Maximizing TLP with loop-parallelization on SMT

Diego Puppin
Massachusetts Institute of Technology
77 Massachusetts Avenue, NE43-618,
Cambridge, MA, 02139
diego@mit.edu
(617) 253-6284

Dean Tullsen
University of California, San Diego
9500 Gilman Drive
La Jolla, CA, 92093
tullsen@cs.ucsd.edu

Abstract

This paper describes research in exploiting loop-level parallelism on a simultaneous multithreading processor. We discuss some general and ad-hoc techniques for loop parallelization that proved to be effective with SMT, and how they were tuned for it. These techniques have been tested on the well-known Livermore loops, chosen for their variety of behaviors. The set of optimizations used produced significant improvement overall: we were able to improve average IPC from 2.72 to 3.97, and to gain an average speedup of 1.39 over optimized single-thread code, using up to eight threads.

We also describe a simple but effective method for determining the best number of threads to be used for parallel loops on a multithreaded processor. The model uses compile-time information to predict the most efficient point.

Keywords: simultaneous multithreading, loop parallelization, compiling

1 Introduction

The simultaneous multithreading (SMT) processor [12] is a computing paradigm that allows multiple threads to share the processing resources at the level of functional units each cycle. This allows thread-level parallelism to be exploited at a very fine granularity. The role of a parallel compiler for SMT is to extract parallelism, and to tune parallelization to take advantage of its peculiar resource sharing.

Parallelization on a SMT processor presents different challenges than a conventional parallel processor, due to the unique features of the processor. First, because threads share resources at such a fine level, increasing instruction-count to introduce parallelization can actually decrease performance if spare fetch and execution bandwidth is not available. Second, because all execution resources are available to even a single thread, increasing parallelism beyond the level that maximizes instruction-level parallelism, or saturates execution bandwith, is unnecessary and potentially harmful.

Thus, parallelization on SMT requires more careful tuning of parallelism and parallel optimizations, balancing the cost vs. benefit. Also, because thread-level parallelism is not constrained by memory layout (all threads share the same memory hierarchy), the compiler is free to optimize the number of threads used, and the methods of parallelization, independently for each loop in the program.

This paper is structured as follows. Section 2 describes some related work. Section 3 studies the effectiveness of techniques for loop-parallelization. Explored techniques include iteration interleaving, loop fusion, cyclic reduction, loop peeling, loop-invariant code motion, and local accumulation. Section 4 introduces a method for determining the best number of threads for a specific loop. The last section concludes and presents future work.

2 Related Work

In [13], effective techniques for fine-grained synchronization are discussed. SMT offers communication at the level of the L1 cache. The authors explain how to take advantage of this feature to parallelize tight loops that could not be parallelized on conventional parallel machines. Our work leverages some of their results.

In [6], Mitchell *et al.* were able to predict the performance of a few algorithms by measuring parameters such as data and register locality. However, prediction was strictly problem-dependent, and required a time-consuming data-fitting process. Because we limit our attention to determining the best number of threads, the method proposed here is simpler.

They also demonstrate the complex interactions between ILP-enhancing compiler optimizations and threading, also confirming that too much parallelization is not beneficial after the processor is saturated.

Other techniques that introduce novel approaches to creating thread-level parallelism on a multithreaded processor include slip-streaming [10], and speculative precomputation [2, 14]. However, this paper focuses on more traditional compiler-generated parallelism on a multithreaded processor.

The Cray MTA processor [1] features an advanced threading compiler which is capable of several of the transformations described here; however, the MTA is an LIW, cycle-interleaved multithreading processor; thus, it represents a significantly different execution model with less intimate sharing of and competition for resources between threads.

3 Effectiveness of loop-parallelization techniques

The first part of this research presents case studies of loop-parallelization with the SMT processor. The Livermore loops have been chosen for this purpose because of their wide availability, their well-known features, and their generality and variety. We apply some standard and some ad-hoc parallelization techniques to the various kernels in order to understand which are effective, and what is the performance gain. The goal is to show that a compiler can effectively target parallelization on the SMT processor. For the simulations, the parameters used are the same used in [11]. As explained there, these parameters describe a likely next-generation SMT processor, with out-of-order instruction execution, 8-wide fetch and execute, 2-level on-chip caches, and hardware support for 8 threads. In some instances, we also simulate 16 threads to verify that some benchmarks have optimal points beyond the limits of our machine.

Parallelization was performed at the high-level code (C source). All the kernels were rewritten manually, using general principles. We tried to emulate the work of an advanced compiler in all of our transformations.

Parallelization techniques used here include general loop restructuring, such as loop fusion, loop peeling, invariant code motion, and some more advanced techniques aimed at this architecture. These include interleaving, cyclic reduction, and local accumulation.

In the following analysis, two principal metrics will be discussed: processor utilization and completion time. Even if completion time is the most important

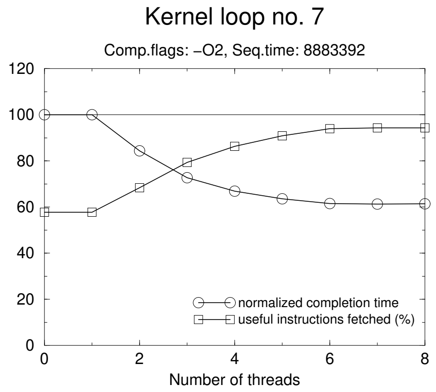

Figure 1: Execution statistics for kernel 7

value when discussing the effect of multithreading, processor utilization is a useful description of how well processor resources are exploited: we cannot expect very high improvement in terms of completion time if processor utilization is high for the sequential version. Processor utilization is measured as the average percentage of useful instructions fetched relative to the total bandwidth (8 instructions per cycle). Completion time is measured as a percentage of the time taken by the sequential version. For both metrics, the values corresponding to 0 threads in the plots refer to the sequential version, as opposed the the single-thread version of the parallel code, which is the 1-thread result.

All simulation was done using the SMTSIM [11] simulator, running Alpha executables and compiled with gcc at the highest level of optimization.

3.1 Independent iterations

The Livermore loop kernels can be classified into four groups: independent-iteration loops, loop-carried dependence loops, accumulation loops, and large loops. This and the following sections will present overall results for all the loops of each type, as well as specific discussion of loops that either had typical or noteworthy behavior.

Loops with independent iterations are basically vector computations that can be carried on independently for every element. They are easy to parallelize, using *iteration interleaving.* This consists in assigning iterations to threads not in blocks, but interleaved. As shown in [5], in these cases this is the most efficient way to express parallelism: better cache and TLB utilization is reached with this solution.

These kind of loops are easily run on the SMT processor, with good speedup. Figure 1 shows statitics

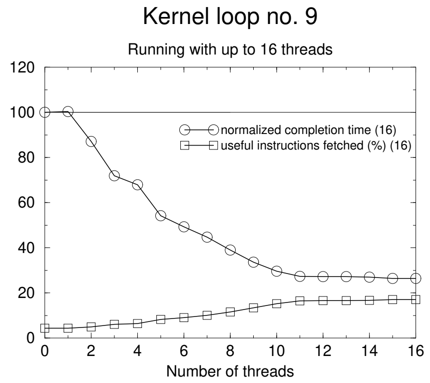

Figure 2: Execution statistics for kernel 9

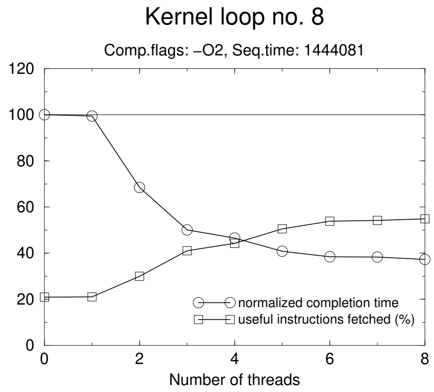

Figure 3: Execution statistics for kernel 8

for kernel 7 as an example. Completion time asymptotically decreases to 60% of sequential time, achieving much higher processor utilization. We cannot expect any more improvement, as almost all the processor bandwidth (92%) is taken by useful instructions. On average, these loops, when parallelized achieve about 1.5 speedup.

The highest available parallelism is found in kernel 9. This code suffers from very bad Dcache utilization (only 70% to 80% Dcache hit ratio) and very high average memory delay (116.5 cycles). When more threads are running, data are moved into the shared cache by the first thread: the other threads will find their data ready in the cache, due to the interleaving, with an effect similar to prefetching. The 8-thread version has an average memory delay of just 46.2 cycles, and runs 2.5 times faster. In figure 2, results are shown for up to 16 threads, which will be discussed further in section 4.

We should note that loop 8 is in this category (independent iterations) with some compiler assistance. In this kernel, some temporary values are stored tidily in a vector, but are never used outside the iteration that created them. We assume that a compiler can determine this fact using some simple techniques (comparing indices...), and then introduce suitable temporary variables, local to each iteration. This transformation makes iterations independent. This change proved to be effective with sequential code as well. The multithreaded code is more than twice as fast. Figure 3 shows this result, with the non-multithreaded code also taking advantage of this optimization.

3.2 Loop-carried dependence loops

Loops featuring loop-carried data dependences (Livermore loops 5, 11, 19, 20, 23) are more difficult to parallelize: interleaving is not useful here, as iterations need to be executed strictly in order. Also, loop skewing failed due to high overhead.

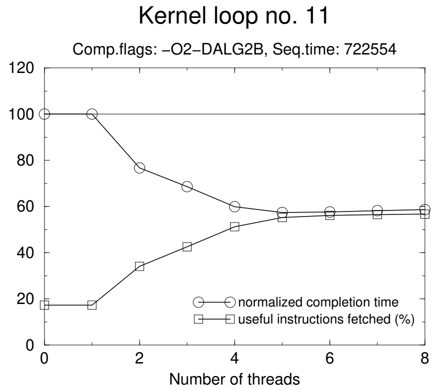

Figure 4: Execution statistics for kernel 11

Nonetheless, kernel 11 was successfully parallelized using *cyclic reduction* [4], a powerful algorithm for the running sum problem, which was tuned to SMT by reducing the level of recursion to 2.

The initial low processor utilization on this kernel offers large opportunity to introduce advanced techniques. Even if more instructions are executed, about 1.7 times as much in this case, the increase in available TLP allows much better instruction throughput, overcoming the large overhead. The multithreaded version is almost twice as fast (see figure 4). We expect this optimization to be very useful with other kinds of loops.

3.3 Accumulation loops

Livermore kernels no. 3, 4, 6, and 13 are particularly interesting, as they feature some independent

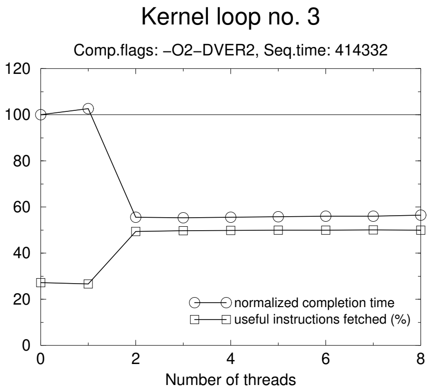

Figure 5: Execution statistics for kernel 3

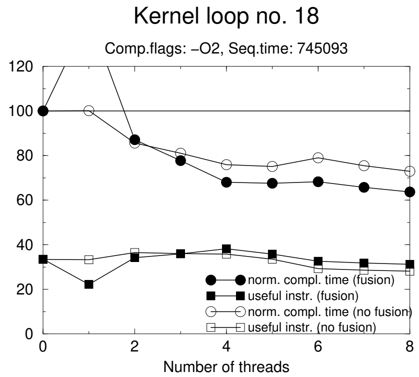

Figure 6: Execution statistics for kernel 18

computation, followed by accumulation of all the values. So, one part of the body could be easily parallelized among threads (interleaving), while accumulation was protected by suitable (ordered) locking to prevent critical races.

To manage these cases, we introduce a technique called *local accumulation*. If the accumulation is carried on by an associative and commutative function, the order is not important: every thread can compute a local summation, which at the end takes part in the global sum, performed by one specific thread.

Figure 5 shows typical behavior for this category (in this case, loop 3). The two-thread version presents a boost in performance with respect to sequential code (1.6 speedup), but after this, the global sum introduces sequentiality, which makes further threads useless. We verified that a tree reduction for summation is not effective, as its cost overwhelms the increased ILP.

It is interesting to note that the optimizations introduced are effective sometimes even if just one thread is running. The MT version of kernel 6 with one thread runs faster than the original sequential version (see table 1). In this case, we introduced a temporary variable to store the local summation, the value of which was then summed to the final results. This transformation allowed better register allocation and memory usage even when no actual TLP was exploited.

3.4 Larger loops

Five of the 24 Livermore loops featured larger, more complex loops, with complex branching and data-dependent memory-accesses. In these cases, the general technique was to interleave iterations, introducing ordered locking to protect possible dependences. We expect that a compiler will not be able to improve the performance significantly due to the difficulties of analyzing the complex code. In these cases, the programmer may be able to give more directions to the compiler, e.g. by augmenting the code with some compilation directives.

Nonetheless, some other standard techniques proved useful in some cases. The three loops composing kernel 18 are just a smart splitting of a larger loop, to increase ILP. When loops are fused, more opportunity for thread-level parallelism emerges, which is exploited when more threads are used. This solution is not good with few threads, because the increased ILP provided by loop distribution is greater than the TLP provided by loop fusion.

In figure 6, a comparison between the two versions (loop fusion and loop distribution) is given. To achieve the best performance at any number of threads, the compiler would have to produce multiple versions of the code, which could be selected at runtime based on the number of hardware contexts available. This idea is gaining importance for traditional processors also, under the broader denomination of *feedback-directed optimization* [7].

Another note is about kernel 24. This loop scans a vector looking for the first minimum. It was parallelized successfully (see figure 7) considering the minimum as an accumulation, using local variables to store the local minimum for each thread. This type of restructuring can be debatable, as this requires the compiler to recognize that the test `x[k] < x[m]` is a way to compute an accumulation function, but we believe that the usage of a library function `min` could make this transformation automatic (some libraries, such as *MPI*, feature specific parallel implementations for the minimum).

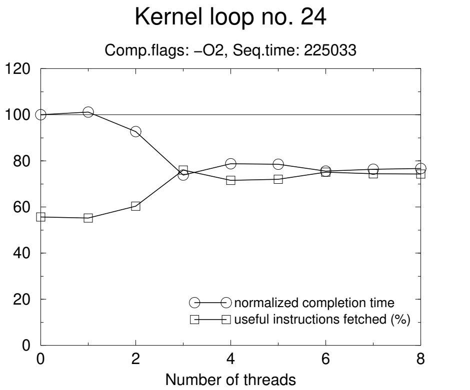

Figure 7: Execution statistics for kernel 24

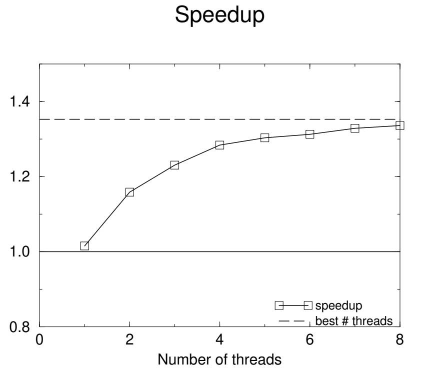

Figure 8: Average speedup for the Livermore loops

3.5 Overall speedup

In most cases, good speedups were achieved by applying varied optimization techniques appropriate to each loop. Overall results obtained with these techniques are given in figures 8, where the dashed line represents the speedup that can be reached choosing the best number of threads independently for every kernel. With the techniques discussed here, we were able to achieve significant speed-up: IPC is increased on average from 2.72 to 3.97, and completion time exhibits speedups averaging 1.39.

In this work, we assume a low-cost thread spawning mechanism, or that, in parallel code, threads available for parallel execution would be waiting on synchronization variables between parallel loops. This latter model can be implemented with low overhead, due to the fast on-chip communication offered by SMT.

Table 1 shows the results with more details. The *type* column represents the category in which the loop falls: independent iterations (IND), loop-carried dependences (DEP), accumulation (ACC) and large loops (LRG). The table reports completion time and average speedup for the parallel code with 1, 2, 4 and 8 threads (results for the other numbers of threads are not shown for simplicity), normalized with respect to the sequential time, set equal to 100. We also set completion time equal to 100 if the MT code was actually slower, as we can always run sequential code if needed. The table then reports the completion time and average speedup reached with the best number of threads. It shows also IPC for the sequential code and useful IPC for the best number of threads, that is computed as:

$$useful\ IPC = sequential\ IPC * speedup$$

loop	type	1	2	4	8	best	IPC	MT IPC
1	IND	100	100	100	100	100	5.32	5.34
2	LRG	100	100	100	100	100	6.10	6.10
3	ACC	100	56	56	57	55	2.18	3.94
4	ACC	100	65	67	72	65	2.48	3.80
5	DEP	100	100	100	100	100	1.22	1.22
6	ACC	87	68	60	64	60	1.34	2.25
7	IND	100	84	67	61	61	4.62	7.53
8	IND	100	69	47	37	37	1.68	4.50
9	IND	100	98	67	37	37	0.35	0.93
10	IND	100	91	86	90	86	2.56	2.97
11	DEP	100	77	60	59	57	1.39	2.42
12	IND	100	76	76	75	75	4.92	6.56
13	ACC	100	68	41	39	39	1.53	3.95
14	LRG	100	100	84	83	82	3.17	3.86
15	IND	100	82	76	73	72	4.86	6.72
16	LRG	100	100	100	100	100	1.21	1.21
17	IND	100	51	27	25	24	1.68	6.90
18	LRG	100	87	68	64	64	2.68	4.21
19	DEP	100	100	100	100	100	1.01	1.01
20	DEP	100	100	100	100	100	0.70	0.70
21	IND	100	100	97	99	96	6.11	6.37
22	IND	100	84	71	46	46	2.54	5.57
23	DEP	100	100	100	100	100	1.26	1.26
24	LRG	100	93	79	77	74	4.46	6.03
average		99	85	76	73	72	2.72	3.97
speedup %		100.58	117.19	131.28	136.75	138.62		

Table 1: Experimental data for the Livermore loops

This way, we do not count any overhead introduced by multi-threading, we measure only how efficiently the useful instructions of the sequential code were run by the parallel code. In 7 cases we were able to have more than 6 useful IPC, out of a total bandwidth of 8. For instance, while kernel 11 reaches an actual processor utilization of about 60%, i.e. more than 4.5 IPC, useful IPC is only 2.42.

As the reader can observe, the best performance is not always reached with eight threads. For accumulation loops, for instance, a few threads are usually enough. Therefore, it is crucial to be able to determine the right number of threads to best exploit the available resources.

There are two reasons to limit thread use to the optimal point (or below). In some cases, performance decreases significantly when more threads are used. Second, from a system-level view, this presents the system with more options to use the idle threads for other purposes. Even if we determine that an application will saturate the processor (because it saturates a particular resource, for example the floating point execution units), system-level performance could still be improved if other threads are introduced which do not bottleneck on the same resource. Techniques for identifying such threads are discussed in [9].

In the next section, we introduce a method that allows the compiler to determine the optimal number of threads to use for each loop.

4 Determining the best number of threads

Let's call *critical path length* (CPL) the length of the instruction critical path (longest chain of dependent instructions) within a basic block. To compute the CPL, we consider instruction latency. For memory operations, we use average memory access time as measured by the simulator on the sequential version of each kernel.

We wrote a small program that allowed us to collect CPL figures automatically, using Atom [3], a tool able to augment binary Alpha code with analysis routines. Atom is also used to determine which are the most stressed basic blocks, in the discussion that follows.

If the block represents a loop body, a single iteration will take at least CPL cycles. Not all the functional units will typically be used in this time: many of them may remain empty due to low ILP. We can have a performance gain if we can squeeze another copy of the loop body into the schedule, filling the empty slots.

A higher number of threads does not in general guarantee higher performance. Further performance improvement is limited if one of the functional units is *saturated*. In our terminology, this means that the number of instructions of a given type cannot be executed within CPL cycles by the available resources. In particular, we track the number of integer, load-store, synchronization and floating point functional units, as well as total issue bandwidth (five different counts). If a functional unit is not saturated, more copies of the body loop can run together as different SMT threads, as long as iterations can be executed partially or fully in parallel.

$$saturation_i \text{ if } numinsts_i > CPL * bandwidth_i$$

where $bandwidth_i$ is the bandwidth (available functional units) of instructions of type i, and $numinsts_i$ the count of instruction of type i. In this case, we break all instructions into the 5 categories just described.

We make the assumption that the best number of threads is the minimum number that reaches full processor utilization before saturating. Keeping the minimum number, we have a better utilization of shared resources. The best number of copies can be computed as:

$$best\ copies = \begin{cases} CPL * \min\limits_{1..5}\left(\dfrac{bandwidth_i}{numinsts_i}\right) \\ \quad \text{if independent iterations} \\ \\ 1 \quad \text{if dependent iterations} \end{cases}$$

This approach has been tried, with the results shown in table 2.

Refinement of this model to have a more precise approximation of the best number of threads is a continuing direction of this research, but we are encouraged by the success of the initial techniques. Also, the current model manages only those loops (14 out of 24) for which one single basic block is responsible for most of the running time (>80%). Despite its current simplicity, the presented model still enables the following observations:

- For kernels featuring loop-carried dependences, adding more threads is not useful when naïve parallelization is used.
- Very high expected values describe a situation in which increasing the number of threads is useful; particularly interesting is kernel no. 9, which scales up to 16 threads (see figure 2): the model says that it would scale well even to a really higher number of threads.
- Low expected values can be observed when the processor-utilization curve features a minimum; these situations require careful tuning of the number of threads.

The most unexpected result is given by kernel no. 6, which does not scale beyond 4 threads. Its low performance is mostly determined by a very high memory latency, the effects of which are hidden by fewer threads than expected by the model.

Giving the compiler control over the number of threads created, as well as the tools to choose the right number, can enable significantly higher performance than naïvely creating the maximum number of threads. This maximizes both per-application performance and system-level performance (not shown directly in this work) when the other thread contexts are made available for other purposes. This ability is maximized if the processor has the ability to dynamically allocate and deallocate threads during the execution of the program without high overhead.

Kernel	1	3	4	5	6	7	8
Actual	8	2	2	1	4	7	8
Expect.	8.8	5.1	5.1	1	17.4	6.3	5.5

Kernel	9	10	11	12	13	21	23
Actual	8(16)	7	1	2	5	3	1
Expect.	35.3	9.2	1	4	3.3	6.0	1

Table 2: Actual best number of threads compared with expected best number of copies

In this direction, we believe that this model can be used to predict symbiosis [8] effects. The model can tell which units are saturated and which are not, and how many threads are best if the available resources are reduced, in a certain sense predicting the interaction of two multi-threaded programs. If the two programs have different requests and they can run at their best with a limited number of threads, we can expect them to run together effectively. This will be an interesting topic of future research.

5 Conclusion and Future Work

We have shown that standard and ad-hoc parallelization techniques can improve significantly the performance of loops on an SMT processor: IPC increasead on average from 2.72 to 3.97, and completion time achieves an average speedup of 1.39 over the sequential version, using up to eight threads.

The demonstrated optimizations should be within the capabilities of a reasonable compiler. The Cray MTA compiler already does some of these optimizations for that architecture.

We have also shown a model to determine the best number of threads for a given loop. We were able to make predictions about the number that offered the best performance for specific loops, using compile time information. We believe that a compiler featuring such a model can boost the performance by running each loop with the most appropriate number of threads. We also believe that the model can improve system-level performance, predicting which programs can run together with good symbiotic effect.

Future work will be in the direction of developing and implementing an advanced parallelizing compiler for the SMT. This will require support of automatic rewriting for loops, using techniques such as those shown here, and a performance model that best exploits the TLP features of the architecture.

Acknowledgements

The authors would like to thank the reviewers for their useful comments. This work was funded in part by NSF Career award MIP-9701708 and equipment grants from Compaq Computer Corporation.

References

[1] R. Alverson, D. Callahan, D. Cummings, and B. Koblenz. The tera computer system. 1990 International Conference on Supercomputing, September 1990.

[2] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. 28th International Symposium on Computer Architecture, 2001.

[3] Compaq. Alpha c compiler libraries: Atom. Online documentation.

[4] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[5] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. 30th Annual International Symposium on Microarchitecture (Micro-30), December 1997.

[6] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Ilp versus tlp on smt. Supercomputing '99, November 1999.

[7] M. D. Smith. Overcoming the challenges to feedback-directed optimization. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 00), Boston, MA, January 2000.

[8] A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Explorations in symbiosis on two multithreaded architectures. Workshop on Multithreaded Execution, Architecture, and Compilation, January 1999.

[9] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. Architectural Support for Programming Languages and Operating Systems, pages 234–244, November 2000.

[10] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. Architectural Support for Programming Languages and Operating Systems, pages 257–268, November 2000.

[11] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May 1996. Reprinted in Readings in Computer Architecture.

[12] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995. Reprinted in 25 Years of the International Symposia on Computer Architecture: Selected Papers, 1998.

[13] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. 5th International Symposium on High Performance Computer Architecture, January 1999.

[14] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. 28th International Symposium on Computer Architecture, 2001.