Exposing Speculative Thread Parallelism in SPEC2000

Manohar K. Prabhu
Stanford University
Computer Systems Laboratory
Stanford, California 94305
mkprabhu@stanford.edu

Kunle Olukotun
Stanford University
Computer Systems Laboratory
Stanford, California 94305
kunle@stanford.edu

ABSTRACT

As increasing the performance of single-threaded processors becomes increasingly difficult, consumer desktop processors are moving toward multi-core designs. One way to enhance the performance of chip multiprocessors that has received considerable attention is the use of thread-level speculation (TLS). As a case study, we manually parallelized several of the SPEC CPU2000 floating point and integer applications using TLS. The use of manual parallelization enabled us to apply techniques and programmer expertise that are beyond the current capabilities of automated parallelizers. With the experience gained from this, we provide insight into ways to aggressively apply TLS to parallelize applications for high performance. This information can help guide future advanced TLS compiler design.

For each application, we discuss how and where parallelism was located within the application, the impediments to extracting this parallelism using TLS, and the code transformations that were required to overcome these impediments. We also generalize these experiences to a discussion of common hindrances to TLS parallelization, and describe methods of programming that help expose application parallelism to TLS systems. These guidelines can assist developers of uniprocessor programs to create applications that can easily port to TLS systems and yield good performance. By using manual parallelization on SPEC2000, we provide guidance on where thread-level parallelism exists in these well known benchmarks, what limits its extraction, how to reduce these limitations and what performance can be expected on these applications from a chip multiprocessor system with TLS.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; C.1.4 [**Processor Architectures**]: Parallel Architectures.

General Terms

Performance, Algorithms, Design, Measurement.

PPoPP'05, June 15-17, 2005, Chicago, Illinois, USA.

Keywords

SPEC CPU2000, thread-level speculation, chip multiprocessors, manual parallel programming, multithreading, feedback-driven optimization.

1. INTRODUCTION

As high-performance uniprocessors have increased in complexity, further improvements in performance have become tremendously difficult to implement. The combined difficulties caused by increasing verification and validation times, wire delays, power dissipation and circuit unreliability have bolstered arguments against continuing to develop such highly centralized architectures. As a result, virtually every major manufacturer of high-performance processors has announced one or more chip multiprocessors (CMPs) [6][9][10][11][15]. Multiple cores will easily provide benefits for multithreaded workloads, but many applications are written for uniprocessors and will therefore not automatically benefit from CMP designs. While these applications could be parallelized into threads, they usually contain data and control flow dependences that render this a daunting task, especially for poorly-understood legacy code designed by previous programmers.

Thread-level speculation (TLS) hardware simplifies this task by providing support for speculative threads, which can execute in parallel, but dynamically roll back and re-execute if dependences exist between threads. Numerous publications have shown that many applications are amenable to parallelization with TLS [16][17][18][23][25][26]. SPEC CPU2000 benchmark applications are commonly used, but few publications provide insight into where and how parallel threads are extracted from the source code. Typical results instead specify only "overview" statistics such as the number of regions parallelized, the percent coverage of the application, the average thread-length and perhaps the functions parallelized.

In this paper, we describe how we manually parallelized several SPEC2000 applications using TLS. This is useful in many ways. The use of manual parallelization meant that we were not constrained by the need to use techniques that can be automated. We made minor modifications to the applications and used programmer expertise to obtain higher performance than current automated tools can generate. By describing the obstacles to parallelization and the aggressive parallelization techniques we have used to overcome them, we provide examples of methods that may potentially be automated and implemented by advanced TLS compilers in the future. In this paper, we also provide a deep understanding of where thread-level parallelism (TLP) exists in

Table 1. Memory system specifications

Characteristic	Memory system		
	L1 cache	L2 cache	Main memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16 KB each	2 MB	256 MB
Bus width	32-bit connection to CPU	256-bit read bus and 32-bit write bus	64-bit SDRAM at half of CPU speed
Access time	1 CPU cycle	5 CPU cycles	At least 50 cycles
Associativity	4-way	4-way	N/A
Line size	32 bytes	64 bytes	4 KB pages
Write policy	Writethrough, no write allocate	Writeback, allocate on writes	"Writeback" (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

several important SPEC2000 benchmarks, which is useful imformation for others pursuing related research using these applications. Finally, our experience parallelizing these applications gave us an understanding of common programming styles that inhibit parallelization, and we provide guidelines for uniprocessor programmers to avoid creating code that obscures the inherent parallelism in their applications.

In the next section, we provide a short review of TLS. In Section 3 we describe the process and techniques of using TLS to manually parallelize applications. Following this, Section 4 explains the specific way in which each SPEC2000 application was parallelized. In Section 5 we present our observations and experiences arising from this study. Section 6 relates our research to other research in this field, and in Section 7 we summarize our conclusions.

2. THREAD-LEVEL SPECULATION (TLS)

We will provide only a brief explanation of TLS and our implementation of it here. More details on TLS can be found in [16][17][18][23][25]. With TLS, the programmer cuts sequential execution into sequentially ordered, non-overlapping threads, even when some memory locations may still exhibit true data dependences between the threads. The TLS hardware then attempts to execute the threads in parallel, while tracking all

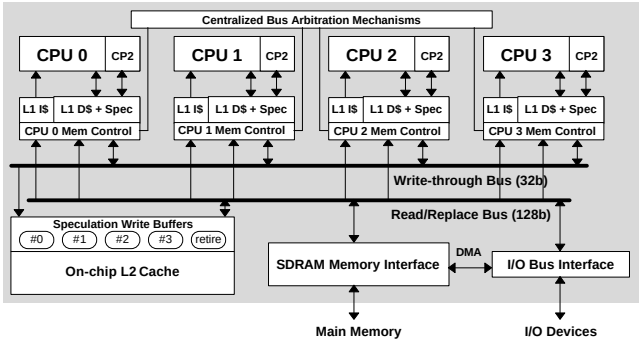

Figure 1. Hydra chip multiprocessor

Table 2. Loop-only TLS overheads

Overheads for loop-only TLS	Software handler	Instruction count
Regular events	Start loop	~30
	End of each loop iteration	12
	Finish loop	~22
Irregular events	Violation: local	7
	Violation: receive from another CPU	7
	Hold: buffer full	12
	Hold: exception	17 + OS

memory accesses to detect dependences. Anti-dependences are addressed through write buffering and in-order commits. True dependences are enforced using memory access tracking and thread rollback/restart capabilities. In this manner, the TLS system provides the appearance of sequential execution.

The TLS hardware implementation used for this paper is the Stanford Hydra chip multiprocessor. It comprises four pipelined, single-issue, MIPS-based R3000 processor cores. Each core has private instruction and data L1 caches, and the four cores share a write-back L2 cache via a snoopy bus, as shown in Figure 1. Essential details of this architecture are provided in Table 1 and more information can be found in [8]. Table 2 specifies the TLS system software handler overheads that exist for this system to start a speculative region (e.g. a loop parallelized speculatively), to finish a speculative region and to complete each thread within a region (e.g. each loop iteration). True dependences also incur overhead from a small exception handler. The speculation and violation overheads are important, as TLS parallelization focuses heavily on selecting threads and designing communication between them in a manner that reduces these overheads. We decided to use the TLS system in its simplest and lowest-overhead mode, loop-only speculation, which was intended specifically to parallelize single-level loops. This mode allows speculation on only one region at a time, without the ability to begin nested speculation on a second region within a first region. Performance on the TLS CMP was measured on a cycle-accurate simulator under three scenarios: a realistic memory model, a perfect memory model and a perfect memory model with no TLS system software handler overheads. The realistic memory model accurately simulates the effects of bus contention and memory access queuing. The performance measurements presented here were done using applications compiled by GCC 2.7.2 with optimization level -O2 on an SGI workstation running IRIX 6.5.

3. MANUAL TLS PARALLELIZATION

While many studies have concentrated on automated techniques for TLS parallelization, we use manual parallelization. Manual parallelization enables the extraction of more parallelism than is currently possible for automated methods and can point the way for further development of automated tools. We use manual parallelization techniques without regard to whether these can be automated or not. By not restricting our analysis to automatically extractable parallelism or legal compiler transformations, we can more closely approach the limits of TLS parallelization. We also provide insight into where parallelism actually exists in these applications, with the hope of bringing alternate approaches to parallelization to the attention of the developers of automated tools.

3.1 Parallelization Techniques

We will discuss the use of a variety of techniques to parallelize applications within SPEC2000. The techniques will be briefly listed here, but more detail on these has been published in [7][12][19][20][26]. The techniques fall into two categories. Techniques in the first category have been automated to some extent in previous research, while techniques in the second category require more extensive knowledge of the application and possibly changes to the program that most compilers cannot make.

In the first category, techniques that can be easily automated within compilers, are simple code motion, loop chunking/slicing, parallel reductions and explicit synchronization. Code motion seeks to make the first exposed load and the last write to each memory location as close together as possible, as this reduces the critical execution window during which a thread executing this code could suffer a violation. Secondarily, it seeks to move this window as close to the start of the thread as possible, to reduce the lost execution time if a violation does occur. Loop chunking is the technique of grouping multiple iterations of a loop together as a single thread in order to reduce per thread overheads. These sets of loops grouped in each thread can also be unrolled, to further reduce overheads. Loop slicing is the opposite, where long iterations of a loop are split into multiple threads, either to allow for more parallelism or to reduce the lost execution when a thread suffers a violation, by effectively checkpointing the iteration before it completes. Parallel reductions allow summations, multiplications and similar operations to complete in parallel, rather than forcing contention for a single accumulator variable. Explicit synchronization is useful when violations occur frequently and at a regular spot in a thread, and these violations can be avoided to prevent discarded execution.

The second category, techniques that are difficult to automate, include speculative pipelining, algorithm/data structure adaptation and complex value prediction. These techniques are described briefly here and in more detail in [19]. Speculative pipelining refers to converting fairly independent sequential portions of a program into iterations of a loop, and then executing the iterations of this loop in parallel speculatively. This often requires dynamically selecting the next iteration to execute. Algorithm/data structure changes refers to making limited changes to an application's algorithm or data structures to remove significant impediments that obscure the parallelism otherwise inherent in the application. These changes are beyond the capabilities of current compilers, since they actually modify the executed algorithm. However, all changes we propose are relatively minor; none require a fundamental redesign or rewrite of the application for multiprocessing. Complex value prediction refers to adding code to predict the future value of a memory location that often causes violations. This allows the memory location to be updated early, reducing the critical execution window during which violations can occur, when the prediction is correct. These predictions are usually difficult to automate, because making an accurate early prediction often requires some high-level knowledge of the intent or design of the algorithm.

3.2 The Parallelization Process

Our applications were profiled using reference input data sets on native hardware with a processor similar to the ones modeled in the simulated TLS system. The profiles were done with global optimizations both on and off. With global optimizations on, more accurate statistics were generated for the percentage of execution time spent in each section of the code. With global optimizations off, a more informative basic-block, butterfly profile could be generated. This listed the execution time and number of times executed for each basic block, and also the calling pattern between the basic blocks.

Table 3. Benchmark characteristics

Benchmark		Application category	Lines of code
CFP 2000	177.mesa	3-D graphics library	61,343
	179.art	Image recognition/neural networks	1,270
	183.equake	Seismic wave propagation simulation	1,513
	188.ammp	Computational chemistry	14,657
CINT 2000	175.vpr	FPGA circuit placement and routing	17,729
	181.mcf	Combinatorial optimization	2,412
	300.twolf	Place and route simulator	20,459

By examining this data, a programmer can rapidly identify candidate locations for speculative threads based upon the number of loop iterations executed, the thread lengths that would be generated and the percentage of execution time that would be parallelized. Upon examining these locations, a programmer or automated tool can determine which variables will be shared between threads, and these shared variables may not remain register-allocated. The programmer or automated tool inserts code tags to track shared variables and split execution into threads, and the program is then executed speculatively. Finally, the resulting performance statistics are examined to determine the variables which are causing the largest performance losses due to violations, techniques for exposing parallelism are applied, and the cycle of testing, performance analysis and program modification are repeated until no further speedup can be generated.

Due to the manual nature of this process, only a few locations in the source code are generally chosen for parallelization. Supplementing this manual parallelization with automated parallelization of the remainder of the code is always possible, of course, but this paper's objective is to focus on parallelism that is difficult to expose automatically, so we only present results from our manual efforts.

4. SPEC CPU2000 PARALLELIZATION

The SPEC2000 suite contains 14 floating point and 12 integer benchmarks representative of compute-intensive applications. For TLS parallelization, we selected the four floating point applications that are coded in C, since they are more difficult to parallelize than the Fortran benchmarks. We then selected three of the integer benchmarks based upon their source code size and indications from profiling that they would be amenable to manual parallelization with TLS. For example, a high concentration of execution time within just a small number of functions was considered a good sign. Information on the selected benchmarks is given in Table 3.

In the remainder of this section, we describe the ways in which each application was parallelized and the characteristics that limited speedup. We located useful parallelism within many

Table 4. Code transformations

Transformation	SPEC CFP2000				SPEC CINT2000		
	177 mesa	179 art	183 equake	188 ammp	175 vpr	181 mcf	300 twolf
Loop chunking/slicing		X	X	X		X	
Parallel reductions		X			X	X	X
Explicit synchronization					X		X
Speculative pipelining				X	X	X	X
Adapt algorithms or data structures					X	X	X
Complex value prediction					X	X	X

applications. Because many researchers are very familiar with the applications in SPEC2000, we are very specific, even listing the names of subroutines and variables within each application. Table 4 shows the TLS techniques that were utilized to expose parallelism in each application. These techniques were incrementally added, and they built upon each other to generate the final speedup. For example, in certain cases a simple technique may greatly increase performance, but only following the application of a complex technique. We discuss the simpler applications first, and then cover the more complex ones.

Table 5 provides the speedup that was achieved on the speculative regions in each application at various stages of parallelization. At each stage, the total speedup and the incremental speedup arising from the last techniques applied are provided for a realistic memory model and also a perfect one with and without TLS software handler overheads. All results utilized the reference input data sets. Because of long simulation times and the large-time-scale cyclic behavior of the reference runs, representative samples encompassing whole application cycles were selected for simulation, similar to the strategy suggested by [21]. It should be noted that in the following discussions, whenever we discuss the execution time, violation frequency or some other statistic for a subroutine, we are including the contributions not only of that subroutine, but of all other subroutines that it calls, as well.

4.1 183.equake

`Equake` is a seismic wave propagation simulation. It is extremely simple to parallelize. The five main calculation loops in the time integration loop were parallelized, along with the loop in the `smvp` subroutine. For the five loops nested in the time integration loop, each loop was individually parallelized, and ten iterations of each loop were chunked together to form each thread, providing a slight additional increase in performance. The parallelization of `smvp` used one iteration per thread.

4.2 179.art

`Art` is an image recognition application that uses neural networks. Once again, it is extremely simple to parallelize, with 95% of the execution time split between two subroutines, `match` and `train_match`. In these, each loop to update a member of the `f1_layer` (`P`, `Q`, `U`, `V`, `W`, `X`, `Y`) was treated as a speculative region, with ten iterations of each loop chunked together as a single thread. Additionally, the `P`, `V`, `W` and `Y` loops used parallel

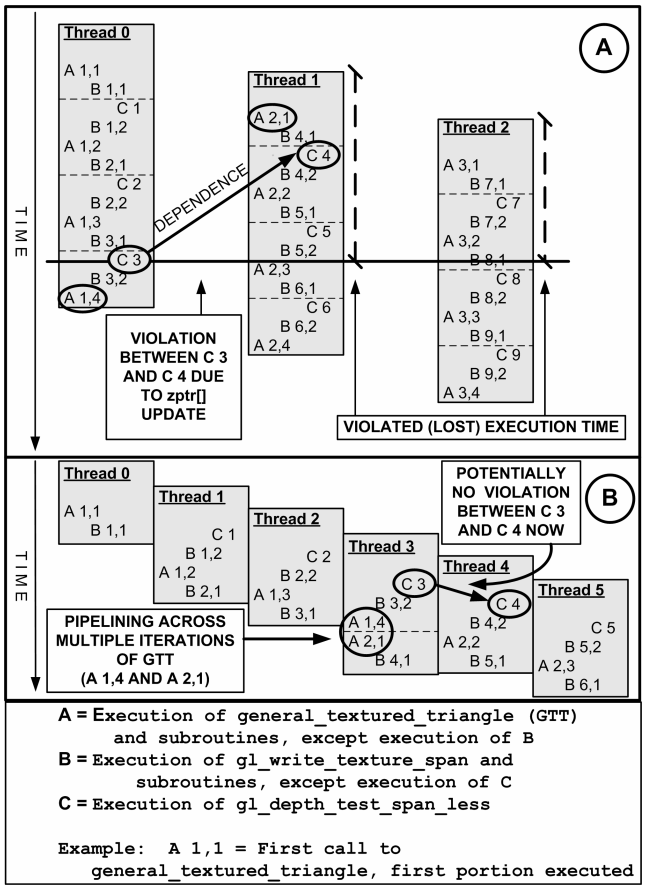

Figure 2. Execution pattern and violations of 177.mesa

reductions for summations, and the `Y` loop also utilized a parallel reduction for a logical `AND`. Each of the techniques provided a substantial gain in performance. The poor performance of the basic parallel version under the perfect memory model results from instruction count increases due to both the TLS software handler overheads and the need to force register allocated variables into the caches to allow the detection of violations. These instruction count increases affect the real memory speedups less than the perfect memory speedups, since their effects are masked by the memory delays in the real memory model.

4.3 177.mesa

`Mesa` is a 3-D graphics library. The bulk of execution (84%) occurs within `general_textured_triangle`, which is called many times per execution of `gl_render_vb` in `vbrender.c`. By simply making each call to `general_textured_triangle` a speculative thread, a 175% speedup is gained on this code. However, there is a problem with this approach. The problem arises from the fact that each call to `general_textured_triangle` calls `gl_write_texture_span` several times (an average of 3.5), where most of the execution time actually occurs. This subroutine invokes `gl_depth_test_span_less` in `depth.c` once each time, where reads and writes to an array `zptr[]` occur, causing occasional violations. These violations are especially expensive because each one can force the discarding of all more

Table 5. Speedup resulting from each additional transformation

Application		Specu-lative regions	Location of top level of speculative region(s). Line numbers are for SPEC CPU2000, version 1.00.	Percent execution time coverage	Last transformation applied	Cumul. speedup, real	Increm. speedup, real	Cumul. speedup, perfect	Increm. speedup, perfect	Cumul. speedup, perfect, no overhead	Increm. speedup, perfect, no overhead
CFP2000	177. mesa	1	vbrender.c, lines 897-901	84%	Basic	175%	175%	179%	179%	179%	179%
	179. art	7	scanner.c, lines 405-477 (7 loops) and 545-617 (7 loops)	95%	Basic	60%	60%	0%	0%	0%	0%
					Parallel reductions	122%	39%	43%	43%	112%	112%
					Loop chunking/slicing	154%	14%	282%	167%	294%	86%
	183. equake	6	quake.c, lines 449-478 (5 loops) and 1195-1220	100%	Basic	135%	135%	185%	185%	195%	195%
					Loop chunking/slicing	145%	4%	196%	4%	200%	2%
	188. ammp	1	rectmm.c, lines 562-1123	86%	Basic	61%	61%	59%	59%	62%	62%
					Speculative pipelining, loop chunking/slicing	99%	24%	69%	6%	76%	9%
CINT2000	175. vpr (place)	1	place.c, lines 506-513	100%	Basic	7%	7%	16%	16%	17%	17%
					Complex value prediction	55%	45%	67%	44%	68%	44%
					Parallel reductions, explicit synchronization	111%	36%	128%	37%	129%	36%
	175. vpr (route)	1	route.c, lines 518-541	97%	Speculative pipelining	17%	17%	16%	16%	27%	27%
					Algorithm/data structure changes	67%	43%	60%	38%	72%	35%
					Complex value prediction	88%	13%	113%	33%	128%	33%
	181. mcf	6	implicit.c, lines 246-272	44%	Loop chunking/slicing, algorithm/data structure changes	70%	70%	126%	126%	151%	151%
			mcfutil.c, lines 75-76	5%	Loop chunking/slicing, complex value prediction	24%	24%	>300%	>300%	>300%	>300%
			mcfutil.c, lines 80-109	19%	Parallel reductions, speculative slices, speculative pipelining, complex value prediction	55%	55%	10%	10%	16%	16%
			pbeampp.c, lines 96-121	7%	Speculative pipelining, algorithm/data structure changes	84%	84%	95%	95%	119%	119%
			pbeampp.c, lines 161-174	4%	Basic	64%	64%	146%	146%	197%	197%
			pbeampp.c, lines 181-195	20%	Loop chunking/slicing	89%	89%	150%	150%	211%	211%
	300. twolf	1	uloop.c, lines 154-361	100%	Speculative pipelining	18%	18%	21%	21%	23%	23%
					Parallel reductions, explicit synchronization, algorithm/data structure changes	43%	21%	53%	26%	59%	29%
					Complex value prediction	60%	12%	67%	9%	72%	8%

speculative threads, each thread containing multiple calls to `gl_write_texture_span`, as shown in Figure 2A.

One might theorize that less execution time would be discarded if threads were formed at a finer granularity. For example, using speculative pipelining, we could break the original threads into new, smaller threads along the dotted lines in each thread shown in Figure 2A. This results in the threads shown in Figure 2B, where some may even span multiple calls to `general_textured_triangle`. This could potentially eliminate the dominant violation due to updates to the `zptr` array. However, the recoding necessary to do this is much more complex. Also, it may not provide better performance, if violations increase in frequency due to the sharing of the numerous variables local to `gl_write_texture_span`. Even if violations are not a problem, this still introduces more overhead because of the need to force shared variables out of the registers and into the caches to allow for dependence tracking. This is an example of where a programmer may decide between potentially better performance with a large programming effort, or an adequate speedup with far less effort, depending on the requirements for the performance and the cost of redesign for the final parallel program.

4.4 188.ammp

`Ammp` is a computational chemistry application. 87% of the execution time is devoted to the subroutine `mm_fv_update_nonbon` in `rectmm.c`, which conducts an update of the nonbonded potentials of the atoms. While this subroutine possesses considerable thread-level parallelism (TLP), it is spread between two levels of nested loops, with program execution frequently switching between the two levels, as shown in Figure 3. About one-third of the execution time of this subroutine is spent at the outer level, while the remaining two-thirds are spent on the inner loops, which are invoked by 8% of the outer loop iterations. Parallelizing just the inner loops misses the parallelism in the outer loop. On the other hand, using only the outer loop iterations to construct threads results in 8% of the threads being enormous, leading to thread stalls caused by load imbalance.

While it is optimal to parallelize both levels together, our TLS system is not capable of speculating on multiple concurrent regions. To make our research results applicable to a wide variety of TLS architectures, we deliberately chose a simple, low-overhead, loop-only TLS system. However, simple modifications to the source code using a switch-case statement allow nested loops to be rewritten as a single-level loops using speculative pipelining [19]. With this transformation, threads can be constructed from different segments of the code, as shown in Figure 3. These four different types of threads can be executed in parallel, and this provides a significant performance boost, as shown in Table 5. Loop chunking was required to provide good load balancing by making most instances of the four types of threads similar in size. With this formulation of threads, the remaining limitations to speedup are due to violations caused by late updates of the variable `i_max`.

4.5 175.vpr (place)

`Vpr` is a FPGA circuit place and route application. Because the place and route portions of the program are so different, we will discuss them here as two separate applications. Execution of `vpr (place)` almost entirely comprises repeated calls within `try_place` to the subroutine `try_swap`. Parallelizing this loop yields only a tiny speedup, because of frequent violations due to the serial nature of the pseudorandom number generator used by the routine. Each pseudorandom number is used to generate the next one, and this occurs throughout each loop iteration.

Code segment	Line numbers in rectmm.c	Percent of total application execution time
A	562 - 567	5%
B	571 - 955	27%
C	957 - 978	5%
D	979 - 986	3%
E	987 - 993	4%
F	994 - 1121	42%

```
outer loop
{
    A ( 5% execution time)                          THREAD TYPE 0
    if (92% true)
        if (68% true * 92% = 62% combined)
            B (27% execution time)
    else (8%)
        if (90% true *  8% =  8% combined)
        {
            inner loop
            {
                C ( 5% execution time)
            }
            inner loop                              THREAD TYPE 1
            {
                D ( 3% execution time)
            }
            inner loop                              THREAD TYPE 2
            {
                E ( 4% execution time)
            }
            inner loop                              THREAD TYPE 3
            {
                F (42% execution time)
            }
        }
}
```

Figure 3. Thread formulation for 188.ammp

This problem was overcome by determining that the generator is typically called four times by each thread. To improve performance, the rewritten application performs value prediction by assuming the generator function will be called four times. At the start of each thread, the thread computes a predicted seed value for the next thread and writes this prediction to a separate shared variable. This prediction is used by the next thread to set the seed for its random number generator, after which it likewise makes a prediction for the next thread after it. At the end of all possible calls to the generator within a thread, this prediction is checked against the true final value of the seed. If a misprediction has occurred, the prediction is updated to the correct value, resulting in a violation in the next thread and re-exeution with the correctly predicted seed. However, because the prediction is usually correct, violations are unlikely.

This example illustrates the standard way in which complex value prediction can be done in a TLS system. This TLS performance technique may be possible to automate using an advanced

compiler in conjunction with application profiling. An alternative solution for this application would have been to remove the artificial dependency by just redesigning the random number generator to avoid this obvious serialization. While this may be apparent to a programmer, unfortunately compilers cannot generally redesign programs.

In addition to using complex value prediction, performance for `vpr (place)` was augmented via parallel reductions for the summation of `success_sum`, `av_cost` and `sum_of_squares`.

4.6 175.vpr (route)

`Vpr (route)` executes almost entirely within the subroutine `route_net` in the nested loop that routes each individual pin. This loop considers the cost of each new path and expands the search around any that are lower cost. Basic parallelization using each iteration of this nested loop to form a single thread is not useful for a number of reasons. Load imbalances occur because approximately half the iterations call `expand_neighbours`, and these iterations are approximately twice as long as iterations in which this does not occur. Speculative pipelining can be used to split iterations that call `expand_neighbours` into multiple threads of a similar length. Once this is done, late updates to shared variables, especially those related to the heap structures, such as `heap_free_head` and `heap_tail`, become a problem. Because the late updates occur within subroutines called from within `route_net`, the best solution was to inline the functions and then promote late updates to the earliest possible point in each iteration.

With these modifications, many of the remaining violations are due to stores that do change the value of a variable, but in a way that only infrequently affects the control flow of concurrently executing threads. For example, when there are no more free elements to add to the heap (`heap_free_head == NULL`) or the heap is empty (`heap_tail == 1`), special actions must be taken. However, most updates to the head or tail of these structures, such as adding or removing a member, do not create these special situations. However, every time the tail or head are updated, a violation will occur on all conditionals predicated on these variables in more speculative threads, anyway.

The solution is to recognize that there are two uses for each of these variables. For example, `heap_tail` is used both to point to the next element to be processed and to indicate when the heap is empty. The next element to be processed changes with a high frequency and is difficult to predict in advance. In contrast to this, whether the heap is empty changes with a low frequency and is easy to predict. By creating an additional Boolean variable to store the value of this low frequency information and then using it to control conditional execution in more speculative threads, we achieve a reduction in violations and therefore better performance.

4.7 300.twolf

`Twolf` is a place and route simulator. Almost all execution is contained within the main loop in `uloop.c`, where the algorithm attempts to place moveable cells in different blocks and measure the cost function. While each of these iterations could form a thread of appropriate length, these threads suffer too many violations.

Approximately three quarters of the execution is spent in calls to the subroutine `ucxx2` in `ucxx2.c`. We inlined this code and separated each iteration of the loop in `uloop` into eight fairly independent portions, some of which are only conditionally executed. Speculative pipelining was then used to dynamically assign one of these eight different portions to each thread. The first portion is all of the code from the main loop in `uloop.c` leading up to the call to `ucxx2`. The last two portions are the code in `uloop.c` following the call to `ucxx2`. The middle five portions are formed from the inlined code from `ucxx2`. This speculative pipelining provided some speedup. However, violations due to shared variables are severely limiting.

After threads have been created with speculative pipelining, the most significant violations between them occur on summation variables such as `cost` and `delta_vert_cost` and on accesses to `netarray` in various subroutines in `dimbox.c` called from `ucxx2`. By using parallel reductions for the summation variables and synchronizing threads that are accessing `netarray`, performance was significantly enhanced. Finally, as in `vpr (place)`, the pseudorandom number generator introduces an unnecessary serialization into the application. Advanced value prediction and communication of the expected final seed value is conducted early in each thread to mitigate this serialization.

4.8 181.mcf

`Mcf` is a combinatorial optimization application. Essentially the entire application can be parallelized by parallelizing four subroutines, `price_out_impl` in `implicit.c`, `refresh_potential` in `mcfutil.c` and `primal_bea_mpp` and `sort_basket` in `pbeampp.c`.

Parallelizing `price_out_impl` requires parallelizing a short inner loop. We chunked four iterations together to reduce the speculation overheads for these short loops. Pointer chasing on the `arcin` variable causes violations due to late updates, but by using simple code motion, these updates can be hoisted to the top of each thread. Prior to executing each of the four iterations within each thread, a check is done of whether the arcin pointer is NULL. If so, the thread terminates before completing any further iterations, and this condition causes completion of the speculative region.

`Refresh_potential` comprises two loops. The first loop, resetting the nodes, was parallelized using chunks of 100 iterations to form each thread. In the original application, once each iteration the `node` induction variable must be tested. In the speculative version, advanced value prediction is used to reduce this to just one conditional test per chunk of 100 iterations in most cases. This explains the superlinear speedups under the perfect memory model in Table 5. Under the real memory model, memory stall time limits the achievable speedup.

The second loop processes a tree in a depth-first manner. The critical path is the tree traversal code, especially due to memory delays under the real memory model. Hence, to parallelize it we made every alternate thread a speculative slice, which is a thread containing only instructions from the critical path [26]. The intervening threads conduct the actual computation at each node. Unlike the parallelization of `price_out_impl`, the pointer

chasing in `refresh_potential` consumes a large percentage of the total execution time. For this reason, it was assigned to its own separate speculative-slice threads, rather than simply moved to the beginning of each thread. Advanced value prediction is used in the speculative slices to predict the final value of the sibling search early, and this prediction is checked and updated, if necessary, when the thread completes its search. We also used parallel reductions for the checksum in the computation threads. The short lengths of both the computation and the node traversal threads, in conjunction with the prefetching effectively conducted by the traversal threads, explain the larger speedups obtained under the real memory model than the perfect memory models.

The two loops in the non-initialization section of `primal_bea_mpp` were trivial to parallelize, with three iterations per thread used for the second loop. Parallelizing `sort_basket` was not easy, due to the recursive nature of the algorithm. However, because the recursion is not deep, typically recursing less than ten times, and because the number of instructions executed at each level of the recursion is very small, it can be parallelized well. This recursive sorting can be modeled as a binary tree, where each node of the tree represents a sorting operation. This tree has the special property that the sorting operation represented by a node does not affect the sorting operation of any node of the tree that is not a descendent of that node. Speculative pipelining was used to create eleven iterations. The first seven iterations conduct the sorting operations in the top seven nodes of this tree that represents the recursive algorithm. This yields eight nodes of sorting operations at the third/fourth level of the tree of sorting operations. These eight sort baskets are assigned to the four processors in the remaining four threads. The first and last baskets of the array to be sorted are assigned to the first processor, the second and the next-to-last baskets to the second processor, etc., in order to provide better load balancing, as the number of elements to be sorted and the degree of sortedness of the baskets varies across the array. The approach could obviously be extended to allow for scalability to more processors.

5. DISCUSSION

As expected, the floating point applications were very simple to parallelize compared to the integer ones. Few complex techniques were used on them, and they were very uniform in the nature of their threads. This is in contrast to the many, varying speculative regions in the integer applications, each of which was substantially different from the speculative regions in the rest of the application. While high parallel coverage was managed for all the applications, the integer applications required more effort and more complex parallelization techniques, and each region parallelized required a different approach to parallelization. The complexity of the parallelization of the integer applications made speculative pipelining an essential technique, as threads often had to be constructed from execution segments that did not belong to simple loops, the form of parallelism required by a simple TLS system. In spite of these challenges, very good speedup was managed even on these integer applications.

Table 6 provides the average lengths of the TLS threads used in the final versions of these applications. These thread lengths represent just the number of instructions in the original, unmodified applications. Specifically, they do not include any of the overheads of executing the TLS software handlers, adapting the applications to expose parallelism or forcing register-allocated variables into the caches to allow the detection of violations. The thread sizes vary considerably, but all are in the range of hundreds to thousands of instructions long. As a result, the parallelism extracted from these applications is thread-level parallelism and is orthogonal to the instruction-level parallelism (ILP) that could also be extracted from these same applications. Because of the lengths of the threads chosen to parallelize these applications, the extraction of ILP within each thread, for example by an aggressive out-of-order processor, would be expected to have little effect upon the speedups generated by TLS parallelization.

Table 6. Speculative thread lengths

Application		Average dynamic thread length in original code (instructions)
CFP 2000	177.mesa	7,800
	179.art	450
	183.equake	1,300
	188.ammp	450
CINT 2000	175.vpr (place)	5,100
	175.vpr (route)	200
	181.mcf	250
	300.twolf	700
Column mean		2,030

In the remainder of this section, we will provide a summary of the obstacles we encountered to parallelizing these applications, and what guidelines a programmer can follow to avoid developing applications that obscure parallelism.

5.1 Hindrances to TLS Parallelization

From our experience parallelizing SPEC2000 applications, we observed a number of common hindrances. We will briefly summarize them here, starting with those that are inherent to the application and working down to those that pertain to our specific TLS system.

Many parts of integer applications are inherently difficult to statically parallelize into threads. While speculative pipelining allows for some dynamic adaptation of the way in which code is divided into threads, parallelizing this kind of code may benefit from hardware that provides more support for dynamic thread creation, similar to [2].

Some algorithms within applications interact badly with TLS, in general. For example, deeply recursive algorithms with extensive execution at each level of the recursion interact very badly with TLS. This is especially true if the recursion depth is very different at different parts of a tree structure and the tree changes often, as there is no indication of proper places to cut the algorithm into load-balanced speculative threads. Even worse, reuse of global variables during recursion generally obscures all parallelism and completely serializes the subroutine.

Some applications interact badly with our TLS implementation. For example, our choice to use less complex TLS hardware results in longer communication delays between processors than in more tightly coupled TLS architectures. This makes some thread formulations infeasible, because the thread lengths are too short to amortize the TLS overheads. Another problem is applications with iterations that vary greatly in length. They overflow

speculative buffers, causing stalls, for long threads and suffer from load imbalance on short threads. While the former problem can be avoided with most applications, the latter is an issue that frequently arises. Methods have been proposed to address this problem [4]. However, speculative pipelining can be used to dynamically redistribute work between threads to conduct adequate load balancing, as shown for `ammp`.

Load imbalance stalls are caused by sudden decreases in thread length, but not sudden increases in thread length. This is because speculative state must be committed in order, so short threads may need to wait for longer, prior threads to complete. Figure 4 shows various thread length sequences. Figure 4A shows an ideal thread length sequence resulting in no stall time. Figure 4B shows that gradual decreases and even sudden increases in thread length also cause no stalls. However, as Figure 4C shows, sudden, large decreases in length do cause stall time. Figure 4D shows likewise that large variances in thread length can cause stalls.

5.2 TLS-Friendly Uniprocessor Programming

The manner in which a uniprocessor program is written can profoundly affect the ease with which it can be parallelized. We have identified several simple rules that are easy for a uniprocessor programmer to follow that allow for rapid porting of the final program to a TLS platform.

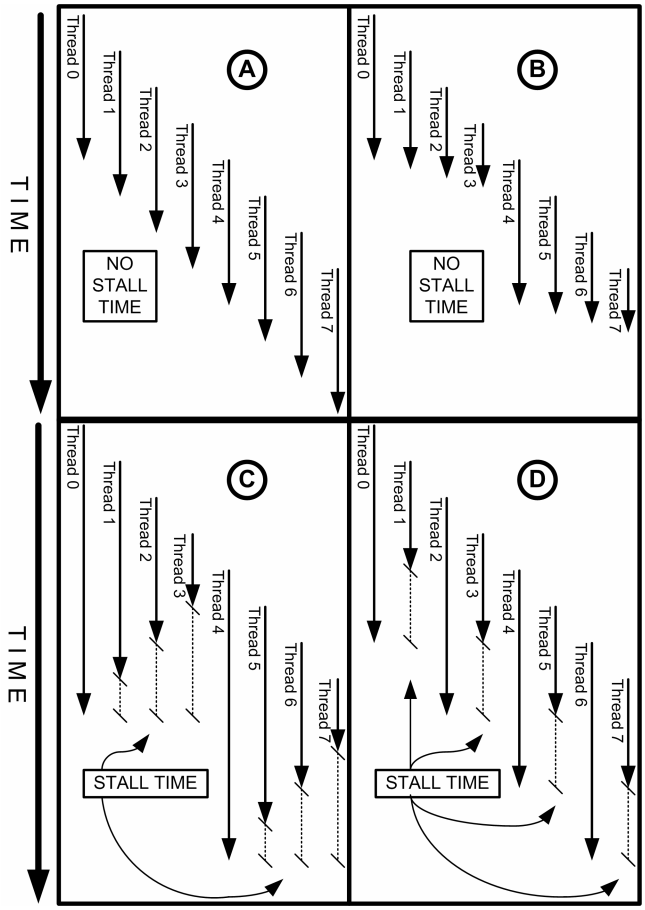

Figure 4. Good and bad thread length sequences

1) *Avoid recursive subroutines that return or modify critical values needed by parents:* Because TLS systems need to commit threads in order, one can only place each call into its own thread if each level of recursion is not dependent upon its children, which must go into "later" threads. This is trivially the case if a recursive subroutine call is the last statement in the parent subroutine, and it does not return a value that is used, but this is rare. The problem is that TLS utilizes a FIFO of threads, while recursion uses a stack of calls. Likewise, some recursive subroutine calls that do not affect their parent or sibling recursive subroutine calls can be parallelized fairly easily. However, the need for good load balancing and the requirement to store speculative state require that the depth of recursion be both limited and similar for all children of a parent subroutine. This is the case for `sort_basket` in `mcf`, which allowed it to be parallelized.

2) *Avoid the use of tailored algorithms for standard purposes:* For example, certain applications benefit marginally from complex sorting algorithms tailored to the characteristics of the data being sorted. Replacing these with calls to standard library sorting algorithms can facilitate their replacement with standard parallelized library calls.

3) *Avoid data structures and algorithms with undesirable communication patterns:* Algorithms utilizing binary trees generate data "hot spots" that cause contention and frequent violations. Also, the use of complex sorting algorithms that swap distant array members rather than adjacent neighbors can be much more difficult than simpler algorithms to parallelize or synchronize. Often these complex data structures or algorithms are used to provide optimal uniprocessor performance. Because TLS systems can provide a good parallel speedup on many applications, programmers may wish to avoid the best uniprocessor data structure or algorithm, if it is only a little more efficient than a more easily parallelizable one.

4) *Avoid unnecessary reuse of variables:* Using a single variable for multiple purposes, especially a global variable, can often cause unnecessary violations. For example, the use of queues, stacks or heaps accessed in multiple spots during the same subroutine, but for different purposes, will cause violations when the head and tail pointers or the queue length are updated. Using separate data structures for independent portions of execution eliminates this problem. Likewise, using the queue length to check for an empty queue can cause unnecessary violations every time an element is added or removed. Creating a new Boolean variable for the condition of an empty queue, rather than reusing the integer queue length variable, can prevent this, as we demonstrated with `vpr (route)`.

5) *Encapsulate functionality and avoid global variables:* Large, monolithic subroutines and applications that extensively use global variables make the task of identifying the communication patterns in a program very difficult, thereby complicating efforts to determine good separation points for threads.

6) *Avoid unnecessarily serializing algorithms:* Algorithms such as random number generators that must pass a seed to the next generator call serialize the algorithm unnecessarily if the number of times the generator will be called within a thread cannot be

accurately predicted. These algorithms should be replaced with more distributed versions, instead.

6. RELATED WORK

Research has been done on automatic parallelization [1][13] and speculation [4][17][18][22][26] at various universities. However, this research primarily focuses on developing methods to automatically parallelize applications. We instead investigate two other areas. First, we use manual parallelization, so that design modifications and programmer expertise can be utilized to yield higher parallel performance. In this manner, we approach the upper bounds of the parallelism that can be extracted from these applications, using a loosely coupled (L2-cache-connected) CMP that supports only fairly simple TLS. Second, we use our experience with manually parallelizing these applications to explain precisely where in these important benchmarks TLP exists, how to extract it and how to overcome the obstacles to parallelization of each application.

This research supplements related research by providing a rough indication of an upper bound on performance for similar TLS systems exploiting TLP of the same granularity. Together with an explanation of where and how we extracted this parallelism, it can help direct related research to see if some of these techniques can be automated and if improvements can be made to automated thread selection algorithms.

While we have parallelized floating point applications, we specifically excluded Fortran applications. Research on parallelizing Fortran applications has been conducted by others [4][20][23][25]. Often this has been done with additional, specialized support from the hardware for specific tasks such as parallel reductions or non-blocking commits of speculative state. We have instead focused on C programs, including integer applications. Like us, some other researchers have focused on general purpose applications. The Wisconsin Multiscalar team achieves excellent speedups on general purpose applications, including integer applications, using a more tightly coupled CMP TLS system than ours [16][26]. Their system allows register-to-register communication between the processors, and requires a more complex and high-speed architecture than ours, a different hardware/software design space from our intentionally simple TLS system. Research on automated parallelization by the CMU STAMPede team [22][23][24] and at the University of Illinois [4][5][25] uses less closely coupled processors in a system more similar to ours.

Substantial research on techniques to exploit value prediction and dynamic synchronization has been conducted in [3][5][7][12][16][22]. We utilize these techniques where applicable and extend upon them. For example, we use complex value prediction to increase performance. This extends research from earlier studies that explores only predictions of values that do not change or that change with a simple stride.

7. CONCLUSION

Previous studies have shown that conventional parallelization is improved [14][18] and manual parallelization vastly simplified [19] by the availability of TLS support. We have provided a short description of the process by which a TLS programmer profiles, analyzes and parallelizes an unfamiliar legacy application. By using basic-block-level, butterfly profiles and run-time violation performance monitoring, the programmer can rapidly locate parallelism and identify variables that limit parallel performance.

Having described this process, we took a detailed look at the parallelism in a number of SPEC2000 applications. For each application, we described the specific location of this parallelism as a roadmap to other researchers who may wish to utilize this information for their own purposes. We have described the key impediments in each application that blocks extraction of the parallelism. Finally, we have discussed the methods we used to overcome these impediments, and we have provided performance results for each of these parallelized applications.

While `equake`, `art` and `mesa` are relatively simple to parallelize and can be done via automatic parallelization, the rest of the applications benefit from the expertise of a programmer. In `ammp` this is due to having the parallelism evenly divided between two levels of a nested loop, and having complex load imbalances based upon the outcome of a conditional statement. `Vpr (route)` also exhibits complex load balance issues based upon a conditional branch. In addition, it contains frequently updated variables that are utilized as both integer and Boolean variables. Splitting these uses into two separate variables enhances the parallelism. Redesigning the recursion and shortening the critical paths in parts of `vpr (route)` also enable its parallelization. `Vpr (place)` suffers from artificial serialization due to a random number generator, as does `twolf`. Finally, several of the parallelized applications benefit from synchronization to eliminate a number of preventable violations.

With the strong movement of mainstream computing toward single-chip multiprocessors, the potential exists for TLS support to be added to future CMPs. This would simplify the process of parallel programming and enable higher performance on applications with parallelism that is difficult to extract using static methods. With this in mind, we have provided broad guidelines to uniprocessor programmers on how to design programs that will port easily to TLS CMP platforms. While this was done specifically with TLS platforms in mind, all of these guidelines also facilitate porting applications to non-TLS parallel platforms.

In summary, by providing this detailed explanation of the parallelism in several SPEC2000 applications, we show that significant parallelism can be extracted using TLS, even from several integer applications. Furthermore, we show the way in which this can be done manually, with the hope that these examples will help inform future efforts on automated TLS parallelization.

8. ACKNOWLEDGMENTS

This work was supported by DARPA/Department of the Air Force contract F29601-03-2-0117, NSF contract CCR-0220138, the Intel graduate fellowship program and the Alliance for Innovative Manufacturing at Stanford. The authors would also like to thank Lance Hammond for extensive discussions and key insights on this paper and for support of the Hydra CMP simulator.

9. REFERENCES

[1] B. Blume, et. al, "Restructuring programs for high-speed computers with Polaris," *Proc. 1996 ICPP Workshop on. Challenges for Parallel Processing*, pp. 149-161, Aug. 1996.

[2] M. Chen and K. Olukotun, "The JRPM system for dynamically parallelizing Java programs," *Proc. 30th Annual Intl. Sym. on Computer Architecture (ISCA)*, San Diego, CA, pp. 434-445, Jun. 2003.

[3] G.Z. Chrysos and J.S. Emer, "Memory dependence prediction using store sets," *ISCA-25*, Barcelona, Spain, pp. 142-153, June 1998.

[4] M. Cintra, J. Martínez and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," *ISCA-27*, Vancouver, Canada, pp. 13-24, June 2000.

[5] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for Multiprocessors," *Proc. 8th Intl. Sym. on High-Performance Computer Architecture (HPCA)*, Cambridge, Massachusetts, pp. 43-54, Feb. 2002.

[6] J. Clabes, et al., "Design and implementation of the POWER5 microprocessor," *IEEE Intl. Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, Feb. 15-19, 2004.

[7] F. Gabbay and A. Mendelson, "Using value prediction to increase the power of speculative execution hardware," *ACM Transactions on Computer Systems*, vol. 16, pp. 234-270, Aug. 1998.

[8] L. Hammond, et al., "The Stanford Hydra CMP," *IEEE Micro*, pp. 71-84, Mar.-Apr. 2000.

[9] P. Kongetira, "A 32-way multithreaded SPARC® processor," *Hot Chips 16*, Stanford, California, Aug. 22-24, 2004.

[10] K. Krewell, "AMD vs. Intel in dual-core duel," *Microprocessor Report*, Scottsdale, AZ, July 6, 2004.

[11] D. Lammers, "Intel cancels Tejas, moves to dual-core designs," *EETimes*, Manhasset, New York, May 7, 2004.

[12] K.M. Lepak, G.B. Bell, and M.H. Lipasti, "Silent stores and store value locality," *IEEE Transactions on Computers*, vol. 50, pp. 1174-1190, Nov. 2001.

[13] S.W. Liao, et al., "SUIF Explorer: An Interactive and Interprocedural Parallelizer," *Proc. Sym. Principles and Practices of Parallel Programming 1999 (PPOPP 1999)*, pp. 37-48, Atlanta, Georgia, Aug. 1999.

[14] J.F. Martinez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," *Proc. 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, pp. 18-29, Oct. 2002.

[15] C. McNairy and R. Bhatia, "Montecito - The next product in the Itanium® Processor Family," *Hot Chips 16*, Stanford, California, Aug. 22-24, 2004.

[16] A. Moshovos, S.E. Breach, T.N. Vijaykumar, G.S. Sohi, "Dynamic speculation and synchronization of data dependences," *ISCA-24*, Denver, Colorado, pp. 181-193, June 1997.

[17] K. Olukotun, L. Hammond, and M. Willey, "Improving the performance of speculatively parallel applications on the Hydra CMP," *Proc. 13th ACM International Conference on Supercomputing (ICS)*, Rhodes, Greece, pp. 21-30, June 1999.

[18] C.-L. Ooi, et al., "Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor," *ICS-15*, June 2001.

[19] M. Prabhu and K. Olukotun, "Using thread-level speculation to simplify manual parallelization," *Proc. Sym. PPOPP'03*, San Diego, CA, pp. 1-12, June 11-13, 2003.

[20] L. Rauchwerger, N. Amato, and D. Padua, "Run–time methods for parallelizing partially parallel loops," *ICS-9*, Barcelona, Spain, pp. 137-146, July 1995.

[21] T. Sherwood and B. Calder, "Time varying behavior of programs," *Tech. Rep. No. CS99-630*, Dept. of Computer Science and Eng., UCSD, Aug. 1999.

[22] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "Improving value communication for thread-level speculation," *HPCA-8*, Cambridge, Massachusetts, pp. 65-75, Feb. 2002.

[23] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A scalable approach to thread-level speculation," *ISCA-27*, Vancouver, Canada, pp. 1-12, June 2000.

[24] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, "Compiler optimization of scalar value communication between speculative threads," *ASPLOS-X*, San Jose, California, pp. 171-183, Oct. 2002.

[25] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," *HPCA-5*, Orlando, Florida, pp. 135-141, Jan. 1999.

[26] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *ISCA-28*, Goteborg, Sweden, pp. 2-13, July 2001.