Tuning Compiler Optimizations for Simultaneous Multithreading

Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen*

Dept. of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
{jlo, sparekh, eggers, levy}@cs.washington.edu

*Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
tullsen@cs.ucsd.edu

Abstract

Compiler optimizations are often driven by specific assumptions about the underlying architecture and implementation of the target machine. For example, when targeting shared-memory multiprocessors, parallel programs are compiled to minimize sharing, in order to decrease high-cost, inter-processor communication.

This paper reexamines several compiler optimizations in the context of simultaneous multithreading (SMT), a processor architecture that issues instructions from multiple threads to the functional units each cycle. Unlike shared-memory multiprocessors, SMT provides and benefits from fine-grained sharing of processor and memory system resources; unlike current uniprocessors, SMT exposes and benefits from inter-thread instruction-level parallelism when hiding latencies. Therefore, optimizations that are appropriate for these conventional machines may be inappropriate for SMT. We revisit three optimizations in this light: loop-iteration scheduling, software speculative execution, and loop tiling. Our results show that all three optimizations should be applied differently in the context of SMT architectures: threads should be parallelized with a cyclic, rather than a blocked algorithm; non-loop programs should not be software speculated, and compilers no longer need to be concerned about precisely sizing tiles to match cache sizes. By following these new guidelines, compilers can generate code that improves the performance of programs executing on SMT machines.

1 Introduction

Compiler optimizations are typically driven by specific assumptions about the underlying architecture and implementation of the target machine. For example, compilers schedule long-latency operations early to minimize critical paths, order instructions based on the processor's issue slot restrictions to maximize functional unit utilization, and allocate frequently used variables to registers to benefit from their fast access times. When new processing paradigms change these architectural assumptions, however, we must reevaluate machine-dependent compiler optimizations in order to maximize performance on the new machines.

Simultaneous multithreading (SMT) [32][31][21][13] is a multithreaded processor design that alters several architectural assumptions on which compilers have traditionally relied. On an SMT processor, instructions from multiple threads can issue to the functional units each cycle. To take advantage of the simultaneous thread-issue capability, most processor resources and all memory subsystem resources are dynamically shared among the threads. This single feature is responsible for performance gains of almost 2X over wide-issue superscalars and roughly 60% over single-chip, shared memory multiprocessors on both multi-programmed (SPEC92, SPECint95) and parallel (SPLASH-2, SPECfp95) workloads; SMT achieves this improvement while limiting the slowdown of a single executing thread to under 2% [13].

Simultaneous multithreading presents to the compiler a different model for hiding operation latencies and sharing code and data. Operation latencies are hidden by instructions from *all* executing threads, not just by those in the thread with the long-latency operation. In addition, multi-thread instruction issue increases instruction-level parallelism (ILP) to levels much higher than can be sustained with a single thread. Both factors suggest reconsidering uniprocessor optimizations that hide latencies and expose ILP at the expense of increased dynamic instruction counts: on an SMT the latency-hiding benefits may not be needed, and the extra instructions may consume resources that could be better utilized by instructions in concurrent threads.

Because multiple threads reside within a single SMT processor, they can cheaply share common data and incur no penalty from false sharing. In fact, they benefit from cross-thread spatial locality. This calls into question compiler-driven parallelization techniques, originally developed for distributed-memory multiprocessors, that partition data to physically distributed threads to avoid communication and coherence costs. On an SMT, it may be beneficial to parallelize programs so that they process the same or contiguous data.

This paper investigates the extent to which simultaneous multithreading affects the use of several compiler optimizations. In particular, we examine one parallel technique (loop-iteration scheduling for compiler-parallelized applications) and two optimizations that hide memory latencies and expose instruction-level parallelism (software speculative execution and loop tiling). Our results prescribe a different usage of all three optimizations when compiling for an SMT processor.

We found that, while blocked loop scheduling may be useful for distributing data in distributed-memory multiprocessors, cyclic iteration scheduling is more appropriate for an SMT architecture, because it reduces the TLB footprint of parallel applications. Since SMT threads run on a single processor and share its memory hierarchy, data can be shared among threads to improve locality in memory pages.

Software speculative execution may incur additional instruction overhead. On a conventional wide-issue superscalar, instruction throughput is usually low enough that these additional instructions simply consume resources that would otherwise go unused. However, on an SMT processor, where simultaneous, multi-thread instruction issue increases throughput to roughly 6.2 on an 8-wide processor, software speculative execution can degrade performance, particularly for non-loop-based applications.

Simultaneous multithreading also impacts loop tiling techniques and tile size selection. SMT processors are far less sensitive to variations in tile size than conventional processors, which must find an appropriate balance between large tiles with low instruction overhead and small tiles with better cache reuse and higher hit rates. SMT processors eliminate this performance sweet spot by hiding the extra misses of larger tiles with the additional thread-level parallelism provided by multithreading. Tiled loops on an SMT should be decomposed so that all threads compute on the same tile, rather than creating a separate tile for each thread, as is done on multiprocessors. Tiling in this way raises the performance of SMT processors with moderately-sized memory subsystems to that of more aggressive designs.

The remainder of this paper is organized as follows. Section 2 provides a brief description of an SMT processor. Section 3 discusses in more detail two architectural assumptions that are affected by simultaneous multithreading and their ramifications on compiler-directed loop distribution, software speculative execution, and loop tiling. Section 4 presents our experimental methodology. Sections 5 through 7 examine each of the compiler optimizations, providing experimental results and analysis. Section 8 briefly discusses other compiler issues raised by SMT. Related work appears in Section 9, and we conclude in Section 10.

2 The microarchitecture of a simultaneous multithreading processor

Our SMT design is an eight-wide, out-of-order processor with hardware contexts for eight threads. Every cycle the instruction fetch unit fetches four instructions from each of two threads. The fetch unit favors high throughput threads, fetching from the two threads that have the fewest instructions waiting to be executed. After fetching, instructions are decoded, their registers are renamed, and they are inserted into either the integer or floating point instruction queues. When their operands become available, instructions (from any thread) issue to the functional units for execution. Finally, instructions retire in per-thread program order.

Little of the microarchitecture needs to be redesigned to enable or optimize simultaneous multithreading -- most components are an integral part of any conventional, dynamically-scheduled superscalar. The major exceptions are the larger register file (32 architectural registers per thread, plus 100 renaming registers), two additional pipeline stages for accessing the registers (one each for reading and writing), the instruction fetch scheme mentioned above, and several per-thread mechanisms, such as program counters, return stacks, retirement and trap logic, and identifiers in the TLB and branch target buffer. Notably missing from this list is special per-thread hardware for scheduling instructions onto the functional units. Instruction scheduling is done as in a conventional, out-of-order superscalar: instructions are issued after their operands have been calculated or loaded from memory, without regard to thread; the renaming hardware eliminates inter-thread register name conflicts by mapping thread-specific architectural registers onto the processor’s physical registers (see [31] for more details).

All large hardware data structures (caches, TLBs, and branch prediction tables) are shared among all threads. The additional cross-thread conflicts in the caches and branch prediction hardware are absorbed by

SMT's enhanced latency-hiding capabilities [21], while TLB interference can be addressed with a technique described in Section 5.

3 Rethinking compiler optimizations

As explained above, simultaneous multithreading relies on a novel feature for attaining greater processor performance: the coupling of multithreading and wide-instruction issue by scheduling instructions from different threads in the same cycle. The new design prompts us to revisit compiler optimizations that automatically parallelize loops for enhanced memory performance and/or increase ILP. In this section we discuss two factors affected by SMT's unique design, data sharing among threads and the availability of instruction issue slots, in light of three compiler optimizations they affect.

Inter-thread data sharing

Conventional parallelization techniques target multiprocessors, in which threads are physically distributed on different processors. To minimize cache coherence and inter-processor communication overhead, data and loop distribution techniques partition and distribute data to match the physical topology of the multiprocessor. Parallelizing compilers attempt to decompose applications to minimize synchronization and communication between loops. Typically, this is achieved by allocating a disjoint set of data for each processor, so that they can work independently [34][10][7].

In contrast, on an SMT, multiple threads execute on the same processor, affecting performance in two ways. First, both real and false inter-thread data sharing entail *local* memory accesses and incur no coherence overhead, because of SMT's shared L1 cache. Consequently, sharing, and even false sharing, is beneficial. Second, by sharing data among threads, the memory footprint of a parallel application can be reduced, resulting in better cache and TLB behavior. Both factors suggest a loop distribution policy that clusters, rather then separates, data for multiple threads.

Latency-hiding capabilities and the availability of instruction issue slots

On most workloads, wide-issue processors typically cannot sustain high instruction throughput, because of low instruction-level parallelism in their single, executing thread. Compiler optimizations, such as software speculative execution and loop tiling (or blocking), try to increase ILP (by hiding or reducing instruction latencies, respectively), but often with the side effect of increasing the dynamic instruction count. Despite the additional instructions, the optimizations are often profitable, because the instruction overhead can be accommodated in otherwise idle functional units.

Because it can issue instructions from multiple threads, an SMT processor has fewer empty issue slots; in fact, sustained instruction throughput can be rather high, roughly 2 times greater than on a conventional superscalar [13]. Furthermore, SMT does a better job of hiding latencies than single-threaded processors, because it uses instructions from one thread to mask delays in another. In such an environment, the aforementioned optimizations may be less useful, or even detrimental, because the overhead instructions compete with useful instructions for hardware resources. SMT, with its simultaneous multithreading capabilities, naturally tolerates high latencies *without* the additional instruction overhead.

4 Methodology

Before examining the compiler optimizations, we describe the methodology used in the experiments. We chose applications from the SPEC 92 [12], SPEC 95 [30] and SPLASH-2 [35] benchmark suites (Table 2). All programs were compiled with the Multiflow trace scheduling compiler [22] to generate DEC Alpha object files. Multiflow was chosen, because it generates high-quality code, using aggressive static scheduling for wide-issue, loop unrolling, and other ILP-exposing optimizations. Implicitly-parallel applications (the SPEC suites) were first parallelized by the SUIF compiler [15]; SUIF's C output was then fed to Multiflow.

A blocked loop distribution policy commonly used for multiprocessor execution has been implemented in SUIF; because we used applications compiled with the latest version of SUIF [5], but did not have access to its source, we implemented an alternative algorithm (described in Section 5) by hand. SUIF also finds tileable loops, determines appropriate multiprocessor-oriented tile sizes for particular data sets and caches, and then generates tiled code; we experimented with other tile sizes with manual coding. Speculative execution was enabled/disabled by modifying the Multiflow compiler's machine description file, which specifies which instructions can be moved speculatively by the trace scheduler. We experimented with both statically-generated and profile-driven traces; for the latter, profiling information was generated by instrumenting the applications and then executing them with a training input data set that differs from the set used during simulation.

The object files generated by Multiflow were linked with our versions of the ANL [4] and SUIF runtime libraries to create executables. Our SMT simulator processes these unmodified Alpha executables and uses emulation-based, instruction-level simulation to model in detail the processor pipelines, hardware support for out-

	Application	Data set	Instructions simulated	LD	SSE	T
SPECfp95	applu	33x33x33 array, 2 iterations	272 M	X	X	
	hydro2d	2 iterations	474 M	X	X	
	mgrid	64x64x64 grid, 1 iteration	3.2 B	X	X	
	su2cor	16x16x16x16, vector len. 4K, 2 iterations	5.4 B	X	X	
	swim	512x512 grid, 10 iterations	419 M	X	X	
	tomcatv	513x513 array, 5 iterations	189 M	X	X	
SPLASH-2	fft	64K data points	32 M		X	
	LU	512x512 matrix	431 M		X	
	radix	256K keys, radix 1K, 512K max key value	6 M		X	
	water-nsquared	512 molecules, 3 timesteps	870 M		X	
	water-spatial	512 molecules, 3 timesteps	784 M		X	
SPECint95	compress	train input set	64 M		X	
	go	train input set, 2stone9	700 M		X	
	li	train input set	258 M		X	
	m88ksim	test input set, dhrystone	164 M		X	
	perl	train input set, scrabble	56 M		X	
SPEC92	mxm from NASA7	matrix multiply of 256x128 and 128x64 arrays	29 M			X
	gmt from NASA7	500x500 Gaussian elimination	354 M			X
	adi integration	1Kx1K stencil computation for solving partial differential equations	16 M			X

Table 1: Benchmarks. The last three columns identify the studies in which the applications are used. (LD = loop distribution, SSE = software speculative execution, and T = tiling).

	L1 I-cache	L1 D-cache	L2 cache
Cache size (bytes)	128K / 32K	128K / 32K	16 M / 2 M
Line size (bytes)	64	64	64
Banks	8	8	1
Transfer time/bank	1 cycle	1 cycle	4 cycles
Accesses/cycle	2	4	1/4
Cache fill time (cycles)	2	2	8
Latency to next level	10 / 18	10 / 18	68

Table 2: Memory hierarchy parameters. When there is a choice of values, the first (the more aggressive) represents a forecast for an SMT implementation roughly three years in the future and is used in all experiments. The second set is more typical of today's memory subsystems and is used to emulate larger data set sizes [29]; it is used in the tiling studies only.

of-order execution, and the entire memory hierarchy, including TLB usage. The memory hierarchy in our processor consists of two levels of cache, with sizes, latencies, and bandwidth characteristics, as shown in Table 2. We model the cache behavior, as well as bank and bus contention. Two TLB sizes were used for the loop distribution experiments (48 and 128 entries), to illustrate how the performance of loop distribution policies is sensitive to TLB size. The larger TLB represents a probable configuration for a (future) general-purpose SMT; the smaller is more appropriate for a less aggressive design, such as an SMT multimedia co-processor, where page sizes are typically in the range of 2-8MB. For both TLB sizes, misses require two full memory accesses, incurring a 160 cycle penalty. For branch prediction, we use a McFarling-style hybrid predictor with a 256-entry, 4-way set-associative branch target buffer, and an 8K entry selector that chooses between a global history predictor (13 history bits) and a local predictor (a 2K-entry local history table that indexes into a 4K-entry, 2-bit local prediction table) [24].

Because of the length of the simulations, we limited our detailed simulation results to the parallel computation portion of the applications (the norm for simulating parallel applications). For the initialization phases of the applications, we used a fast simulation mode that only simulates the caches, so that they were warm when the main computation phases were reached. We then turned on the detailed simulation model.

5 Loop distribution

To reduce communication and coherence overhead in distributed-memory multiprocessors, parallelizing compilers employ a *blocked* loop parallelization policy to distribute iterations across processors. A blocked distribution assigns each thread (processor) continuous array data and iterations that manipulate them (Figure 1). Figure 2 presents SMT speedups for applications parallelized using a blocked distribution with two TLB sizes. Good speedups are obtained for many applications (as the number of threads is increased), but in the smaller TLB the performance of several programs (hydro2d, swim, and tomcatv) degrades with 6 or 8 threads. The 8-thread case is particularly important, because most applications will be parallelized to exploit all 8 hardware contexts in an SMT. Analysis of the simulation bottleneck metrics indicated that the slowdown was the result of thrashing in the data TLB, as indicated by the TLB miss rates of Table 3.

The TLB thrashing is a direct result of blocked partitioning, which increases the total working set of an application because threads work on disjoint data sets. In the most severe cases, each of the 8 threads requires many TLB entries, because loops stride through several large arrays at once. Since the primary data sets are usually larger than a typical 8KB page size, at least one TLB entry is required for each array.

The swim benchmark from SPECfp95 illustrates an extreme example. In one loop, 9 large arrays are accessed

a) original loop

```
for (j = 1; j < n; j++)
  for (i = 0; i < m; i++)
    u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

b) blocked parallelization

```
for (j = max(n * myid / numthreads + 1, 1);
     j < min(n * (myid+1) / numthreads + 1, n);
     j++)
  for (i = 0; i < m; i++)
     u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

c) cyclic parallelization

```
for (j = myid + 1; j < n; j += numthreads)
  for (i = 0; i < m; i++)
     u1[j][i] = u2[j][i] + t1/8.0 * Z[j][i]
```

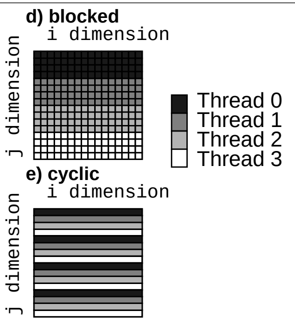

Figure 1: A blocked and cyclic loop distribution example. The code for an example loop nest is shown in a). When using a blocked distribution, the code is structured as in b). The cyclic version is shown in c). On the right, d) and e) illustrate which portions of the array are accessed by each thread for the two policies. (For clarity, we assume 4 threads). Assume that each row of the array is 2KB (512 double precision elements). With blocked distribution (d), each thread accesses a dif ferent 8KB page in memory . With cyclic (e), however, the loop is decomposed in a manner that allows all four threads to access a single 8KB page at the same time, thus reducing the TLB footprint.

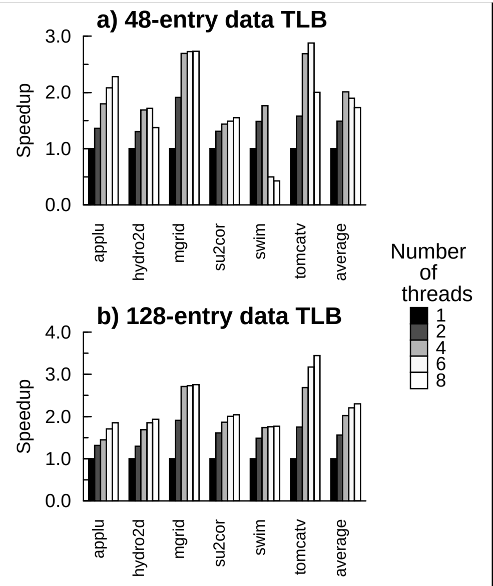

Figure 2: Speedups over one thread for blocked parallelization.

on each iteration of the loop. When the loop is parallelized using a blocked distribution, the data TLB footprint is 9 arrays * 8 threads = 72 TLB entries, excluding the entries required for other data. With any TLB size less than 72, significant thrashing will occur and the parallelization is not profitable.

The lesson here is that the TLB is a shared resource that needs to be managed efficiently in an SMT. At least three approaches can be considered: (1) using fewer than 8 threads when parallelizing, (2) increasing the data TLB size, or (3) parallelizing loops differently.

The first alternative unnecessarily limits the use of the thread hardware contexts, and neither exploits SMT nor the parallel applications to their fullest potential. The second choice incurs a cost in access time and hardware, although with increasing chip densities, future processors may be able to accommodate. [1] Even with larger TLBs, however, it is desirable to reduce the TLB footprint on an SMT. A true SMT workload would be multiprogrammed: for example, multiple parallel applications could execute together, comprising more threads than hardware contexts. The thread scheduler could schedule all 8 threads for the first parallel application, then context switch to run the second, and later switch back to the first. In this type of environment it would be performance-wise to minimize the data TLB footprint required by each application. (As an example, the TLB footprint of a multiprogrammed workload consisting of swim and hydro2d would be greater than 128 entries.)

The third and most desirable solution relies on the compiler to reduce the data TLB footprint. Rather than distributing loop iterations in a blocked organization, it could use a *cyclic* distribution to cluster the accesses of multiple threads onto fewer pages. (With cyclic partitioning, swim would consume 9 rather than 72 TLB entries). Cyclic partitioning also requires less instruction overhead in calculating array partition bounds, a non-negligible, although much less important factor. (Compare the blocked and cyclic loop distribution code and data in Figure 1.)

Figure 3 illustrates the speedups attained by a cyclic distribution over blocked, and Table 4 contains the corresponding changes in data TLB miss rates. With the 48-entry TLB all applications did better with a cyclic distribution. In most cases the significant decrease in data TLB misses, coupled with the long 160 cycle TLB miss penalty, was the major factor. Cyclic increased TLB conflicts in tomcatv at 2 and 4 threads, but, because the number of misses was so low, overall program performance did not suffer. At 6 and 8 threads, tomcatv's

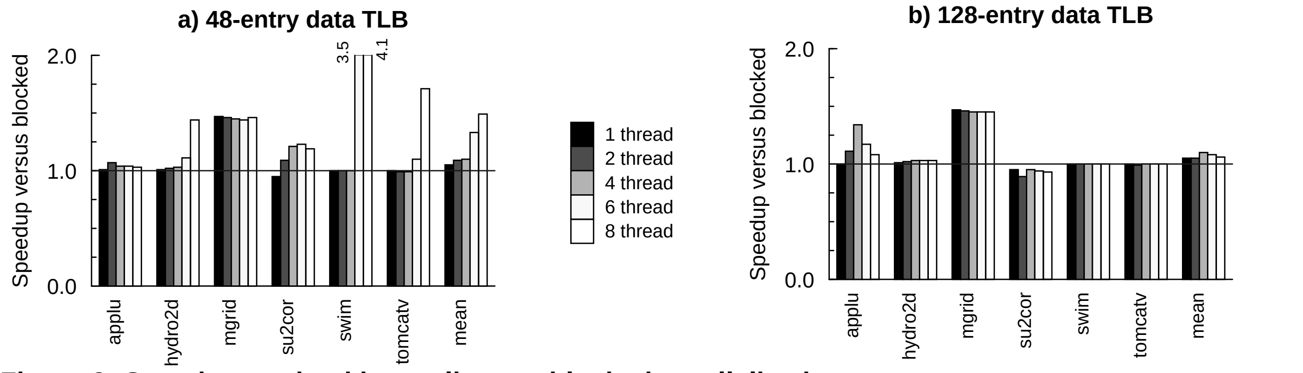

Figure 3: Speedup attained by cyclic over blocked parallelization. For each application, the execution time for blocked is normalized to 1.0 for all numbers of threads. Thus, each bar compares the speedup for cyclic over blocked with the same number of threads.

Application	48-entry TLB					128-entry TLB				
	Number of threads					Number of threads				
	1	2	4	6	8	1	2	4	6	8
applu	0%	50%	58%	53%	15%	0%	91%	98%	85%	69%
hydro2d	0%	0%	14%	91%	99%	0%	0%	0%	0%	14%
mgrid	0%	0%	0%	0%	50%	0%	0%	0%	0%	0%
su2cor	14%	99%	99%	99%	97%	0%	0%	98%	91%	94%
swim	0%	0%	0%	99%	99%	0%	0%	0%	0%	0%
tomcatv	0%	-60%	-60%	96%	99%	0%	-60%	-60%	-60%	-60%

Table 4: Improvement (decrease) in TLB miss rates of cyclic distribution over blocked.

blocked data TLB miss rate jumped to 2% and 11%, causing a corresponding hike in speedup for cyclic.

Absolute miss rates in the larger data TLB are low enough (usually under 0.2%, except for applu and su2cor, which reached 0.9%) that most changes produced little or no benefit for cyclic. In contrast, su2cor saw degradation, because cyclic scheduling increased loop unrolling instruction overhead. This performance degradation was not seen with the smaller TLB size, because cyclic's improved TLB hit rate offset the overhead.

Mgrid saw a large performance improvement for both TLB sizes, because of a reduction in dynamic instruction count. As Figures 1b and 1c illustrate, cyclic parallelization requires fewer computations and no long-latency divide.

In summary, these results suggest using a cyclic loop distribution for SMT, rather than the traditional blocked distribution. For parallel applications with large data footprints, cyclic distribution increased program speedups. (We saw speedups as high as 4.1, even with the smallish SPECfp95 reference data sets.) For applications with smaller data footprints, cyclic broke even. Only in one application, where there was an odd interaction with the loop unrolling factor, did cyclic worsen performance.

In a multiprocessor of SMT processors, a cyclic distribution would still be appropriate within each node. A hybrid parallelization policy might be desirable, though, with a blocked distribution across processors to minimize inter-processor communication.

6 Software speculative execution

Today's optimizing compilers rely on aggressive code scheduling to hide instruction latencies. In global scheduling techniques, such as trace scheduling [22] or hyperblock scheduling [23], instructions from a predicted branch path may be moved above a conditional branch, so that their execution becomes speculative. If at runtime, the other branch path is taken, then the speculative instructions are useless and potentially waste processor resources.

On in-order superscalars or VLIW machines, software speculation is necessary, because the hardware provides no scheduling assistance. On an SMT processor (whose execution core is an out-of-order superscalar), not only are instructions dynamically scheduled and speculatively executed by the hardware, but multithreading is also used to hide latencies. (As the number of SMT threads is increased, instruction throughput also increases.) Therefore, the latency-hiding benefits of software speculative execution may be needed less, or even be unnecessary, and the additional instruction overhead introduced by incorrect speculations may degrade performance.

Our experiments were designed to evaluate the appropriateness of software speculative execution for an SMT processor. The results highlight two factors that determine its effectiveness for SMT: static branch prediction accuracy and instruction throughput.

Correctly-speculated instructions have no instruction overhead; incorrectly-speculated instructions, however, add to the dynamic instruction count. Therefore, speculative execution is more beneficial for applications that have high speculation accuracy, e.g., loop-based programs with either profile-driven or state-of-the-art static branch prediction.

Table 5 compares the dynamic instruction counts between (profile-driven)[2] speculative and non-speculative versions of our applications. Small increases in the dynamic instruction count indicate that the compiler (with the assistance of profiling information) has been able to accurately predict which paths will be executed.[3] Consequently, speculation may incur no penalties. Higher increases in dynamic instruction count, on the other hand, mean wrong-path speculations, and a probable loss in SMT performance.

While instruction overhead influences the effectiveness of speculation, it is not the only factor. The level of instruction throughput in programs without speculation is also important, because it determines how easily speculative overhead can be absorbed. With sufficient instruction issue bandwidth (low IPC), incorrect speculations may cause no harm; with higher

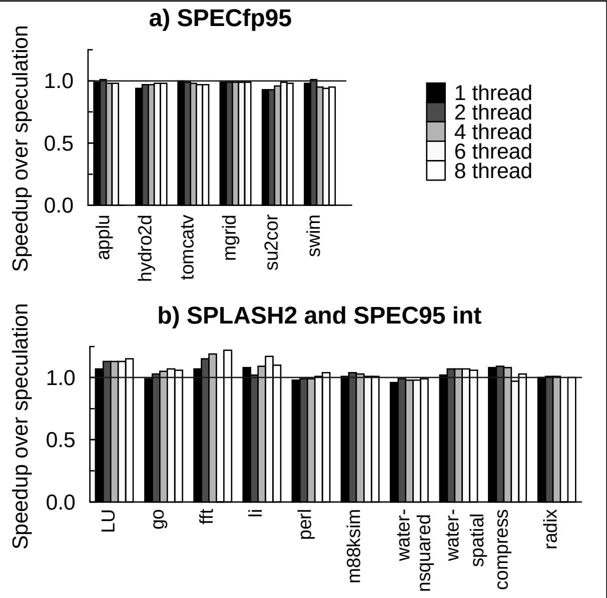

Figure 4: Speedups of applications executing without software speculation over with speculation (speculative execution cycles / no speculation execution cycles). Bars that are greater than 1.0 indicate that no speculation is better.

per-thread ILP or more threads, software speculation should be less profitable, because incorrectly-speculated instructions are more likely to compete with useful instructions for processor resources (in particular, fetch bandwidth and functional unit issue). Table 6 contains the instruction throughput for each of the applications. For some programs IPC is higher with software speculation, indicating some degree of absorption of the speculation overhead. In others, it is lower, because of additional hardware resource conflicts, most notably L1 cache misses.

Speculative instruction overhead (related to static branch prediction accuracy) and instruction throughput *together* explain the speedups (or lack thereof) illustrated in Figure 4. When both factors were high (the non-loop-based fft, li, and LU), speedups without software speculation were greatest, ranging up to 22%.[4] If one factor was low or only moderate, speedups were minimal or nonexistent (the SPECfp95 applications, radix and water-nsquared had only high IPC; go, m88ksim and perl had only speculation overhead).[5] Without either factor, software speculation helped performance, and for the same reasons it benefits other architectures -- it hid latencies and executed the speculative instructions in otherwise idle functional units.

The bottom line is that, while loop-based applications should be compiled with software speculative execution, non-loop applications should be compiled *without* it. Doing so either improves SMT program performance or maintains its current level -- performance is never hurt.[6]

7 Loop tiling

In order to improve cache behavior, loops can be tiled to take advantage of data reuse. In this section, we examine two tiling issues: tile size selection and the distribution of tiles to threads.

If the tile size is chosen appropriately, the reduction in average memory access time more than compensates for the tiling overhead instructions [20][11][6]. (The code in Figures 6b and 6c illustrates the source of this overhead). On an SMT, however, tiling may be less beneficial. First, SMT's enhanced latency-hiding capabilities may render tiling unnecessary. Second, the additional tiling instructions may increase execution time, given SMT's higher (multithreaded) throughput. (These are the same factors that influence whether to software speculate.)

To address these issues, we examined tileable loop nests with different memory access characteristics, executing on an SMT processor. The benefits of tiling vary when the size of the cache is changed. Smaller caches require smaller tiles, which naturally introduce more instruction overhead. On the other hand, smaller tiles also produce lower average memory latencies -- i.e., fewer conflict misses -- so the latency reducing benefit of tiling is better. We therefore varied tile sizes to measure the performance impact of a range of tiling overhead. We also simulated two memory hierarchies to gauge the interaction between cache size, memory latency and tile size. The larger memory configuration represents a probable SMT memory subsystem for machines in production approximately 3 years in the future (see Section 4). The other configuration is smaller, modeling today's memory hierarchies, and is designed to provide a more appropriate ratio between data set and cache size, modeling loops with larger, i.e., more realistic, data sets than those in our benchmarks. For these experiments, each thread was given a separate tile (the tiling norm).

Figure 5 presents the performance (total execution cycles, average memory access time, and dynamic instruction count for a range of tile sizes and the larger memory configuration) of an 8-thread SMT execution of each application and compares it to a single-thread run

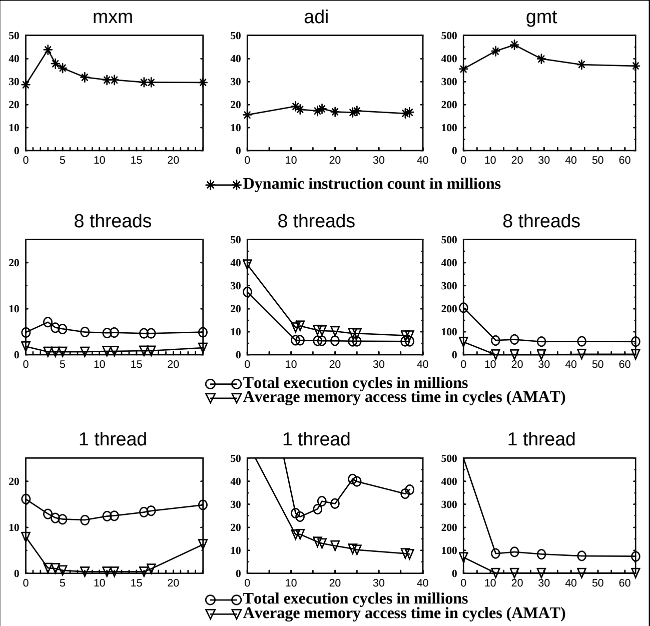

Figure 5: Tiling results with the larger memory subsystem and separate tiles/thread. All the horizontal axes are tile size. A tile size of 0 means no tiling; sizes greater than 0 are one dimension of the tile, measured in array elements. The vertical axes are metrics for evaluating tiling: dynamic instruction count, total execution cycles and AMAT.

(approximating execution on a superscalar [13]). The results indicate that tiling is profitable on an SMT, just as it is on conventional processors. Mxm may seem to be an exception, since tiling brings no improvement, but it is an exception that shows there is no harm in applying the optimization. Programs executing on an SMT appear to be insensitive to tile size; at almost all tile sizes examined, SMT was able to hide memory latencies (as indicated by the flat AMAT curves), while still absorbing tiling overhead. Therefore SMT is less dependent on static algorithms to determine optimal tile sizes for particular caches and working sets. In contrast, conventional processors are more likely to have a tile size sweet spot. Even with out-of-order execution, modern processors, as well as alternative single-die processor architectures, lack sufficient latency-hiding abilities; consequently, they require more exact tile size calculations from the compiler.

Tile size is also not a performance determinant with the less aggressive memory subsystem (results not shown), indicating that tiling on SMT is robust across

a) original loop
```
for (j = 1; j < N; j++
   for (k = 1; k < L; k++)
      for (i = 1; i < M; i++)
         c[j][k] += a[i][k]*b[j][i];
```
b) blocked tiling
```
ub = min(N*(myid+1)/numthreads+1, N);
lb = max(N*myid/numthreads+1, 1);
for (jT=lb; jT <= ub; jT+=jTsize)
   for (kT=1; kT <= L; kT+=kTsize)
      for (iT=1; iT < M; iT+=iTsize)
         for (j=jT; j <= min(N,jT+jTsize-1);j++)
            for (k=max(1, kT);
                   k <= min(L,kT+kTsize-1); k++)
               for (i=iT; i <= min(M,iT+iTsize-1);i++)
                  c[j][k] += a[i][k]*b[j][i];
```
c) cyclic tiling
```
ub = min(N * (myid + 1)/numthreads + 1, N);
lb = max(N * myid / numthreads + 1, 1);
for (jT = lb; jT <= ub; jT += jTsize)
   for (kT = 1; kT <= L; kT += kTsize)
      for (iT = 1; iT < M; iT += iTsize)
         for (j = jT+myid; j <= min(N,jT+jTsize-1);
                j += numthreads)
            for (k=max(1, kT);
                   k <= min(L,kT+kTsize-1); k++)
               for (i=iT; i <= min(M,iT + iTsize-1);
                      i++)
                  c[j][k] += a[i][k]*b[j][i];
```
Figure 6: Code for blocked and cyclic versions of a tiled loop nest.

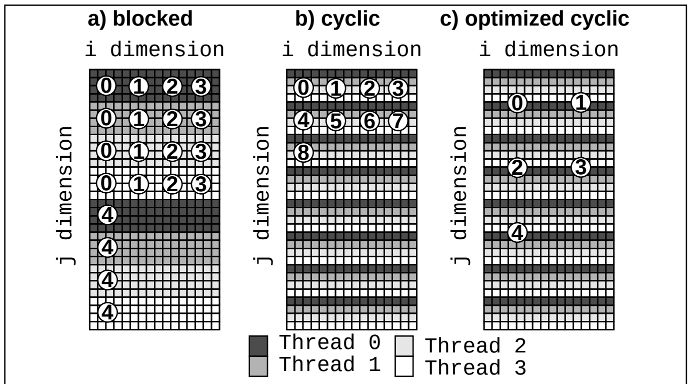

Figure 7: A comparison of blocked and cyclic tiling techniques for multiple threads. The blocked tiling is shown in a). Each tile is a 4x4 array of elements. The numbers represent the order in which tiles are accessed by each thread. For cyclic tiling, each tile is still a 4x4 array, but now the tile is shared by all threads. In this example, each thread gets one row of the tile, as shown in b). With cyclic tiling, each thread works on a smaller chunk of data at a time, so the tiling overhead is greater . In c), the tile size is increased to 8x8 to reduce the overhead. Within each tile, each thread is responsible for 16 of the elements, as in the original blocked example.

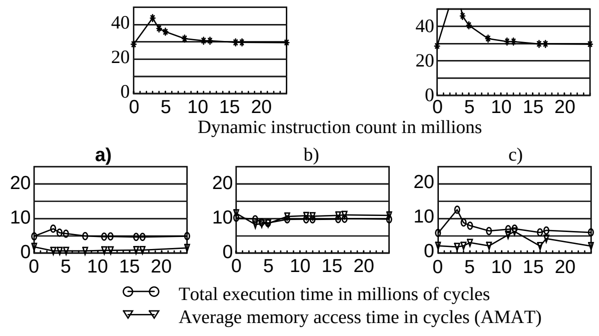

Figure 8: Tiling performance of 8-thread mxm. Tile sizes are along the x-axis. Results are shown for a) blocked tiling and the larger memory subsystem, b) blocked tiling with the smaller memory subsystem, and c) cyclic tiling, also with the smaller memory subsystem.

memory hierarchies (or, alternatively, a range of data set sizes). Execution time is, of course, higher, because performance is more dependent on AMAT parameters, rather than tiling overhead. Only adi became slightly less tolerant of tile size changes. At the largest tile size measured (32x32), its AMAT increased sharply, because of inter-thread interference in the small cache. For this loop nest, either tiles should be sized so that all fit in the cache, or an alternative tiling technique (described below) should be used.

The second loop tiling issue is the distribution of tiles to threads. When parallelizing loops for multiprocessors, a different tile is allocated to each processor (thread) to maximize reuse and reduce inter-processor communication. On an SMT, however, tiling in this manner could be detrimental. Private, per-thread tiles discourage inter-thread tile sharing and increase the total-thread tile footprint on the single-processor SMT (the same factors that make blocked loop iteration scheduling inappropriate for SMT).

Rather than giving each thread its own tile (called *blocked* tiling), a single tile can be shared by all threads, and loop iterations can be distributed cyclically across the threads (*cyclic* tiling). (See Figure 6 for a code explanation of blocked and cyclic tiling, and Figure 7 for the effect on the per-thread data layout).

Because the tile is shared, cyclic tiling can be optimized by increasing the tile size to reduce overhead (Figure 7c). With larger tiles, cyclic tiling can drop execution times of applications executing on small memory SMTs closer to that of SMTs with more aggressive memory hierarchies. (Or, put another way, the performance of programs with large data sets can approach those with smaller.) For example, Figure 8c illustrates that with larger tile sizes (greater than 8 array elements per dimension) cyclic tiling reduced mxm's AMAT enough to decrease average execution time on the smaller cache hierarchy by 51% (compare to blocked in Figure 8b) and to within 35% of blocked tiling on a memory subsystem several times the size (Figure 8a). Only at the very smallest tile size did an increase in tiling overhead overwhelm SMT's ability to hide memory latency.

Cyclic tiling is still appropriate for a multiprocessor of SMTs. A hierarchical [8] or hybrid tiling approach might be most effective. Cyclic tiling could be used to maximize locality in each processor, while blocked tiling could distribute tiles across processors to minimize inter-processor communication.

8 Other compiler optimizations

In addition to the optimizations studied in this paper, compiler-directed prefetching, predicated execution and software pipelining should also be re-evaluated in the context of an SMT processor.

On a conventional processor, compiler-directed prefetching [26] can be useful for tolerating memory latencies, as long as prefetch overhead (due to prefetch instructions, additional memory bandwidth, and/or cache interference) is minimal. On an SMT, this overhead is more detrimental: it interferes not only with the thread doing the prefetching, but also competes with other threads.

Predicated execution [23][16][28] is an architectural model in which instruction execution can be guarded by boolean predicates that determine whether an instruction should be executed or nullified. Compilers can then use if-conversion [2] to transform control dependences into data dependences, thereby exposing more ILP. Like software speculative execution, aggressive predication can incur additional instruction overhead by executing instructions that are either nullified or produce results that are never used.

Software pipelining [9][27][18][1] improves instruction scheduling by overlapping the execution of multiple loop iterations. Rather than pipelining loops, SMT can execute them in parallel in separate hardware contexts. Doing so alleviates the increased register pressure normally associated with software pipelining. Multithreading could also be combined with software pipelining if necessary.

Most of the optimizations discussed in this paper were originally designed to increase single-thread ILP. While intra-thread parallelism is still important on an SMT processor, simultaneous multithreading relies on multiple threads to provide useful parallelism, and throughput often becomes a more important performance metric. SMT raises the issue of compiling for throughput or for a single-thread. For example, from the perspective of a single running thread, these optimizations, as traditionally applied, may be desirable to reduce the thread's running time. But from a global perspective, greater throughput (and therefore more useful work) can be achieved by limiting the amount of speculative work.

9 Related work

The three compiler optimizations discussed in this paper have been widely investigated in non-SMT architectures. Loop iteration scheduling for shared-memory multiprocessors has been evaluated by Wolf and Lam [34], Carr, McKinley, and Tseng [7], Anderson, Amarasinghe, and Lam [3], and Cierniak and Li [10], among others. These studies focus on scheduling to minimize communication and synchronization overhead; all restructured loops and data layout to improve access locality for each processor. In particular, Anderson et al., discuss the blocked and cyclic mapping schemes, and present a heuristic for choosing between them.

Global scheduling optimizations, like trace scheduling [22], superblocks [25] and hyperblocks [23], allow code motion (including speculative motion) across basic blocks, thereby exposing more ILP for statically-scheduled VLIWs and wide-issue superscalars. In their study on ILP limits, Lam and Wilson [19] found that speculation provides greater speedups on loop-based numeric applications than on non-numeric codes, but their study did not include the effects of wrong-path instructions.

Previous work in code transformation for improved locality has proposed various frameworks and algorithms for selecting and applying a range of loop transformations [14][33][6][17][34][7]. These studies illustrate the effectiveness of tiling and also propose other loop transformations for enabling better tiling. Lam, Rothberg, and Wolf [20], Coleman and McKinley [11], and Carr et al., [6] show that application performance is sensitive to the tile size, and present techniques for selecting tile sizes based on problem-size and cache parameters, rather than targeting a fixed-size or fixed-cache occupancy.

10 Conclusions

This paper has examined compiler optimizations in the context of a simultaneous multithreading architecture. An SMT architecture differs from previous parallel architectures in several significant ways. First, SMT threads share processor and memory system resources of a *single* processor on a fine-grained basis, even within a single cycle. Optimizations for an SMT should therefore seek to benefit from this fine-grained sharing, rather than avoiding it, as is done on conventional shared-memory multiprocessors. Second, SMT hides intra-thread latencies by using instructions from other active threads; optimizations that expose ILP may not be needed. Third, instruction throughput on an SMT is high; therefore optimizations that increase instruction count may degrade performance.

An effective compilation strategy for simultaneous multithreading processors must recognize these unique characteristics. Our results show specific cases where an SMT processor can benefit from changing the compiler optimization strategy. In particular, we showed that (1) cyclic iteration scheduling (as opposed to blocked scheduling) is more appropriate for an SMT, because of its ability to reduce the TLB footprint; (2) software speculative execution can be bad for an SMT, because it decreases useful instruction throughput; (3) loop tiling algorithms can be less concerned with determining the exact tile size, because SMT performance is less sensitive to tile size; and (4) loop tiling to increase, rather than reduce, inter-thread tile sharing, is more appropriate for an SMT, because it increases the benefit of sharing memory system resources.

Acknowledgments

We would like to thank John O'Donnell of Equator Technologies, Inc. and Tryggve Fossum of Digital Equipment Corp. for the source to the Alpha AXP version of the Multiflow compiler; and Jennifer Anderson of the DEC Western Research Laboratory for providing us with SUIF-parallelized copies of the SPECfp95 benchmarks. We also would like to thank Jeffrey Dean of DEC WRL and the referees, whose comments helped improve this paper. This research was supported by the Washington Technology Center, NSF grants MIP-9632977, CCR-9200832, and CCR-9632769, DARPA grant F30602-97-2-0226, ONR grants N00014-92-J-1395 and N00014-94-1-1136, DEC WRL, and a fellowship from Intel Corporation.

References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, p. 308–317, June 1988.

[2] J. Allen, et al. Conversion of control dependence to data dependence. In *Conf. Record of the Tenth Ann. ACM Symp. on Principles of Programming Languages*, p. 177–189, January 1983.

[3] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, p. 166–178, July 1995.

[4] J. Boyle, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., 1987.

[5] E. Bugnion, et al. Compiler-directed page coloring for multiprocessors. In *Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, p. 244–255, October 1997.

[6] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92*, p. 114–124, November 1992.

[7] S. Carr, K. S. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, p. 252–262, October 1994.

[8] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of the Ninth Int'l Parallel Processing Symp.*, p. 239–245, April 1995.

[9] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(9):18–27, December 1981.

[10] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, p. 205–217, June 1995.

[11] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, p. 279–290, June 1995.

[12] K. Dixit. New CPU benchmark suites from SPEC. In *COMPCON '92 digest of papers*, p. 305–310, February 1992.

[13] S. J. Eggers, et al. Simultaneous multithreading: A platform for next-generation processors. In *IEEE Micro*, October 1997.

[14] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *J. of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[15] M. W. Hall, et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[16] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *13th Ann. Int'l Symp. on Computer Architecture*, p. 386–395, June 1986.

[17] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing, 6th Int'l Workshop*, p. 301–319. August 1993.

[18] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, p. 318–328, June 1988.

[19] M. Lam and R. Wilson. Limits of control flow on parallelism. In *19th Ann. Int'l Symp. on Computer Architecture*, p. 46–57, May 1992.

[20] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, p. 63–74, April 1991.

[21] J. L. Lo, et al. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. on Computer and Systems*, 15(3), August 1997.

[22] P. G. Lowney, et al. The Multiflow trace scheduling compiler. *J. of Supercomputing*, 7(1/2):51–142, May 1993.

[23] S. A. Mahlke, et al. Effective compiler support for predicated execution using the hyperblock. In *25th Int'l Symp. on Microarchitecture*, p. 45–54, December 1992.

[24] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-Western Research Laboratory, June 1993.

[25] W.W. Hwu, et al. The superblock: An effective technique for VLIW and superscalar compilation. *J. of Supercomputing*, 7(1/2):229–248, May 1993.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, p. 62–75, September 1992.

[27] B. R. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Ann. Workshop on Microprogramming*, p. 183–197, October 1981.

[28] B. R. Rau, et al. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22:12–35, January 1989.

[29] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 27(7):42–50, July 1993.

[30] SPEC. *SPEC CPU '95 Technical Manual*. August 1995.

[31] D. M. Tullsen, et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Ann. Int'l Symp. on Computer Architecture*, p. 191–202, May 1996.

[32] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Ann. Int'l Symp. on Computer Architecture*, p. 392–403, June 1995.

[33] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, p. 30–44, June 1991.

[34] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[35] S. C. Woo, et al. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Ann. Int’l Symp. on Computer Architecture*, p. 24–36, June 1995.