Exploring the Capacity of a Modern SMT Architecture to Deliver High Scientific Application Performance *

Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
{valia,anastop,kkourt,nkoziris}@cslab.ece.ntua.gr

Abstract. Simultaneous multithreading (SMT) has been proposed to improve system throughput by overlapping instructions from multiple threads on a single wide-issue processor. Recent studies have demonstrated that heterogeneity of simultaneously executed applications can bring up significant performance gains due to SMT. However, the speedup of a single application that is parallelized into multiple threads, is often sensitive to its inherent instruction level parallelism (ILP), as well as the efficiency of synchronization and communication mechanisms between its separate, but possibly dependent, threads. In this paper, we explore the performance limits by evaluating the tradeoffs between ILP and TLP for various kinds of instructions streams. We evaluate and contrast speculative precomputation (SPR) and thread-level parallelism (TLP) techniques for a series of scientific codes executed on an SMT processor. We also examine the effect of thread synchronization mechanisms on multithreaded parallel applications that are executed on a single SMT processor. In order to amplify this evaluation process, we also present results gathered from the performance monitoring hardware of the processor.

1 Introduction

Despite the efficiency of code optimization techniques and the continued advances in caches, memory latency still dominates the performance of many applications on modern processors. This CPU-memory gap seems difficult to be alleviated; on the one hand, CPU clock speeds continue to advance more rapidly than memory access times, on the other hand, the data working sets increase and complexity of conventional applications sets a limit on ILP.

One approach to maintain high throughput of processors despite the large relative memory latency has been Simultaneous Multithreading (SMT). SMT is a hardware technique that allows a processor to issue and execute instructions from multiple independent threads in the same cycle. The dynamic sharing of

* This research is supported by the Pythagoras II Project (EPEAEK II), co-founded by the European Social Fund (75%) and National Resources (25%).

the functional units allows for the substantial increase of throughput, compensating for the two major impediments to processor utilization - long latencies and limited per-thread parallelism.

Thread-level parallelism (TLP) and speculative precomputation (SPR) have been proposed to utilize the multiple hardware contexts of the processors for improving performance of a single program. With TLP, sequential codes are parallelized so that the total amount of work is decomposed into independent parts which are assigned to a number of software threads for execution. In SPR, the execution of programs is facilitated with the introduction of additional threads, which speculatively prefetch data that is going to be used by the sibling computation threads in the near future, thus hiding memory latencies and reducing cache misses [13], [5], [10].

The benefit of multithreading on SMT architectures depends on the application and its level of tuning. In this paper we demonstrate that significant performance improvements are really difficult to achieve for optimized, fine-tuned parallel applications running on SMT processors. We tested two different configurations. Firstly, we balanced the computational workload of a parallel benchmark on two threads, statically partitioning the iteration space to minimize dynamic scheduling overhead. Secondly, we ran a main computation thread in parallel with a helper-prefetching thread. The latter was spawned to speculatively precompute L2 cache misses. Synchronization of the two threads is essential, in order to avoid the helper thread from running too far ahead, evicting useful data from the cache.

The rest of the paper is organized as follows. Section 2 describes related prior work. Section 3 deals with implementation aspects of software techniques to exploit hardware multithreading. Section 4 explores the performance limits and TLP-ILP tradeoffs, by considering a representative set of instruction streams. Section 5 describes the experimental framework, presents performance measurements obtained from each application, and discusses their evaluation. Finally, we conclude with section 6.

2 Related Work

SMT [12] is said to outperform previous execution models because it combines the multiple-instruction-issue features of modern superscalar architectures with the latency-hiding ability of multithreaded ones. However, the flexibility of SMT comes at a cost. When multiple threads are active, the static partitioning of resources (e.g., instruction queue, reorder buffer, store queue) affects codes with relative high instruction throughput. Static partitioning, in the case of identical thread-level instruction streams, limits performance, but mitigates significant slowdowns when non-similar streams of microinstructions are executed [11].

Cache prefetching is a technique that reduces the observed latency of memory accesses by bringing data into the cache before it is accessed by the CPU. Numerous thread-based prefetching schemes, either static or dynamic, have recently been proposed, including Collins et al., Speculative Precomputation [3],

and Kim et al., Helper-Threads [5]. The key idea is to utilize otherwise idle hardware thread contexts to execute speculative threads on behalf of the main thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of access by the non-speculative (main) thread, so that the memory miss latency can be masked. A common implementation pattern was used in these studies. A compiler identifies either statically or with the assistance of a profile the memory loads that are likely to cause cache misses with long latencies. Such load instructions, known as delinquent loads, may also be identified dynamically in hardware triggering speculative-helper threads [13]. SPR targets load instructions that exhibit unpredictable irregular, data-dependent or pointer chasing access patterns. Traditionally, these loads have been difficult to handle via either hardware or software prefetchers.

3 SPR and Synchronization Implementation Issues

There are two main issues that must be taken into account in order to effectively perform software prefetching using the multiple execution contexts of a hyper-threaded processor. First of all, the distance at which the precomputation thread runs ahead of the main computation thread, has to be sufficiently regulated. This requirement can be satisfied by imposing a specific upper bound on the amount of data to be prefetched. In our codes it ranges from $\frac{1}{A}$ ([10]) to $\frac{1}{2}$ of the L2 cache size, where A is the associativity of the cache (8 in our case). Whenever this upper bound is reached but the computation thread has not yet started using the prefetched data, the precomputation thread must stop its forward progress in order to prevent potential evictions of useful data from cache. It can only continue when it is signaled that the computation thread starts consuming the prefetched data. In our program codes, this scenario is implemented using synchronization barriers which enclose program regions (precomputation spans) whose memory footprint is equal to the upper bound we have imposed. In the general case, and considering their relatively lightweight workload, precomputation threads reach always first the barriers.

For codes whose access patterns were difficult to determine a-priori, we had to conduct memory profiling using the Valgrind simulator[8]. From the profiling results we were able to determine and isolate the instructions that caused the majority(92% to 96%) of L2 misses. In all cases, precomputation threads were constructed manually from the original code of the main computation threads, preserving only the memory loads that triggered the majority of L2 misses; all other instructions were eliminated.

Secondly, we must guarantee that the co-execution of the precomputation thread does not result in excessive consumption of shared resources that could be critical for the sibling computation thread. Despite the lightweight nature of the precomputation threads, significant processor resources can be consumed even when they are simply spinning on synchronization barriers.

The synchronization mechanisms have to be as lightweight as possible and for this purpose we have implemented lightweight spin-wait loops as the core of our synchronization primitives, embedding the `pause` instruction in the spin loop [4]. This instruction introduces a slight delay in the loop and de-pipelines its execution, preventing it from aggressively consuming valuable, dynamically shared, processor resources (e.g. execution units, branch predictors).

However, some other units (such as micro-ops queues, load/store queues and re-order buffers), are statically partitioned and are not released when a thread executes a `pause`. By using the privileged `halt` instruction, a logical processor can relinquish all of its statically partitioned resources, make them fully available to the other logical processor, and stop its execution going into a sleeping state. The `halt` instruction is primarily intended for use by the operating system scheduler. Multithreaded applications with threads intended to remain idle for a long period, could take advantage of this instruction to boost their execution. We implemented kernel extensions that allow from user space the execution of `halt` on a particular logical processor, and the wake-up of this processor by sending IPIs to it. By integrating these extensions in the spin-wait loops, we are able to construct long duration wait loops that do not consume significant processor resources. Excessive use of these primitives, however, in conjunction with the resultant multiple transitions into and out of the halt state of the processor, incur extra overhead in terms of processor cycles. This is a performance tradeoff that we took into consideration throughout our experiments.

4 Quantitative analysis on the TLP and ILP limits of the processor

This section explores the ability and the limits of hyper-threading technology on interleaving and executing efficiently instructions from two independent threads. We constructed a series of homogeneous instruction streams, which include basic arithmetic operations (add,sub,mul,div), as well as memory operations (load, store), on integer and floating-point 32-bit scalars. For each of them, we tested different levels of instruction level parallelism.

In our experiments, we artificially increase(decrease) the ILP of the stream by keeping the source and target registers always disjoint, and at the same time expanding(shrinking) the target operands (T). We have considered three degrees of ILP for each instruction stream: minimum ($|T|$=1), medium ($|T|$=3), maximum ($|T|$=6).

4.1 Co-executing streams of the same type

As a first step, we execute each instruction stream alone on a single logical processor, for all degrees of ILP (1thr columns of Table 1). In this way, all execution resources of the physical package are fully available to the thread executing that stream. As a second step, we co-execute within the same physical processor two independent instruction streams of the same ILP, each of which

	CPI					
	min ILP		***med ILP***		***max ILP***	
instr.	***1thr***	***2thr***	***1thr***	***2thr***	***1thr***	***2thr***
fadd	6.01	6.03	2.01	3.28	**1.00**	2.02
fmul	8.01	8.04	2.67	4.19	2.01	**3.99**
faddmul	7.01	7.03	2.34	3.83	1.15	**2.23**
fdiv	**45.06**	99.90	45.09	107.05	45.10	107.43
fload	1049.05	2012.62	1049.06	2012.43	1049.05	**2011.86**
fstore	1050.67	1982.99	1050.68	1983.07	1050.67	**1982.93**
iadd	1.01	**1.99**	1.01	2.02	1.00	2.02
imul	11.02	11.05	11.03	**11.05**	11.03	11.05
idiv	76.18	78.76	76.19	**78.71**	76.18	78.73
iload	2.46	4.00	2.46	3.99	2.46	**3.99**
istore	1.93	4.07	1.93	4.08	**1.93**	4.07

Table 1. Average CPI for different TLP and ILP execution modes of some common instruction streams

gets bound to a specific logical processor (2thr columns of Table 1). This gives us an indication on how various kinds of simultaneously executing streams of a specific ILP level, contend with each other for shared resources, and an estimation whether the transition from single-threaded mode of a specific ILP level to dual-threaded mode of a lower ILP level, can hinder or boost performance. For example, let's consider a scenario where, in single-threaded and maximum ILP mode, instruction A gives an average CPI of $C_{1thr-maxILP}$, while in dual-threaded and medium ILP mode the same instruction gives an average CPI of $C_{2thr-medILP} > 2 \times C_{1thr-maxILP}$. Because the second case involves half of the ILP of the first case, the above scenario prompts that we must probably not anticipate any speedup by parallelizing into multiple threads a program that uses extensively this instruction in the context of high ILP (e.g. unrolling). Bold elements of Table 1 indicate best case performance.

4.2 Co-executing streams of different types

Table 2 presents the results from the co-execution of different pairs of streams (for the sake of completeness, results from the co-execution of a given stream with itself, are also presented). We examine pairs whose streams have the same ILP level. The slowdown factor represents the ratio of the CPI when two threads are running concurrently, to the CPI when the benchmark indicated in the fist column is being executed in single-threaded mode. Note that the throughput of integer streams is not affected by variations of ILP and for this reason we present only exact figures of medium ILP. Slowdown factors that vary less than 0.05 compared to the slowdown factor of the medium ILP case in a specific stream combination, are omitted. Bold elements indicate the most significant slowdown factors.

5 Experimental Framework and Results

We experimented on Intel Xeon processor enabled with HT technology, running at 2.8GHz. With the introduction of HT technology, the performance monitoring

		Co-executed Instruction Streams				
	ILP	***fadd***	***fmul***	***fdiv***	***fload***	***fstore***
fadd	min:	1.004	1.004			
	med:	1.635	1.787	1.010	1.398	1.409
	max:	**2.016**	**2.801**	**2.023**	1.474	1.462
fmul	min:	1.002	1.004	1.006		
	med:	1.433	1.566	1.062	1.391	1.393
	max:	1.384	**1.988**			
fdiv	min:			**2.217**		
	med:	1.017	1.027	**2.374**	1.413	1.422
fload	min:	1.144	1.169			
	med:	1.286	1.255	1.153	**1.919**	**1.907**
	max:	1.684	1.358			
fstore	min:	1.134	1.133			
	med:	1.229	1.229	1.150	**1.897**	**1.887**
	max:	1.625	1.316			
	ILP	***iadd***	***imul***	***idiv***	***iload***	***istore***
iadd	med:	**2.014**	1.316	1.117	1.515	1.405
imul	med:	1.116	1.002	1.008	1.003	1.004
idiv	med:	1.042	1.019	1.033	1.003	1.003
iload	med:	**2.145**	0.941	0.934	1.621	1.331
istore	min:	4.072				
	med:	**4.299**	1.979	1.970	1.986	2.115
	max:	2.160	0.941	0.934	1.622	1.331

Table 2. Slowdown factors from the co-execution of various instruction streams

capabilities of the processor were extended, so that the performance counters could be programmed to select events that are qualified by logical processor IDs, whenever that was possible. To use these performance monitoring capabilities, a simple custom library was developed. For each of the multithreaded execution modes presented in section 3 we present measurements taken for three events:

• **L2 Misses**: The number of 2nd level read misses as seen by the bus unit. For the TLP methods, including the prefetch hybrid method the L2 misses presented are the sum of the misses for both threads. For the pure software prefetch method, only the misses of the working thread are presented.

• **Resource stall cycles**: The number of clock cycles that a thread stalls in the processor allocator, waiting until store buffer entries are available. This performance metric is indicative of the contention that exists between hardware threads. For all cases, the results presented correspond to the sum of stall cycles on behalf of both logical processors.

• **μops retired**: The number of μops that were retired during the execution of the program. For all cases the μops number is the number of those retired for both threads.

We have used the NPTL library for the creation and manipulation of threads. Our operating system was Linux version 2.6.9. To force the threads to be scheduled on a particular logical processor within a physical package, we have used the `sched_setaffinity` system call. All user codes were compiled with gcc 3.3.5 compiler using the O2 optimization level, and linked against glibc 2.3.2.

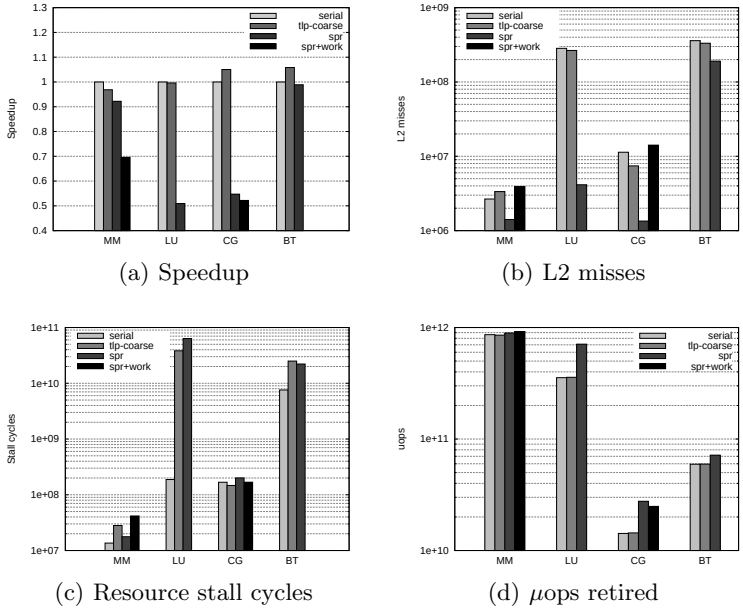
We evaluated performance using two computational kernels, Matrix Multiplication and LU decomposition, and two NAS benchmarks, CG and BT. In MM and LU, we used 4096×4096 matrices, while in CG and BT we considered Class A problem sizes. In MM and LU, we applied tiling choosing tiles that completely fit in L1 cache, since this yielded the best performance. Furthermore, in MM we used blocked array layouts (non-linear layouts) with binary masks [2] and applied loop unrolling. The implementations of CG and BT were based on the OpenMP C versions of NPB suite version 2.3 provided by the Omni OpenMP Compiler Project [1]. We transformed these versions so that appropriate threading functions were used for work decomposition and synchronization, instead of OpenMP constructs. Both CG and BT are characterized by random memory access patterns, with the latter exhibiting somewhat better data locality.

The TLP versions of the codes are based on coarse-grained work partitioning schemes (**tlp-coarse**), where the total amount of work is statically balanced across the participant threads (e.g., different tiles assigned to different threads in MM and LU). The SPR versions use prefetching to tolerate cache misses, following the scheme we described in section 3. In the pure prefetching version (**spr**), the whole workload is executed by just one thread, while the second is just a helper thread that performs prefetching of the next data chunk in issue. In the hybrid prefetching version (**spr+work**), the workload is partitioned in a more fine-grained fashion with both threads performing computations on the same data chunk, while one of them takes on the prefetching of the next data chunk. This latter parallelization scheme was applicable only in MM and CG.

Figure 1 presents the experimental results for the aforementioned benchmarks. HT technology helped us to gain a speedup of $5\% - 6\%$ only in the case of NAS benchmarks, when applying the TLP scheme. In the SPR versions, although a significant reduction in L2 misses of the working thread was achieved in most cases, this was not followed by overall speedup. As Figure 1(d) depicts, in these cases, there was a noticable increase in the total number of μops, as well, due to the instructions required to implement prefetching. For LU and CG, specifically, the total μops were almost double than those of the *serial* case. Since these extra instructions could not be overlapped efficiently with those of the thread performing useful computations, as designated by the increased stall cycles in the *spr* case of all benchmarks compared to their *serial* versions, the reduction of L2 misses itself proved eventually not to be enough for performance improvement.

5.1 Further Analysis

Figure 2 presents the utilization of the busiest processor execution subunits, while running the reference applications. The first column (serial) contains results of the serial versions. The second column (tlp) presents the behavior of one of two threads for the TLP implementation (results for the other thread are identical). The third column (spr) presents statistics of the prefetching thread in the SPR versions. All percentages refer to the portion of the total instructions of each thread that used a specific subunit of the processor. The statistics



(a) Speedup (b) L2 misses

(c) Resource stall cycles (d) μops retired

Fig. 1. Experimental results

were generated by profiling the original application executables using the Pin binary instrumentation tool [6], and analyzing for each case the breakdown of the dynamic instruction mix, as recorded by the tool. Figure 3([4]) presents the main execution units of the processor, together with the issue ports that drive instructions into them. Our analysis examines the major bottlenecks that prevent multithreaded implementations from achieving some speedup.

Compared to the serial versions, TLP implementations do not generally change the mix for various instructions. Of course, this is not the case for SPR implementations. For the prefetcher thread, not only the dynamic mix, but also the total instruction count, differ from those of the worker thread. Additionally, different memory access patterns require incomparable effort for address calculations and data prefetching, and subsequently, different number of instructions.

In the MM benchmark the most specific characteristic is the large number of logical instructions used: at about 25% of total instructions in both serial and TLP versions. This is due to the implementation of blocked array layouts with binary masks that were employed for this benchmark. Although the out-of-order core of the Xeon processor possesses two ALU units (double speed), among them only ALU0 can handle logical operations. As a result, concurrent requests for this unit in the TLP case, will lead to serialization of corresponding instructions, without offering any speedup. In the SPR case of LU, the prefetcher executes at least the same number of instructions as the worker, and also puts the same pressure on ALUs. This is due to the non-optimal data locality, which leads

		Instrumented thread		
	EXECUTION UNIT	***serial***	***tlp***	***spr***
MM	ALU0+ALU1:	27.06%	26.26%	37.56%
	FP_ADD:	11.70%	11.82%	0.00%
	FP_MUL:	11.70%	11.82%	4.13%
	MEM_LOAD:	38.76%	27.00%	58.30%
	MEM_STORE:	12.07%	12.02%	20.75%
	Total instructions:	4590588278	2270133929	202876770
LU	ALU0+ALU1:	38.84%	38.84%	38.16%
	FP_ADD:	11.15%	11.15%	0.00%
	FP_MUL:	11.15%	11.15%	0.00%
	MEM_LOAD:	49.24%	49.24%	38.40%
	MEM_STORE:	11.24%	11.24%	22.78%
	Total instructions:	3205661399	1622610935	3264715031
CG	ALU0+ALU1:	28.04%	23.95%	49.93%
	FP_ADD:	8.83%	7.49%	0.00%
	FP_MUL:	8.86%	7.53%	0.00%
	FP_MOVE:	17.05%	14.05%	0.00%
	MEM_LOAD:	36.51%	45.71%	19.09%
	MEM_STORE:	9.50%	8.51%	9.54%
	Total instructions:	11934228188	7069734891	166842453
BT	ALU0+ALU1:	8.06%	8.06%	12.06%
	FP_ADD:	17.67%	17.67%	0.00%
	FP_MUL:	22.04%	22.04%	0.00%
	FP_MOVE:	10.51%	10.51%	0.00%
	MEM_LOAD:	42.70%	42.70%	44.70%
	MEM_STORE:	16.01%	16.01%	42.94%
	Total instructions:	44973276097	22486809710	8398026979

Fig. 2. Processor subunits utilization from the viewpoint of a specific thread

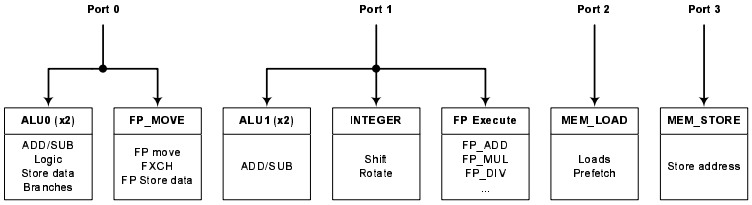


Fig. 3. Instruction issue ports and main execution units of the Xeon processor

prefetcher to execute a large number of instructions to compute the addresses of data to be brought in cache. These facts translate into major slowdowns for the SPR version of LU, despite any significant L2 misses reduction.

As can be seen in Figure 1, TLP mode of BT benchmark was one of few cases that gave us some speedup. The relatively low usage and thus contention on ALUs, in conjunction with non-harmful co-existence of $faddmul$ streams (as Table 2 depicts) which dominate other instructions, and the perfect workload partitioning, are among the main reasons for this speedup.[1]

6 Conclusions

This paper presents performance results for a SMT architecture, the Intel hyper-threaded microarchitecture. We examined scientific codes in which both TLP

and SPR schemes were applied. Our evaluation was based on actual program execution, as well as simulation. The results gathered demonstrated the limits in achieving high performance for such applications.

SPR can achieve a fairly good reduction in L2 cache misses. However, in order to fine tune data prefetching, a considerable number of additional instructions have to be inserted into the pipeline. This increase in the number of μops, in combination with some kind of resource contention, harms performance in terms of execution time. Besides, optimized applications with a relatively high IPC (such as the tested microkernels), are really difficult to achieve even better performance without reducing the μops executed.

Coarse-grained work partitioning schemes do not have a significant impact on the number of μops executed (usually brings a slight increase). Total execution performance would be expected to be improved, especially in cases of L2 cache miss decrease. However, the two working threads, due to their symmetric profiles, compete for the same hardware resources. This contention constitutes in some cases a bottleneck to high performance.

References

1. Omni OpenMP Compiler Project. Released in the International Conference for High Performance Computing, Networking and Storage (SC'03), Nov 2003.
2. E. Athanasaki and N. Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *Proc. of INTERACT'04*, Madrid, Spain.
3. J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of ISCA '01*, Göteborg, Sweden.
4. Intel Corporation. *IA-32 Intel Architecture Optimization*. Order Num: 248966-011.
5. D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In *Proc. of IEEE/ACM CGO 2004*, San Jose, CA.
6. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.
7. D. Marr, F. Desktop, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *ITJ*, Feb 2002.
8. N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proc. of RV'03*, Boulder, CO.
9. D. Patterson and J. Hennessy. *Computer Architecture. A Quantitative Approach*, pages 597–598. Morgan Kaufmann, 3rd edition, 2003.
10. F. Blagojevic T. Wang and D. Nikolopoulos. Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors. In *Proc. of LCR'2004*, Houston, TX.
11. N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of PACT '03*, New Orleans, LA.
12. D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of ISCA '95*, Santa Margherita Ligure, Italy.
13. H. Wang, P. Wang, R. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *ITJ*, Feb 2002.