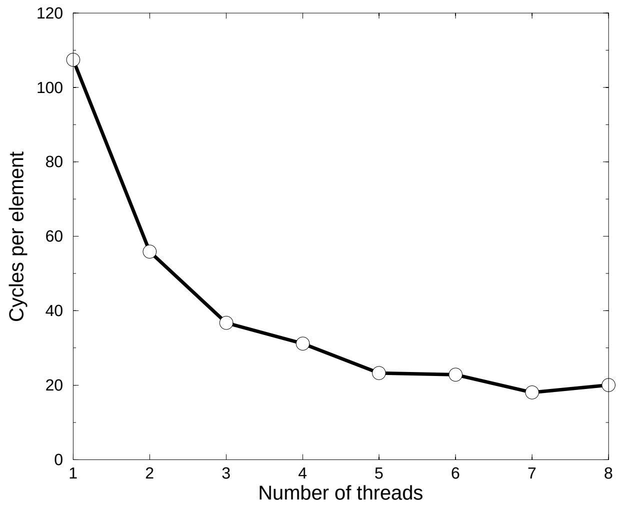

Figure 1: SMT can speed up the naive implementation of matrix multiply by nearly six times on seven threads.

SMT increases flexibility by allowing programmers and compilers to implement available parallelism as either ILP or TLP. Should we take advantage of this flexibility, and always implement parallelism however is most convenient? We argue no. Rather, the way SMT increases flexibility, fine-grained resource sharing, is the very reason we must be careful when implementing parallelism. For example, *increasing* thread concurrency potentially *decreases* per-thread resources. Thus, fine-grained resource sharing introduces new interactions, particular to SMT. We explore these interactions in Section 4.

2.1 Simulation issues

We use the SMT simulator [16] for our experiments. This simulator performs an execution-driven simulation of an SMT processor, including register renaming, branch prediction, three levels of cache, and TLB. In our experiments, the simulator has the parameters shown in Table 1.

As no compiler currently targets SMT, we use a combination of `gcc`[2] and the Digital Unix C compiler. The SMT simulator implements a thread interface (fork, join, etc.) with unused Alpha opcodes. To insert thread primitives, we relied on `gcc`'s mechanism for intermixing assembly code with C programs. We have not quantified the effect of the discrepancy between these compilers' target (an Alpha 21164A) and the actual target (SMT, based on an eight-issue Alpha). However, we believe that hand tuning specifically for the SMT mitigates this discrepancy. Also, SMT issues instructions out-of-order, reducing the effect of processor-specific instruction scheduling.

[2]We used the GNU EGCS compiler, release 1.0.3a.

3 Experimental Evidence

We now present experiments to explore performance on SMT. We study matrix multiply, the Fast Fourier Transform, and integer sort.

3.1 Matrix Multiply

Naive matrix multiply can perform poorly, due in part to high communication-to-computation ratio and high cache and TLB miss rates. However, with proper loop restructuring these latencies can be reduced to the point where floating point operations mask all memory latency [11]. Thus, a naive implementation is latency-bound and an optimized one is computation-bound.

We experiment with varying levels of optimization in twelve implementations of $512 \times 512 \times 512$ matrix multiply.[3] The optimizations include tiling [12] for registers and cache. Tiling is a well-known optimization that enables data locality by reordering memory accesses. We implement register tiling with outer loop unrolling, and cache/TLB tiling by blocking loops. We implement four register tiling variants: no tiling, hoisting the load and store out of the inner loop,[4] and 2x2 and 4x4 tiles. For each of these variants, we implement three cache tiling variants: no tiling, tiled with the outer loop block distributed to threads, and tiled with an inner loop cyclically distributed to threads.[5] Furthermore, we varied the size of

[3]Previous work [8] used $256 \times 128 \times 64$ matrices. They reported that tile size did not affect performance. For their processor configuration, this problem was too small to make cache tiling worthwhile.

[4]`gcc` did not hoist, even on the highest optimization level.

[5]Previously, [8] compared block and cyclic distribution on a variety of problems. They did not tile for registers.

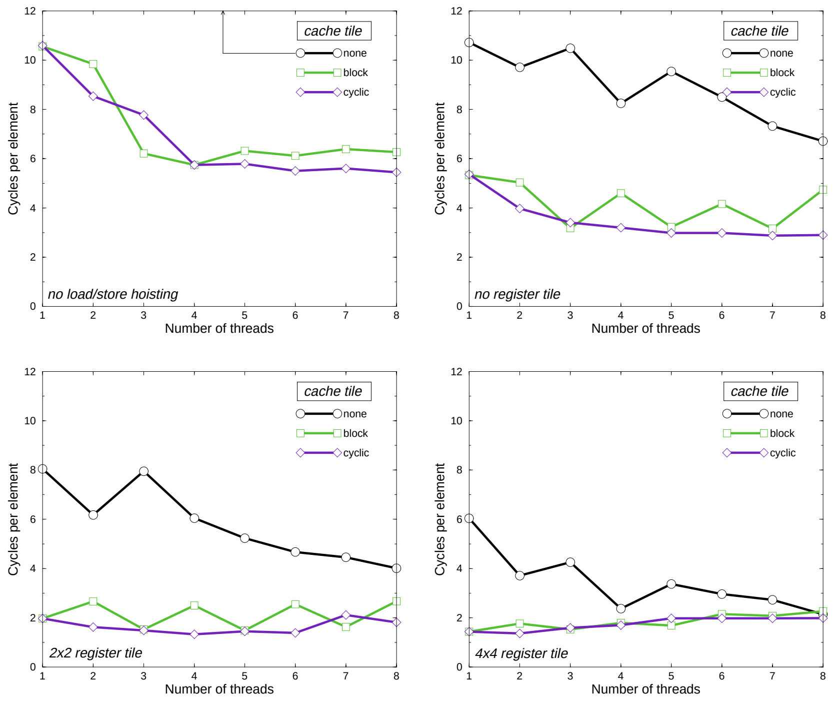

Figure 2: The best *tuned* matrix multiply, at 1.36 cycles per inner loop iteration, uses a 4×4 register tile, a $32 \times 32 \times 32$ cache tile, and two threads. With less tuning, more threads improves performance dramatically. As tuning increases, multithreading has less effect, eventually reducing performance.

the cache tiles from 2 to 128, in powers of 2.

Figure 1 shows that SMT can dramatically speed up the most naive implementation (no hoisting, no register or cache tiling). Work that consumed 107 cycles per iteration on one thread is completed in only 18 cycles per iteration on seven threads. SMT achieves an impressive speedup of 5.95 with 7-way parallelism. On the other hand, careful tuning brings the execution time down to 1.36 cycles per inner loop iteration, using two threads with a 4×4 register tile and a $32 \times 32 \times 32$ cache tile. Figure 2 presents the simulation results for the more highly optimized implementations.

3.2 Fast Fourier Transform

While tuning matrix multiply involves reordering the computation, tuning FFT typically involves a choice of algorithm [9]. We experiment with three FFT algorithms to compute eight one-dimensional FFTs, each with 2^{20} points. The first algorithm, from Numerical Recipes in C [13], is quite naive: it uses an inefficient bit-reversal, doesn't reduce associativity problems, and computes the "twiddle factors" on the fly. We parallelize the computation by distributing the 1-d FFTs cyclically to threads.

Our second algorithm comes from the FFTW package [6]. FFTW (Fastest Fourier Transform in the West) uses a clever technique for choosing the best algorithm based on numerous trial runs.[6] As with the naive version, we parallelize this implementation by distributing 1-d FFTs to threads.

We based our third algorithm on the NAS FT benchmark [3]. FT introduces copying and padding to reduce associativity problems. Furthermore, it interleaves the computation of a number of 1-d FFTs to provide efficient inner loops. Therefore, the parallelization of this implementation differs from the previous two cases. We parallelize the outer loop of the interleaved computation.[7]

Figure 3 shows the results of our simulations. SMT speeds up the Numerical Recipes implementation well: eight threads perform 2.53 times faster than one thread. We observed reduced speedups on the NAS and FFTW implementations. NAS achieved a 1.59x speedup on eight threads and FFTW achieved a 1.3x speedup with four threads (and drops to 1.24x on eight threads). The more highly tuned the algorithm, the less speedup we witnessed. Furthermore, a multithreaded poor algorithm was not a sufficient substitute for a good algorithm, either

[6]FFTW can either estimate a good implementation or determine an optimal algorithm for your target, using dynamic programming. We use the former strategy.

[7]As a consequence, the first three phases only have 1-, 2-, and 4-way parallelism available.

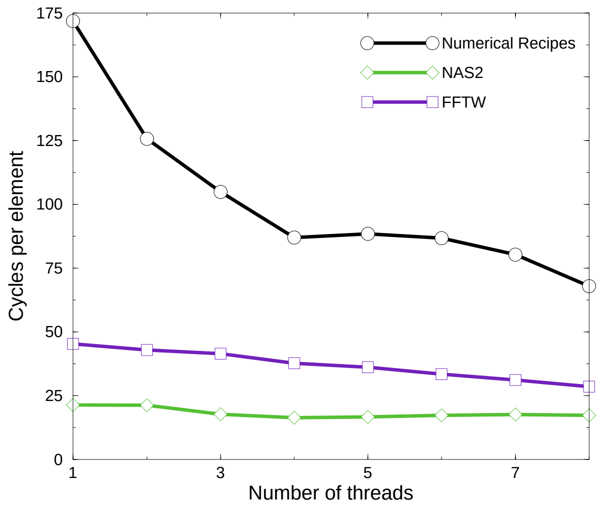

Figure 3: A comparison of three implementations of the Fast Fourier Transform on SMT.

single- or multi-threaded.

3.3 Integer Sort

Given $numkeys$ keys with $maxkey$ possible values, an integer sort counts how many times each key occurs. A standard implementation of integer sort consists of three phases: count the density of key values (countkeys), then perform a running sum of the counts (runsum), and finally rank each of the keys based on this running sum (rankkeys):[8]

```
countkeys   for $i = 0$ to $numkeys$
                $count[key[i]]$++
            end

runsum      for $i = 0$ to $maxkey$
                $count[i]$ += $count[i-1]$
            end

rankkeys    for $i = 0$ to $numkeys$
                $rank[i] = count[key[i]]$--
            end
```

This standard implementation has poor locality and cannot be parallelized efficiently on most architectures.[9] The source of both problems is indirect memory references in the countkeys and rankkeys loops. When sorting randomly distributed data, the countkeys loop makes $numkeys$ references to random locations in $count$. When $maxkey$ is small, the entire $count$ array fits in cache and these references are fast; when $maxkey$ is large, there will be many cache misses. For instance, when sorting two million keys on a Pentium Pro, with $maxkey = 50,000$, the countkeys loop averaged 159 ns per iteration. With $maxkey = 500,000$, this time nearly quadrupled, to 599 ns per iteration.

In previous work, we developed a technique to localize these problematic references [10].[10] Our technique, *bucket tiling* processes the keys into a number of buckets, and then modifies the countkeys and rankkeys loops to process each bucket atomically (compare this with standard tiling, which partitions a computation into regularly-shaped pieces and then computes all the iterations in a given piece atomically [12]).

3.3.1 Processor Overloading

We now study the effect of ILP on processor overloading. While a single bucket does not have enough parallelism to fully exploit SMT,[11] IS has abundant inter-bucket parallelism. We may implement this parallelism as either ILP or TLP. We increase TLP with straightforward SPMD thread parallelization, and ILP by creating code which interleaves the execution of multiple buckets. As the number of keys per bucket may be nonuniform, we load-balance the bucket computations on threads with a work queue.

If ILP and TLP were interchangeable, we would expect the interleaved versions to perform the same as the threaded version. However, this is not the case, due to processor overloading.

The first problem arose due to register pressure. By targeting a 32-register instruction set, gcc generates inefficient code for the two- and four-way interleaved versions; processing more buckets within a single thread increases register live ranges. Rather than spilling, gcc schedules instructions to reduce register live ranges. Consequently, it must increase addressing overhead and memory operations. For example, the 2-way interleaved variant issues 20% more instructions than the threaded version. Also, the threaded version issues a 75-25 mix of integer and load/store instructions, while the 2-way interleaved version has a 69-31 mix.

The second problem arose due to overloading the resources of SMT. On both the threaded and interleaved versions, the major bottleneck was registers. However, the bottleneck had more affect on the ILP threaded versions. To determine the effect of register pressure, we experi-

[8]There are two variants of integer sort, as represented by NAS version 1 and NAS version 2. Version 2 needs less memory bandwidth, as it does not include the rankkeys phase. This paper considers the version 1 variant.

[9]Given very cheap synchronization, naive integer sort parallelizes well. For example, integer sort performs excellently on the Tera [2], due to it's in-memory "full-empty" bits.

[10]A similar technique can parallelize indirect memory references [14].

[11]Therefore loop unrolling itself cannot increase single-thread parallelism.

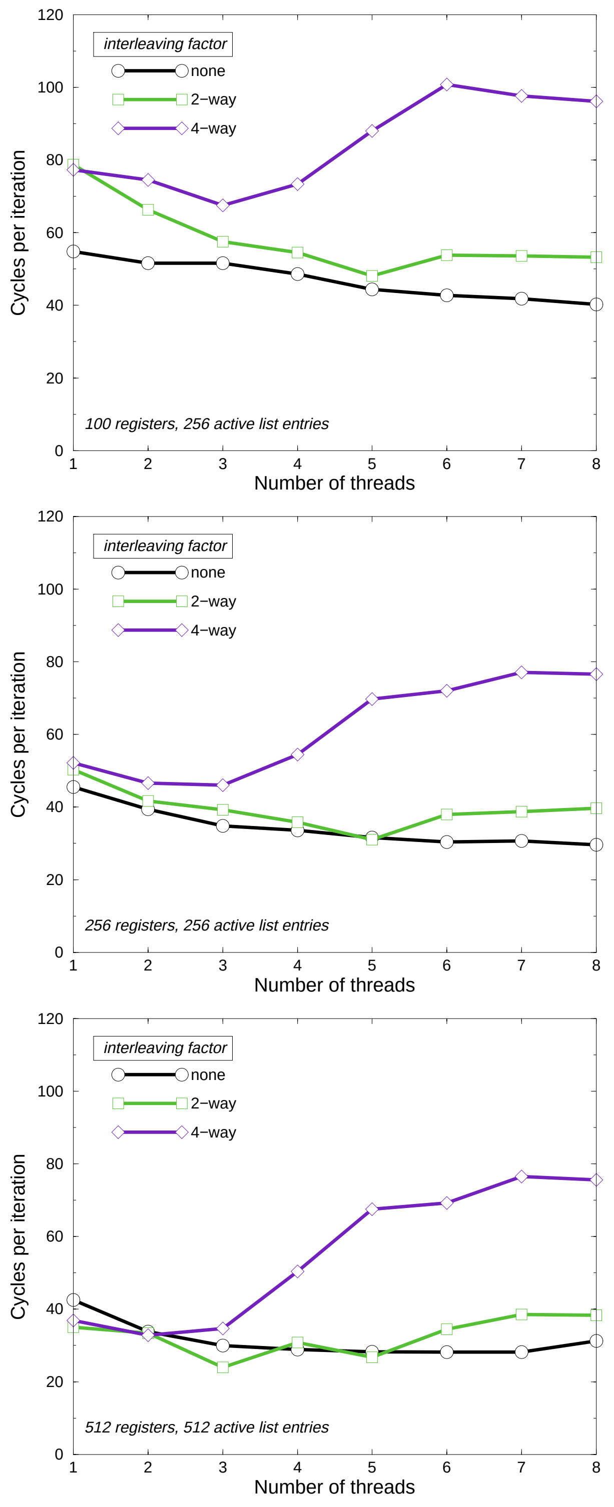

Figure 4: How much parallelism we can implement as ILP (via *interleaving*) depends on the availability of processor resources (number of rename registers and active list entries). Too much ILP with too few processor resources may overload the resources. TLP does not share this problem.

registers	A.L. entries	ILP slowdown	bottleneck
100	256	-44.1%	registers
256	256	-21.7%	A.L. entries
256	512	-3.6%	registers
512	512	-3.5%	A.L. entries

Table 2: In integer sort, we can implement parallelism as either ILP or TLP. Too much ILP in a single thread over-taxes registers and active list (A.L.) entries. Thus, on smaller configurations, ILP performs worse than implementing an equal amount of parallelism as TLP.

mented with four machine configurations, shown in Table 2. As the table shows, with the standard configuration (100 registers and 256 active list entries), the interleaved version was 44% slower than the threaded version. As we increased resources, the threaded and interleaved versions nearly reached parity. Figure 4 gives more detail.

3.3.2 Cache Overloading

Finally, we discuss the impact of TLP on cache and TLB locality in integer sort. We consider two instances of this interaction: between bucket size and amount of TLP, and between the phases of integer sort.

Bucket size: When bucket tiling integer sort, we must choose the size of the buckets. When tiling a computation for a conventional processor, the choice of tile size depends on the size of the problem and the target machine’s characteristics [11]. On SMT, threads share cache and TLB. How does this sharing affect tile choice? Figure 5 shows the effect of the interaction between TLP and locality in integer sort. As we increase TLP, the best bucket size decreases. With one thread the best choice uses a bucket size of 2^{16}, whereas with eight threads, the best choice uses 2^{13}.

Inter-phase interactions: Bucket-tiled IS consists of three computations: bucketize, the main IS computation, and cleanup. The middle phase prefers a different bucket size than the other two, as shown in Figure 6. With one thread, though a larger bucket size increases the middle phase time by 76%, this increase is more than offset by decreased times for the other two phases. On the other hand, with eight threads, the trade-off favors a smaller bucket size.

There are two ways to overcome the disparity between the phases: reducing TLP in the middle phase, and *two-level* bucketing. The working set of the middle section is linearly proportional to both bucket size and number of threads. Thus, using fewer threads in the middle should permit a larger tile size, without harming performance in the middle section. Preliminary results indicate reducing

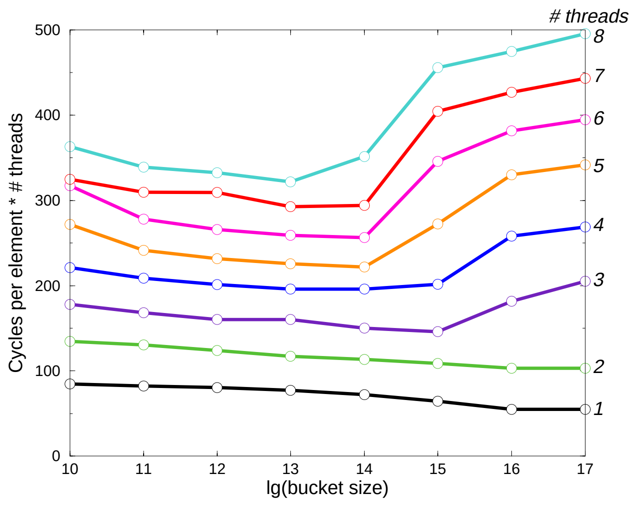

Figure 5: In integer sort, the best bucket size depends on number of threads.

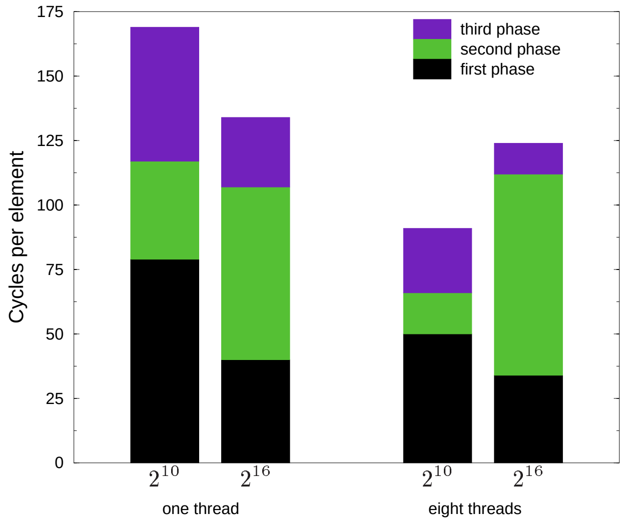

Figure 6: Increasing bucket size from 2^{10} to 2^{16} benefits the first and third phases of tiled integer sort, but harms the middle phase. This trade-off favors larger bucket sizes with fewer threads.

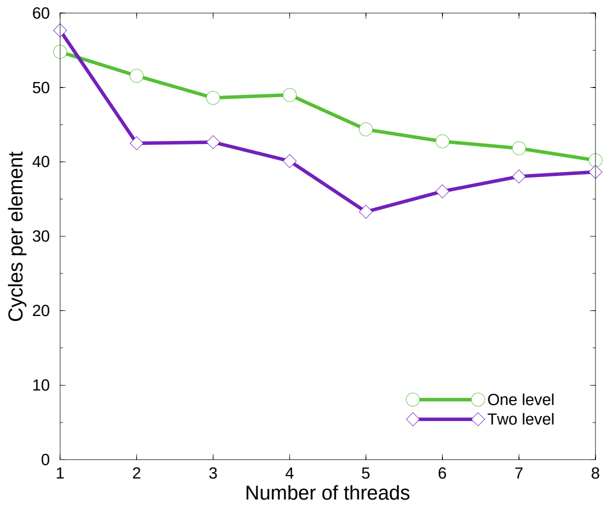

Figure 7: Comparison of two tilings of integer sort: two-level bucketing outperforms one-level bucketing.

	speedup from 2-level bucketing	
threads	SMT	SMP
1	-5%	-6.7%
2	+21.3%	-10.3%
4	+22.2%	-23.4%
8	+4.1%	-31.7%

Table 3: When optimizing integer sort, two levels of bucketing helps an SMT, but not an SMP (an HP/Convex Exemplar).

TLP might be fruitful. Two-level bucketing takes a different tack. By bucketing twice, the working set size of the first and third sections is inversely proportional to the second tile size (rather than first tile size). Figure 7 shows that two levels of bucketing outperforms one level of bucketing, except when using one thread; with one thread, the one-level version does not suffer from multiple threads sharing cache and TLB.

Is two-levels of bucketing an SMT-specific optimization? By sharing cache and TLB among threads, SMT benefits from two levels of bucketing. We hypothesize that two-level bucketing will not help the case where threads do not share caches. To test this theory, we experimented with an SMP.[12] We found that, unlike SMT, two-levels of bucketing did not help this SMP, as shown in Table 3.

[12]Our tests use one node of an HP/Convex Exemplar X Class, model SPP 2000, with 180MHz HP PA-8000 processors. One node of this machine is effectively an SMP with a crossbar to main memory.

performance factor		naive 1 thread	naive 8 threads	best tuned
E (cycles/iter.)		108	20.1	1.36
f_1		2.00	2.00	0.284
f_2	(cycles)	26.4	63.8	8.14
f_3		19.6	14.9	9.8

Table 4: Some of the performance factors on three implementations of matrix multiply, determined experimentally. E is the execution time of each implementation. Section 4 defines the other factors.

performance factor		NR 1 thread	NR 8 threads	FFTW 4 threads
E (cycles/elem.)		1375	543	132
f_1		0.967	0.966	0.541
f_2	(cycles)	33.2	42.3	29.4
f_3		18.9	17.7	14.5

Table 5: Some of the performance factors on three implementations of FFT, determined experimentally. E is the execution time of each implementation. The other factors are defined in Section 4.

4 Predicting Performance

Sometimes TLP and ILP represent two different ways to expose the same parallelism; e.g., loop unrolling versus assignment of consecutive iterations to different threads. In that case, we must decide how to expose the parallelism most effectively. Other times, they can expose very different parallelism; e.g., when very coarse-grain parallelism is available. Even then, it may be that the full exploitation of one may eliminate the need to pursue the other.

In order to make a choice between TLP and ILP, we must understand their performance implications. The performance of an implementation depends on a number of factors, including:

- nt: the number of threads in the implementation
- f_1: the ratio of communication to computation (which is a measure of register locality)
- f_2: cache and TLB locality (measured by average memory access time)
- f_3: demand for processor resources (measured by number of cycles in which a resource had a conflict, averaged over all resources, but excluding execution units).
- f_5: number of instructions in the implementation

In this paper, we explore the effect of nt, f_1, f_2, and f_3 on performance. Table 4 shows the values of these factors on several implementations of matrix multiply; Table 5 does likewise for FFT.

	nt	f_1	f_2	f_3
MM	-0.16	0.60	-0.06	0.43
FFT	-0.11	-0.08	0.48	0.89
IS	-0.02	0.06	0.65	0.66

Table 6: How execution time correlates with the four performance factors; each entry shows, for the three codes, the correlation coefficient of execution time with a performance factor.

Expressing implications of ILP and TLP on each factor, and the effect of each factor on overall performance allows us guide implementation choices.

Implications of ILP: Dedicating parallelism to ILP increases demand for processor resources (f_3) and instruction cache (f_5). For example, it increases number of register names and number of instructions per "unit" of parallelism expressed. If there is lots of register (f_1) and cache reuse (f_2), then most likely ILP will do well.

Implications of TLP: Dedicating parallelism to TLP increases demand for data cache and TLB resources (f_2). If there is little register (f_1) and cache reuse (f_2), then most likely TLP will do well.

Unifying the Factors: Can we use these factors to model performance on SMT? To validate the factors, we must express the combined effect of the factors on performance. We do so by computing the *weighted* sum of the performance factors. We experimentally determine the weights of each of the factors, as in [4]. In that paper, Brewer used linear regression combined with profiling to experimentally, and automatically, derive cost functions. His system takes as input a set of implementation characteristics along with a collection of profiling runs; each run gives the execution time of an implementation with certain characteristics. Then, his system performs a linear regression to fit the input characteristics to execution time; his system also used cross-terms to allow for execution time to vary non-linearly with the characteristics.

We use Brewer's technique to verify the predictive qualities of the performance factors on SMT. Table 6 summarizes the correlation between the four factors and execution time for our three codes. Notice two things. First, in none of the codes is execution time highly correlated with number of threads. Second, the factors highly correlated with execution time vary from code to code. In the remainder of this section, we go into detail for each of the three codes.

4.1 Matrix Multiply

For matrix multiply, we used all the implementations from Section 3.1: multiplication of 512×512 matrices, using one to eight threads, with register tiles ranging from

	linear	quadratic	cubic	quartic
MM	52.3%	75.7%	88.1%	93.6%
FFT	89.7%	99.9%	—	—
IS	67.3%	82.7%	88.4%	95.0%

Table 7: When regressing the four performance factors to execution time, we had to use higher order cross-terms to achieve good fits.

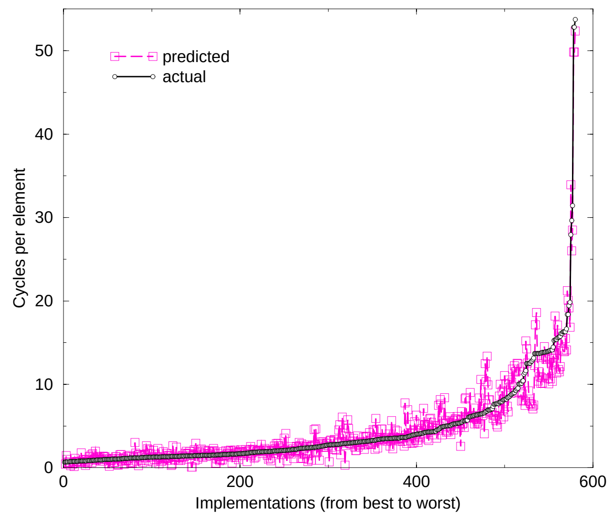

Figure 8: Predicting performance of matrix multiply with all terms through fourth-degree gives an R^2 of 93.6%. Using the 55 most important terms gives an R^2 of 92.5%.

none to 4×4, and with cache tiles ranging from none to 128×128. As Table 6 indicates, none of the factors alone is an overriding determinant of execution time. However, communication-to-computation ratio, f_1, has the highest correlation with performance, with a correlation coefficient of 0.60.

How well do the performance factors predict performance on matrix multiply? As Table 7 shows, using only the four factors as terms in the regression gives an R^2 of 52.3%. However, using all terms up through fourth-degree terms gives an R^2 of 93.6%. Figure 8 visualizes the predictiveness of this latter regression.

4.2 FFT

For FFT, we used implementations with one to eight threads of the three algorithms described in Section 3.2. In these implementations, the factor most highly correlated with execution time for FFT was f_3, demand for processor resources. Recall that f_3 includes processor stalls due to running out of rename registers. FFT is notorious for its demand for registers. Figure 9 shows the result of regressing the factors, including second-degree terms, to

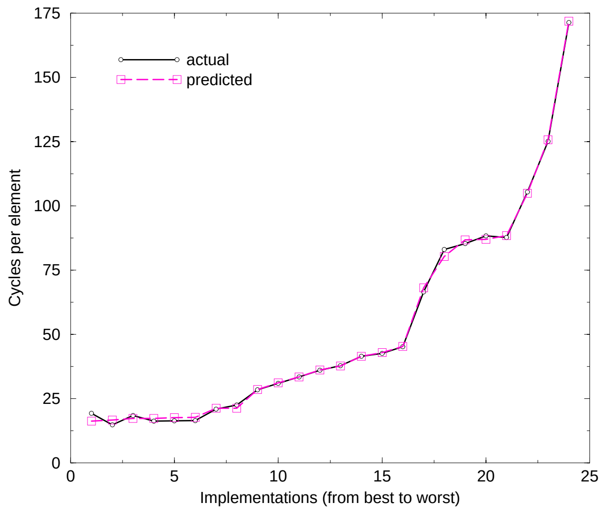

Figure 9: Predicting performance of FFT with all terms through second-degree gives a R^2 of 99.9%. Using only the five most important terms gives an R^2 of 98.37%.

execution time. This regression had an R^2 of nearly 1.

4.3 Integer Sort

For integer sort, we used runs from one to eight threads of four implementations from Section 3.3: one-level bucketing, two-level bucketing, and the two- and four-way interleaved variants. As shown in Table 6, the performance of these implemenations was more highly correlated with f_2, cache and TLB locality, and f_3, demand for processor resources. When regressing the four factors to execution time, we had to use terms up through fourth degree to achieve a reasonable fit. Figure 10 shows this fourth-degree regression, which had an R^2 of 95.03%.

5 Related work

Finally, we summarize related work in four categories:

Architectures: Increasing concurrency has been a target of a number of recent architectural developments. Out-of-order-execution processors aim to increase concurrency in the form of fined-grained, instruction-level parallelism. Simultaneous multithreading processors [18], the Tera Multithreaded Architecture [2], and symmetric multiprocessors all increase concurrency through increasingly coarser, thread-level parallelism (also known as fork-and-join parallelism). Distributed memory machines enable message-passing parallelism.

Architectural evaluation: Thekkath and Eggers evaluated the benefit of a varying number of hardware *contexts* with a fixed number of *threads* [15]. They found that multiple contexts benefit locality-optimized codes more

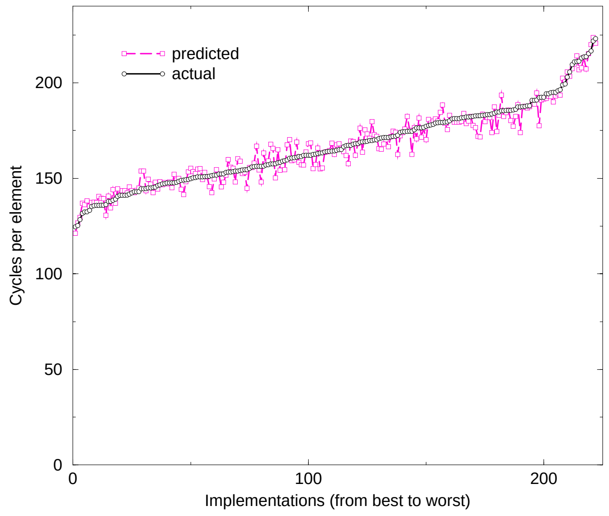

Figure 10: Predicting performance of integer sort with fourth-degree terms gives an R^2 of 95.03%.

than naive codes. They also discovered both decreasing cache conflicts and increasing associativity helped locality-optimized codes more than the naive codes. Their processor switched contexts every 4000 cycles, more coarsely than SMT.

Tullsen *et al.* studied the performance of SMT on multiprogram workloads [18]. They found that SMT sped up benchmarks upwards of five times. They did not look at the performance of SMT on single-program workloads. Lo *et al.* [7] compared single-program performance of SMT and multiprocessor configurations. They found SMT can be as much as 2.68 times faster than a multiprocessor. However, they did not factor in the effect of performance tuning on speedups.

Compiler issues for multithreading: Lo *et al.* previously studied compilation for SMT [8]. In this work, they presented speedups on single-program workloads for two parallelization strategies: blocking the outer loop to threads and cyclic distribution of an inner loop to threads. They did not study the speedup of SMT on highly optimized codes.

Optimization techniques: Tiling partitions nested loops into regular size and shape pieces [20, 21, 19]. Hierarchical tiling [5] attempts to optimize for multiple levels of the memory hierarchy, in concert.

6 Conclusion

One of the major advantages of SMT is the flexibility it offers programmers and compiler writers. With a conventional shared memory multiprocessor (SMP), we have no choice but to implement fine-grained parallelism as ILP and coarse-grained parallelism as TLP: the cost of forking threads, and the context-switch granularity of SMP are simply too large to do otherwise. SMT increases flexibility through *fine-grain resource sharing*.

In this paper we have presented two main results. First, in Section 3, we presented experimental evidence to show that fine-grain resource sharing introduces new interactions (and hence complexities) into the programming and compilation process. We must now decide how to implement available parallelism. Hence, in Section 4, we demonstrated a model for dealing with these interactions. By modeling the *factors* affecting performance, we can make predictions as to the best choice of implementation. In Section 4, we described these factors, and experimentally determined the important factors of performance, for SMT.

References

[1] Anant Agarwal, Ben-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] David Bailey and et. al. NAS parallel benchmarks. `http://science.nas.nasa.gov/Software/NPB/`.

[4] Eric Brewer. High-level optimization via automated statistical modeling. In *Principles and Practice of Parallel Programming*, 1995.

[5] Larry Carter, Jeanne Ferrante, and S. Flynn Hummel. Efficient parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[6] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1997.

[7] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

[8] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, December 1997.

[9] Charles Van Loan. Computational frameworks for the fast fourier transform. In *SIAM Conference on Parallel Processing for Scientific Computing*, Philidelphia, PA, 1992.

[10] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Parallel Architectures and Compilation Techniques*, Newport Beach, CA, October 1999.

[11] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. In *International Journal on Parallel Programming*, June 1998.

[12] Steven S. Munchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1997.

[14] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message-passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[15] Radhika Thekkath and Susan J. Eggers. The effectiveness of multiple hardware contexts. In *Architectural Support for Programming Languages and Operating Systems*, pages 328–337, San Jose, CA, October 1994.

[16] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Computer Measurement Group Conference*, December 1996.

[17] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Hank M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture*, May 1996.

[18] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, 1995.

[19] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.

[20] Michael J. Wolfe. Iteration space tiling for memory hierarchies. In *Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[21] Michael J. Wolfe. More iteration space tiling. In *Supercomputing*, pages 655–664, 1989.