Improving the Performance of Speculatively Parallel Applications on the Hydra CMP

Kunle Olukotun, Lance Hammond and Mark Willey
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
http://www-hydra.stanford.edu/

ABSTRACT

Hydra is a chip multiprocessor (CMP) with integrated support for thread-level speculation. Thread-level speculation provides a way to parallelize sequential programs without the need for data dependence analysis or synchronization. This makes it possible to parallelize applications for which static memory dependence analysis is difficult or impossible. While performance of the baseline Hydra system on applications with medium to large grain parallelism is good, the performance on integer applications with fine-grained parallelism is unimpressive. In this paper, we describe a collection of software and hardware techniques for improving speculation performance over the baseline speculative Hydra CMP. These techniques focus on reducing the overheads associated with speculation and improving the speculation behavior of the applications using code restructuring. When these techniques are applied to a set of eleven integer, multimedia and floating-point benchmarks, significant performance improvements result. In particular, the overall performance of the integer benchmarks is improved by seventy-five percent.

Keywords

Chip multiprocessor, data speculation, multithreading, performance evaluation, parallel programming, feedback-driven optimization.

1. INTRODUCTION

Hardware support for speculative thread parallelism makes it possible to parallelize sequential applications without worrying about data dependencies at compile time. Even if there are dependencies between speculative threads, the hardware support guarantees correct program execution. This support significantly increases the scope of applications that can be automatically parallelized, because there are many applications that have thread-level parallelism, but whose memory dependencies cannot be analyzed at compile time. The Multiscalar architecture was the first architecture to include this support [18]. More recently, there have been proposals to add support for speculative thread parallelism to a chip multiprocessor (CMP) [15] [19]. In this paper, we focus on improving performance with speculative thread support on the Hydra CMP [7].

To generate code for a speculative thread architecture, the program must be broken into threads. A program can be partitioned into arbitrary threads, but to minimize the effect of control hazards it is desirable to pick threads that are either control independent or whose control dependencies are easy to predict. Loop and procedure program constructs provide good candidates for speculative threads [16]. With loops the speculative threads are the loop iterations, while with procedures the speculative threads are the procedure call and the code following the procedure call.

Given a sequence of speculative threads, it is the job of the Hydra hardware to execute these threads correctly. The hardware accomplishes this using data speculation support in the memory system, which records speculatively executed memory references so that they can be squashed when necessary. Speculative threads are executed in parallel, but commit in sequential order. Committing a speculative thread requires the thread to write out any speculative data that was created during the execution of the thread to the sequential state of the machine. At any point before a thread is committed, it may be forced to restart due to a data dependency violation with a less speculative thread, which occurs when a speculative thread reads a data memory location before it has been written by the less speculative thread. The simplest way to deal with violations is to discard the speculative state and restart the speculative region from the first instruction in the thread.

Unfortunately, hardware support for speculation is not sufficient to guarantee good parallel performance for all applications. There are several reasons why parallel performance with speculative threads may be limited for a particular application. The most fundamental limit is a lack of parallelism in the application. This manifests itself as low parallel coverage (the fraction of the sequential execution time that can be parallelized using speculative threads) or reasonable coverage but low performance due to true data dependencies in the parallel portion of the program that continually cause violations. These data dependencies may be inherent to the algorithm, or they may just be an unfortunate programming choice that worked well in sequential execution but causes unnecessary dependencies when the program is speculatively parallelized. Less fundamental performance limits are due to the software overheads associated with managing speculative threads, the increased latency of inter-thread communication through memory (instead of registers), and the amount of wasted work that must be reexecuted when violations occur.

The performance of speculatively parallelized applications can be improved by optimizing the applications themselves and/or the speculation support software and hardware. Performance can be

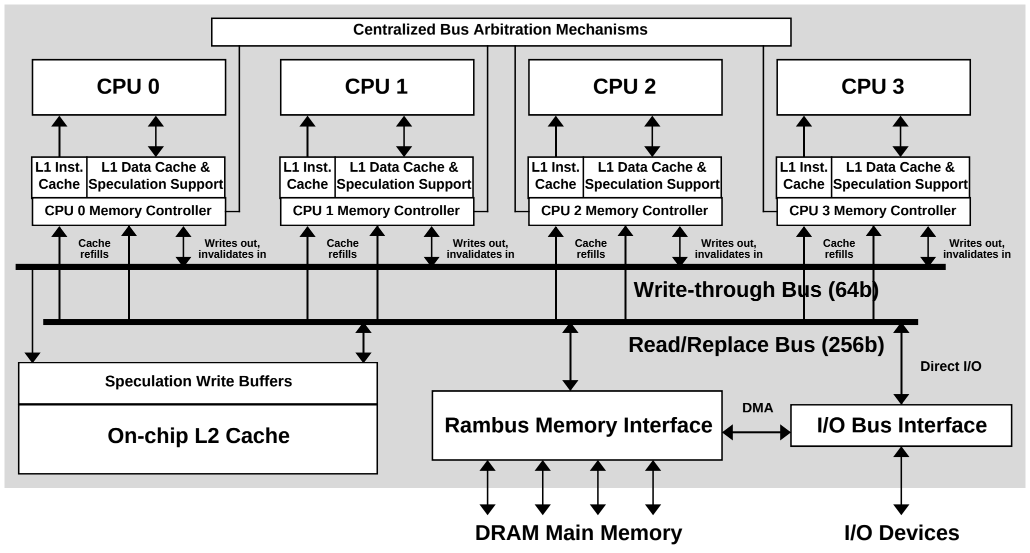

Figure 1. The main datapaths in the Hydra CMP.

improved by selectively choosing threads that have more parallelism. Furthermore, compiler and manual optimization can be used to rearrange code to reduce the likelihood of violations. Speculation runtime software can be optimized to reduce the overheads of managing speculative threads and a combination of hardware and software can be used to minimize the performance losses from violations. In this paper we show how many of these techniques can be combined to improve the performance of speculatively parallel applications on the Hydra CMP.

The rest of this paper describes the Hydra system and shows how the performance of Hydra on speculatively parallelized applications can be improved using both software and hardware. In the next section, we describe previous work in thread speculation. In Section , we briefly describe the Hydra chip multiprocessor and the software support required for speculation. In Section 4, we present and explain the performance of the baseline Hydra system on eleven benchmarks. In Section 5, we describe how the model used to divide the program into threads can be extended to expose more speculative parallelism and explain how violation statistics collected from previous runs of a speculative program can be used to find and eliminate unnecessary dependencies. We also describe how the base support for speculation in the memory system can be improved to reduce the effects of data dependence violations, which provides further performance gains. In Section 6, we present speculation performance results using these software and hardware enhancements. We conclude in Section 7.

2. PREVIOUS WORK

The first mention of speculative threads was in the work of Knight in the context of LISP [10]. However, the Multiscalar paradigm [3] [18] has most directly influenced the Hydra research. The Multiscalar paradigm takes a hardware-centric approach to extracting fine-grained parallelism from sequential integer programs. The hardware required includes processing units whose register files are connected by a ring and a shared primary data cache with speculative memory support provided by an address resolution buffer (ARB) [4]. More recently, the speculative versioning cache work has eliminated the requirement of the shared primary data cache using extensions to a multiprocessor cache coherency protocol [5]. The Multiscalar group has observed that blind speculation on all data dependencies can severely impact performance when the speculation is wrong and that dynamic data dependence prediction techniques can be used to improve performance over blind speculation [13]. Following on the Multiscalar work other researchers have proposed adding speculative data support to a chip multiprocessor [15] [19]. Here the goal is to simplify the task of creating parallel programs for a multiprocessor. The assumption in these proposals is that the processing units are less tightly coupled than in the Multiscalar architecture and so the focus of the CMP work is on extracting and exploiting coarser-grain parallelism.

3. THE HYDRA CMP

Hydra is a chip multiprocessor with data speculation support. We briefly describe the Hydra architecture and software support environment here in sufficient detail to understand the remainder of the paper, but the Hydra system is more completely described in [6] and [7].

3.1 Architecture

The Hydra architecture consists of 4 MIPS processors, each with a pair of private data caches, attached to an integrated on-chip secondary cache using separate read and write busses, as shown in Figure 1. The processors use data caches with a write-through policy to simplify the implementation of cache coherence. All writes propagate through to the write back secondary cache using the dedicated *write bus*. In order to ensure coherence, the other processors' data caches watch this bus — using a second set of cache tags — and invalidate lines when necessary to maintain cache coherence. On-chip communication among the caches and the external ports, such as data cache refills, are supported by the cache-line-wide *read bus*. Both buses are fully pipelined to maintain single-cycle occupancy for all accesses. Off-chip accesses are handled using dedicated main memory and I/O buses. For the applications evaluated in this paper, the bandwidth of these buses is not a performance bottleneck. A summary of the pertinent characteristics of the Hydra CMP memory system appears in Table 1.

	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	16KB each	2 MB	128 MB
Bus Width	32-bit connection to CPU	256-bit read bus and 32-bit write bus	32-bit wide Rambus connection (2 x DRDRAM) at full 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

Table 1. Hydra memory hierarchy characteristics.

Thedata speculation support in Hydra allows threads to execute in parallel but ensures that they commit in sequential order. The hardware guarantees that no speculative thread can change the sequential state of the processor, which is kept in the secondary cache and main memory, until the thread commits. Executing threads in parallel causes memory data hazards. Hydra deals with WAR and WAW hazards using the implicit memory location renaming that occurs when a processor has speculative data in its data cache. This memory renaming avoids the need to stall speculative threads for these memory hazards. RAW hazards caused by true data dependencies result in violations that force the violating processor to back up and start again from the beginning of the speculative region it is currently processing. However, using the write bus to forward speculative data between threads running on different processors reduces the occurrence of these violations.

Data speculation support in Hydra consists of a set of new coprocessor instructions, extra tag bits which are added to the processor data caches, and a set of secondary cache write buffers. The coprocessor instructions are provide an interface to the thread speculation control hardware, the tag bits are used to detect data dependency violations between threads, and the write buffers are used to buffer speculative data until it can be safely committed to the secondary cache or discarded.

3.2. Software Support

There is both compiler and run-time system software support for speculation in the Hydra architecture. A source to source compiler (*hydracat*) is used to transform C `for` and `while` loops into speculative `for` and `while` loops. Currently, the loops are selected by the programmer by designating candidate loops using the `pfor` and `pwhile` keywords. The compiler then automatically transforms these loops to their speculative versions by outlining the loop as a procedure and by analyzing the dataflow in the loop body so that any local variables that have the potential to create loop-carried dependencies are globalized

The other component of the software support for speculation is the runtime system. The runtime system is used to control speculative threads through the coprocessor interface and stores to special I/O locations. This collection of small software control routines is used to control the speculative threads currently active in the system. In our baseline system, these routines control threads created in two different ways: loop iterations and procedure calls. When a procedure call is encountered, the runtime system forks off a thread to speculatively execute the code following the return from the call, while the original thread executes the code within the procedure itself. If the return value from the procedure is predictable, and if no memory referenced as a side effect by the procedure is read immediately following the procedure return, then the two threads may run in parallel. Similarly, when a `pfor` or `pwhile` loop is encountered, the system starts threads on all of the processors to handle iterations of the loop, which are then distributed among the new threads in a round-robin fashion. The routines must also track the order of all speculative threads in the system in order to properly forward data between threads and to signal data dependency violations, which trigger exception routines that squash and restart the violating thread and all threads that are more speculative. See [7] for further details.

4. BASE SPECULATION PERFORMANCE

4.1. Benchmarks and Simulation Methodology

To evaluate our speculative system and identify potential areas for improvement, we used eleven representative benchmarks that cover a significant portion of the general-purpose application space. These benchmarks are listed in Table 2. The speculative versions of the benchmarks were created using Hydracat. The binaries were generated using GCC 2.7.2 with optimization level -O2 on an SGI workstation running IRIX 5.3. Currently, Hydracat only works on C programs. This prevents us from experimenting with any of the SPEC95 floating-point benchmarks, which are all written in FORTRAN. However, from the point of view of extracting parallelism, the C SPEC92 floating point benchmarks are more challenging than the SPEC95 floating point benchmarks, so we would expect the performance on SPEC95 benchmarks to be similar or better.

	Application	Source	Input Data Set
General Integer	compress	SPEC95	train
	eqntott	SPEC92	reference
	grep	UNIX utility [8]	190 line file
	m88ksim	SPEC95	test
	wc	UNIX utility	10,000 character file
Multimedia Integer	ijpeg(compression)	SPEC95	train
	mpeg-2(decoding)	MediaBench [12]	test.m2v
Floating Point	alvin	SPEC92	reference
	cholesky	Num. Recipes [17]	100 ×100
	ear	SPEC92	reference
	simplex	Num. Recipes [17]	40 variables

Table 2. Benchmark summary.

All the performance results we present are obtained by executing the benchmarks on a detailed, cycle-accurate simulator that models the Hydra architecture described in Section 3.1. The execution time of the speculative binaries includes the time spent in the run-time system. Our simulation environment can switch back and forth between execution using the cycle-accurate simulator and execution directly on the real machine. Switching to the real machine allows us to run entire applications on the simulator by quickly executing initialization code and redundant executions of kernels. However, code representing at least 95% of the original sequential execution time is run on the cycle-accurate simulator to ensure that the simulations are representative.

4.2. Baseline Performance

The performance of the base Hydra CMP is shown in Figure 2. The speedups represent the execution time of one of the processors in Hydra running unmodified C code divided by the execution time of the speculative Hydra CMP running speculatively parallelized C code. We see that the performance is highly dependent on the application and varies from 0.6 to 3.4. To understand the performance profile in more detail, we explain how each benchmark was parallelized and what limits the speculative parallel performance on the benchmark**.**

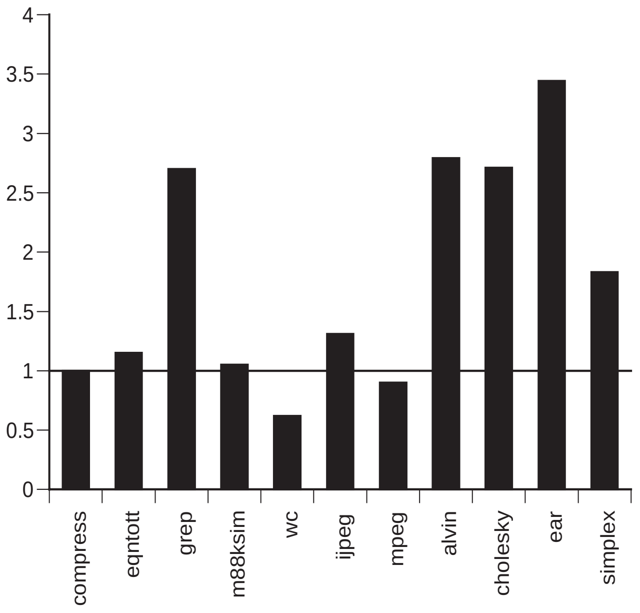

Figure 2. Baseline speculation speedup.

compress (compression): This benchmark compresses an input file using entropy encoding. Compress is dominated by a single small loop (about 100 instructions, on average) that has complex control flow. As observed in [18] and contrary to the analysis in [19] there is a loop-carried dependency that limits the parallelism in this loop. However, there are opportunities to overlap the I/O routines of separate loop iterations. The limited parallelism and small thread size in compress are overwhelmed by the speculation software overheads on the baseline system.

eqntott: This benchmark performs logic minimization on a set of input equations. Unlike the other benchmarks, we did not parallelize the loops in this application, because the only good candidate loop (in the *cmppt* function) is too small to be effectively parallelized on Hydra without significant programmer help. As a result, we used procedure-call based speculation. Despite a reasonable amount of parallelism in the recursive *quicksort* procedure, the speculation software overheads and squashing of non-parallel procedures limit speedup.

grep: This benchmark is the well-known regular expression tool that matches a regular expression to each line of an input file and prints those lines that contain matching strings. The main loop of the program iterates over all the lines of the input file. The speculative thread version of this loop performs very well because the only dependencies that occur are during file input and when a match is found. However, most lines do not match the expression and so once they have been read from the input file they are processed completely in parallel.

m88ksim: This program performs simulates a Motorola 88000 RISC CPU on a cycle-by-cycle basis. It consists of a single, large instruction execution loop that is potentially parallel, but unfortunately also accesses several global variables that tend to exhibit true dependencies in a manner that is very difficult for a compiler to statically analyze. In the speculative version of m88ksim, accesses to these global variables cause a considerable number of violations. The execution time that is lost when these violations occur limits speedup.

wc: This is the UNIX word count utility, which counts the number of characters, words, and lines in an input file. It primarily consists of a single character processing loop that is very small — loop iterations have only 20-40 instructions, on average. In addition, there are loop-carried dependencies, such as the file pointer and *in a word* indicator. The speculative version of this tiny loop is slowed down by the speculation software overheads and the relatively high interprocessor communication latencies, caused by communicating through memory instead of registers. Consequently, wc slows down by a considerable amount.

ijpeg (compression): This is an integer, multimedia benchmark that compresses an input RGB color image to a standard JPEG file, using a lossy algorithm consisting of color conversion, downsampling, and DCT steps. The multiple loops in these steps posses a large amount of inherent parallelism, but were originally coded so that the parallelism is often obscured by existing program semantics, such as numerous pointers, poor code layout, and odd loop sizes. The speculatively parallel version of ijpeg exposes this parallelism, but performance is limited by parallel coverage and speculation software overheads.

mpeg (decoding): This is a benchmark from the Mediabench suite [12] that decodes a short MPEG-2 video sequence to an RGB frame buffer. Parallelization occurs at the macro block level, where the variable-length decoding (VLD) is performed. The VLD step is completely serial, but speculation is able to overlap the processing performed during the other, more parallel stages of decoding (IDCTs and motion estimation) with the serial VLD step of later macroblocks. This parallelization technique is described in greater detail for a hand-parallelized version of mpeg in [9]. Despite a significant amount of potential parallelism, MPEG-2 decoding slows down because of a loop-carried dependency that unnecessarily serializes execution.

alvin: This is a neural network training application for autonomous land vehicle navigation. It is composed of four key procedures which are dominated by doubly-nested loops. Although these

Application	Coverage (%)	Restarts per thread	CPU utilization (%)	Maximum Speculative write state
compress	100	6.4	28	24
eqntott	93	12.1	32	40
grep	90	1.1	75	11
m88ksim	94	15.5	29	28
wc	100	4.6	69	8
ijpeg	60	2.35	42	32
mpeg	92	1275	25	56
alvin	96	0.31	81	158
cholesky	91	0.88	74	4
ear	96	0.32	97	82
simplex	86	0.14	60	14

Table 3. Speculation performance characteristics. The speculative write state is presented in terms of 32 byte cache lines.

loops are parallel, the parallelism is obscured by the way in which the application is coded with pointer variables. However, these loops are easily speculatively parallelized and result in good parallel performance.

cholesky: Cholesky decomposition and substitution is a well-known matrix kernel. The multiply nested loops in the decomposition procedure have significant amounts of parallelism, but they also contain loop-carried dependencies that occur infrequently. In conventional parallelization, these dependencies would have to be detected and synchronized, but with speculation we can obliviously parallelize them. The loops in the substitution routine are also parallel, but here each loop is written in so that a loop-carried dependency serializes the speculative loop. Fortunately the decomposition procedure dominates the sequential execution time and speedup is still quite good.

ear: This benchmark simulates the propagation of sound in the human inner ear. The conventional wisdom is that the structure of this program is a sequential outer loop and a sequence of parallel inner loops [11]. The parallel inner loops are extremely fine grained and therefore are not good candidates for speculative threads on the Hydra CMP. However, the outer loop can be speculatively parallelized and achieves very good speedup. The dependent outer loop pipelines across the processors quite well because each iteration is only dependent on the previous iteration in a way that allows much of the computation to be overlapped. Memory renaming ensures that, even though each iteration of the outer loop uses the same data structures, each processor dynamically gets a private copy.

simplex: This kernel solves problems in linear programming. The three procedures in this kernel have a number of small loops. The dependences between the iterations of these loops are not at all obvious. The speculative parallelization of these loops achieves good speedup, but less than the other numerical benchmarks. This is mainly due to the speculation software overheads that dominate the short running loops.

Table 3 shows some key performance characteristics for the speculative versions of the eleven benchmarks. Except for the results from the eqntott benchmark, the data for this table was collected with the speculation control software optimizations to be discussed in Section 5.1 (but without the other optimizations discussed in Section 5). Except for the ijpeg and simplex benchmarks, more than 90 percent of each benchmark can be speculatively parallelized. However, high coverage does not guarantee good performance when there are a significant number of violations. The number of restarts per committed thread (restart rate) gives a good indication of the inherent parallelism in an application. As expected, the benchmarks fall into two main classes: integer benchmarks with restart rates that are much greater than one and numerical benchmarks with restart rates less than one. The combination of the coverage, the restart rate, the amount of work that is lost due to the restarts, and the speculation software overheads determines the fraction of the time the processors are doing useful work (CPU utilization), which ultimately determines performance.

A key architectural metric is the size of the speculative write state. This metric is important because it indicates the feasibility of implementing a Hydra CMP architecture with reasonably sized speculative write buffers that achieves good performance. Table 3 lists the number of 32 byte cache line buffers required to hold the maximum size write state for the eleven benchmarks. The results indicate that a buffer size of 64 lines (2KB) per processor is sufficient for all but the alvin and ear benchmarks. These two applications would require 8KB write buffers for maximum performance. However, it would be possible to reduce the memory requirements of the alvin benchmark to an arbitrary degree simply by using loop iteration chunking on the inner loop of alvin, as described in Section 5.2.3, instead of our original scheme of speculating on the outer loop. Lastly, the Hydra memory system is capable of handling overflows simply by temporarily halting execution on the speculative processor that experiences the buffer overflow. Thus, it may be possible to capture the write state from most iterations using a smaller buffer, as we showed in [7].

5. IMPROVING SPECULATION PERFORMANCE

There are four key problems that need to be addressed to improve speculation performance: 1) reducing the performance losses from the speculation software overheads, 2) reducing the performance impact of inter-thread communication through memory, 3) increasing the amount of parallelism in the program that is exposed to the speculation system, and 4) reducing both the number of violations

	Routine	Use	Procedure and Loops Overhead	Loop-only Overhead
Procedures	Start Procedure	Forks off the code following a procedure call to another processor, speculatively	~70	—
	End Procedure	Completes processing for a procedure that has forked off its completion code, and starts running another speculative task on the CPU	~110	—
Loops	Start Loop	Prepares the system to speculatively execute loop iterations, and then starts execution	~70	~30
	End of each loop iteration	Completes the current loop iteration, and then attempts to run the next loop iteration (or speculative procedure thread, if present)	~80	12
	Finish Loop	Completes the current loop iteration, and then shuts down processing on the loop	~80	~22
Support	Violation: Local	Handles a RAW violation committed by this processor	~25	7
	Violation: Receive from another CPU	Restarts the current speculative thread on this processor, when a less speculative processor requires this	~80	7
	Hold: Buffer Full	Temporarily stops speculative execution if the processor runs out of primary cache control bits or secondary cache buffers	15	12
	Hold: Exception	Pauses the processor following a SYSCALL instruction, until it is the non-speculative, "head" processor	25 + OS time	17 + OS time

Table 4. Overheads of key speculation software routines.

and the work lost when violations occur. We propose a simple solution to the first problem in Section 5.1. Section 5.2 outlines code motion techniques that help attack both the third and fourth problems. Finally, hardware techniques to address the fourth are presented in Section 5.3. The second problem requires fundamental changes to the underlying interprocessor communication mechanism which are beyond the scope of this paper.

5.1. Loop-only Speculation

The protocol software required to handle loops is much simpler than the protocol software for procedure calls. As soon as a loop starts, its iterations are distributed among the processors in a simple, round-robin fashion. In contrast, procedure calls occur one at a time, on arbitrary processors, in an unpredictable pattern. As a result, the control software must be able to handle a complex, dynamically changing list of speculative threads. This complexity does not lend itself to simple, fast software control routines, as the routines must maintain a dynamic list of threads. Being able to do both loops and procedures simultaneously increases the complexity of both, slowing down the system even more. By limiting speculation to loops alone, we can dramatically simplify the control routines so they require many fewer instructions. Table 4 shows the overhead reductions we were able to obtain by simplifying several key loop-control routines so that they would only work in an environment without procedure speculation. The savings obtained by simplification of the routine that must be called at the end of each loop iteration and the violation-processing routines have the most impact on overall performance.

5.2. Using Feedback From Violation Statistics

Our simulator produces output that explicitly identifies loads and stores in the application that cause violations. In a real system, similar feedback could be obtained by adding speculative load PC memory to each processor, broadcasting store PCs along with data on the write bus, and then interpreting the results from these hardware structures using instrumentation code built into the speculation software routines (at profiling time only — normally the overhead imposed by such code could be eliminated). We developed a tool to translate the load and store addresses produced by the simulator to source code locations, for easier analysis. Armed with this information, we attempted to reduce the number of violations caused by the most critical dependencies using synchronization, code motion, and loop transformations. While we used the feedback information from the violation statistics to make manual modifications to our benchmarks, many of these modifications could be performed using a feedback-directed compiler targeted to a speculatively parallel architecture.

5.2.1. Explicit Synchronization

The first technique we used to minimize effects from these critical data dependencies was to add explicit synchronization to cause the dependent thread to stall instead of causing a dependency violation. This was achieved simply by adding a way to issue a non-speculative load instruction even while the processor is executing speculatively. As depicted in Figure 3, this special load may be used to test lock variables that protect the critical regions of code in which pairs of loads and stores exist that cause frequent dependency violations. Before entering a critical region, synchronizing code spins on the lock variable until the lock is released by another processor. Because the special load is non-speculative, a violation does not occur when the lock is released by a store from another processor. Once the lock is freed, the speculative processor may perform the load at the beginning of the critical region. Finally, when a processor has performed the store at the end of the region, it updates the lock so that the next processor may enter the critical region. This process eliminates all restarts caused by dependent load-store pairs in the critical region, at the expense of forcing the speculative pro-

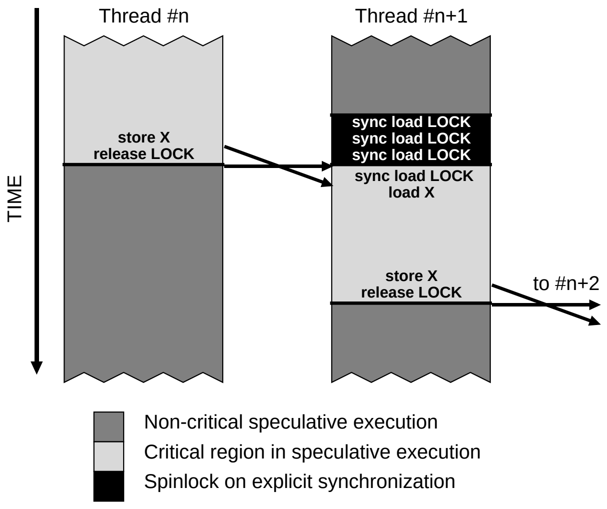

Figure 3. Explicit synchronization.

cessors to serialize through the critical regions, eliminating any possibility of finding parallelism there. The lock handling code also adds a small software overhead to the program.

Similar synchronization mechanisms have been proposed before. In [19], special loads and stores were used to pass data between processors directly and perform explicit synchronization at the same time. This method avoided the overhead of extra synchronization code, but required more complex hardware synchronization mechanisms to handle the special loads and stores. An all-hardware data dependence prediction and synchronization technique was explored in [13] for use with the Multiscalar architecture. Special hardware structures tracked dependencies and then automatically used synchronizing hardware to prevent restarts due to dependent load-store pairs. This hardware-based technique allows a limited degree of automatic synchronization even without programmer intervention, and it would be adaptable for use on a Hydra-like architecture. An alternative design for a superscalar architecture that tracks sets of stores that commonly supply data to a following, dependent load was proposed in [2]. Finally, we presented some preliminary results on the use of synchronization with the compress benchmark in [7].

5.2.2. Code Motion

While explicit synchronization prevents critical dependencies from causing violations, it also forces the speculative processors to serialize their execution. For small critical regions, this is perfectly acceptable, but for large ones it can easily eliminate all of the parallelism that speculative threads are attempting to exploit. To avoid this situation the second technique we use to improve speculation performance is to manually move source code lines with dependent loads and stores in order to shrink the critical path between frequently violating load-store pairs. This makes it possible to reduce the number of violations and often increases the inherent parallelism in the program by lengthening the sections of code that could be overlapped on different processors without causing violations.

This works in two ways. At the tops of critical regions, loads can sometimes be delayed by rearranging code blocks in order to move code without critical dependencies higher up in the loop body. However, this is usually only possible in large loops built up from several non-dependent sections of code that can be interchanged freely. More importantly, we were frequently able to rearrange code to make stores to shared variables occur earlier. Induction variables are an obvious target for early stores. Since the store that updates the induction variable is not dependent upon any computation within the loop, these can safely be moved to the top of the loop body. When this is done, a “local” copy of the old value is made at the same time, with a small amount of additional code, and this local copy is used throughout the remainder of the loop body while the “global” variable is used by other processors starting their own loop iterations. Other variables, that *do* depend upon results calculated in a loop iteration, will not be improved as dramatically by scheduling their critical stores early, but performance can often still be improved significantly over unmodified code using these techniques.

It should also be noted that code motion on a speculative processor is somewhat different from that on a conventional multiprocessor, since only the most critical loads and stores of variables need to be moved to reduce the potential for restarts. For example, variables that often — but not always — act like induction variables can be speculatively treated like induction variables at the top of the loop. Should they later act differently, the variable may be re-written, probably causing restarts on the more speculative processors. However, as long as the value computed and written early is used most of the time, the amount of available parallelism in the program may be vastly increased.

For the measurements taken in this paper, we did all code motion by hand based on violation statistics provided by our simulator. However, while synchronization may easily slow down a program if implemented in an improper fashion, code motion will rarely degrade performance. Hence, a feedback-directed compiler could perform this job almost as well as a human programmer by aggressively moving references that frequently cause violations within speculative code, up to the limits imposed by existing data and control dependencies, in order to reduce the critical path between dependent load-store pairs.

5.2.3. Loop Body Slicing and Chunking

More radical forms of code motion and thread creation are possible by breaking up a loop body up into smaller chunks that execute in parallel or its converse, combining multiple speculative threads together into a single thread. With the former technique, loop slicing, a single loop body is spread across several speculative iterations running on different processors, instead of running only as a single iteration on a single processor. In the latter case, loop chunking, multiple loop iterations may be chunked together into a single, large loop body. Loop bodies that were executed on several processors are combined and run on one. This generally only results in better performance if there are no loop-carried dependencies besides induction variables, which limits the versatility of loop chunking. However, if a parallel loop can be found, chunking can allow one to create speculative threads of nearly optimal size for the speculative system. Figure 4 shows conceptually how slicing and chunking work.

While loop chunking only needs to be performed if the body of the loop is so small that the execution time is dominated by speculation overheads, the motivation for loop slicing are more complex. The primary reason is to break down a single, large loop body, made up of several fairly independent sections, into smaller parts if they are more optimally sized for speculation. In a very large loop body a single memory dependence violation near the end of the loop can result in a large amount of work being discarded. Also, the large loop body may overflow the buffers holding the speculative state.

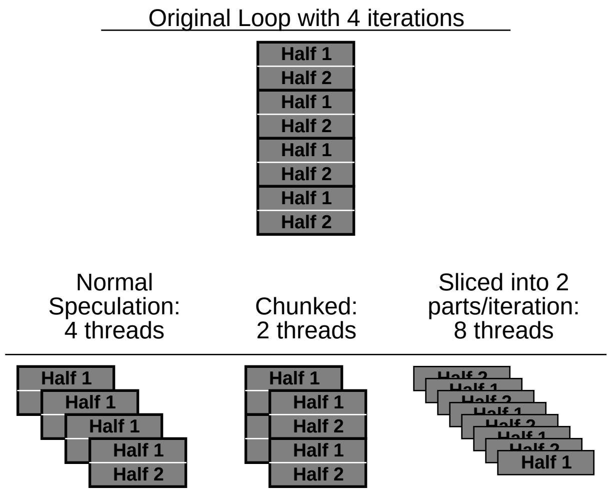
Original Loop with 4 iterations
Half 1
Half 2
Normal Speculation: 4 threads
Chunked: 2 threads
Sliced into 2 parts/iteration: 8 threads

Figure 4. Loop body chunking and slicing.

Buffer overflow prevents a speculative processor from making forward progress until it becomes the head, non-speculative processor, so this should normally be avoided whenever possible. Loop slicing is also a way to perform code motion to prevent violations. If there is code in the loop body that calculates values that will be used in the next loop iteration and this code is not *usually* dependent upon values calculated earlier in the same loop iteration, then this code may be sliced off of the end of the loop body and assigned to its own speculative thread. In this way, the values calculated in the sliced-off region are essentially “precomputed” for later iterations, since they are produced in parallel with the beginning of the loop iteration. The advantage of slicing over normal code motion is that no data dependency analysis is required to ensure it is legal to perform the code motion, since the violation detection mechanism will still enforce all true dependencies that may exist.

Loop chunking may be implemented in a compiler in a similar manner to the way loop unrolling is implemented today, since both are variations on the idea of combining loop iterations together. In fact, the two operations may be merged together, so that a loop is unrolled as it is chunked into a speculative thread. As a result, adding this capability to current compilers should not be difficult. Effective slicing, however, will require more knowledge about the execution of a program than loop unrolling, although the ability to analyze the control structure of the application combined with violation statistics should be sufficient. Slicing should not be performed indiscriminately because it may allow speculative overheads to become significant and/or it may result in significant load imbalance among the processors if some slices are much larger than others. The former problem will obviously slow the system down, while the latter problem will degrade performance in the current Hydra design, since a speculative processor that completes its assigned thread must stall until it becomes the head processor, wasting valuable execution resources in the process. This problem can only be overcome by making the speculation system allow multiple speculative regions within each CPU, so that a processor that completes a speculative region may immediately go on to another while it waits to commit the results from the first. However, implementing this capability has significant hardware overheads [7].

5.3. Hardware Checkpointing

Another technique to prevent large quantities of work from being discarded after violations is to provide a mechanism for taking extra checkpoints in the middle of loop iterations. The basic speculation system takes a checkpoint of the system state only at the beginning of a speculative thread, and must return to that checkpoint after every violation. A possible technique to save time on restarts is to take an extra checkpoint whenever a violation-prone load actually occurs. That way, if the load really does cause a violation later, the work done by the speculative processor before the bad load doesn’t need to be discarded, because the processor will be able to return to the machine state just before the load.

Hardware checkpointing requires four key mechanisms. First, a hardware mechanism must be installed to track the PCs of loads that tend to cause violations. When one of these loads is encountered, the mechanism must signal the speculative hardware to take a checkpoint. If load PCs are already recorded in order to allow violation statistics tracking, as described in Section 5.2, this will be a fairly minor addition. The other three items are necessary to take the checkpoints themselves. Backup copies of the register file are necessary to hold the register file state at the checkpoint. These backups are not normally necessary, since checkpoints are usually made during speculation software routines, when the register file is in a well-known state. However, checkpoints made at violating loads may come at arbitrary times, when the register file is in an arbitrary state. From a hardware point of view, this addition is the most troublesome, since it requires modifications to the existing processor’s register file, where many high-speed signals may be affected. Less problematic are additional sets of bits in the primary cache to indicate that words have been loaded (“read” bits) and additional, smaller store buffers before the secondary cache. For *each* checkpoint, a new set of “read” bits and store buffers are needed to track the speculative state. The other speculation control bits in the primary cache may be safely shared among all the checkpoints, to reduce the hardware requirements. The store buffers do not need to grow significantly in overall memory size, because each original monolithic speculative thread now has its stores scattered over several smaller checkpoint store buffers. Only a relatively small overall size increase is necessary to handle the inevitable load imbalance that will occur among the different checkpoints.

6. IMPROVED SPECULATION PERFORMANCE

After applying the software optimizations to the benchmarks and the hardware enhancements to the Hydra simulator, we collected a new set of performance results. These results are shown in Table 5 and Figure 5. From these results it is clear that the lower overheads of loop-only speculation provide significant performance improvements for most benchmarks. The most dramatic improvement is seen with the wc benchmark, whose performance more than doubles. This is consistent with wc’s small loop body, that was completely dominated by the speculation software overheads in the base run-time system.

We were able to optimize five (compress, eqntott, m88ksim, mpeg, and cholesky) of the eleven benchmarks using the optimizations described in Section 5.2.

	Application	Procedures and Loops	Loops Only	After Software Optimization	With Hardware Checkpoints
General Integer	compress	1.00	1.42	1.57	1.71
	eqntott	1.15	—	1.75	1.8
	grep	2.65	2.86	—	2.85
	m88ksim	1.05	1.13	1.45	1.56
	wc	0.62	1.57	—	1.64
Multimedia Integer	ijpeg (compression)	1.31	1.56	—	1.6
	mpeg (decoding)	0.90	0.96	2.14	2.14
Floating Point	alvin	2.79	3.02	—	—
	cholesky	2.71	2.85	3.16	3.25
	ear	3.44	3.87	—	3.88
	simplex	1.83	2.67	—	2.72

Table 5. Performance with the various system enhancements.

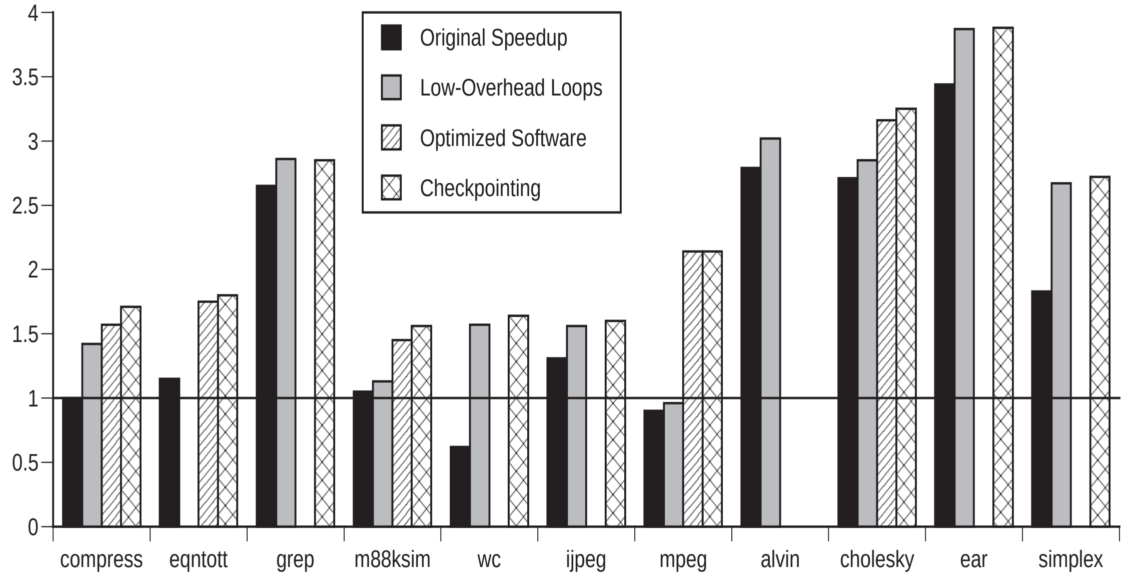

Figure 5. A performance comparison of the baseline system and the enhanced systems.

compress: Synchronization was added to compress to prevent the key loop-carried dependency from causing violations. This reduced the number of restarts by a factor of 3 (to 2.1 per iteration), allowing a greater speedup.

eqntott: The optimization of eqntott is a actually a change to the speculation software. Instead of speculating on all procedure calls, we modified the system so that only calls to the recursive *quicksort* procedure generate new speculative threads. This eliminates the performance losses from blind procedure speculation of procedures with little parallelism.

m88ksim: The large loop body in m88ksim was sliced to expose parallelism within the loop body and to allow code motion for a data value that is read early in each iteration, but is normally computed late in the previous iteration. This code motion decreased the number of restarts by a factor of 5 (to 3.1 per iteration), as the amount of available parallelism was dramatically increased.

mpeg: The performance of the mpeg-2 benchmark is improved substantially by moving an induction-like variable update from the end of the loop body to before the time-consuming DCT step. This code motion makes it possible for DCTs of multiple macroblocks to overlap. The same performance improvement was also achieved with slicing. As with m88ksim, the additional parallelism exposed by these modifications reduced the number of restarts by a factor of 5 (to 280 per iteration). Our speculatively parallel version of this code was actually faster than the macroblock-based hand-parallelized version in [9], because the speculative system does not have to synchronize between processors in a conservative fashion.

Category	Procedures and Loops	Loops Only	After Software Optimization	With Hardware Checkpoints
General Integer	1.04	1.45	1.73	1.81
Multimedia Integer	1.07	1.19	1.80	1.80
Floating Point	2.56	3.04	3.12	3.15
Total	1.33	1.71	2.10	2.14

Table 6. Harmonic mean summaries of performance results.

cholesky: In the cholesky substitution program stage, the direction of the inner loop is reversed so that the loop carried dependency occurs on the last loop iteration, instead of the first. This allows the inner loop iterations from consecutive outer loop iterations to overlap.

These modifications resulted in significant performance improvements in these benchmarks, as is shown in Table 5. In particular, we were able to achieve a speedup of 2.14 on mpeg, whereas the baseline system slowed down.

With eight hardware checkpoints per processor, the performance improvements from hardware checkpointing on the optimized software are quite modest. Furthermore, mixing explicit software synchronization and hardware checkpointing is not useful, based on our observations with the compress benchmark. In effect, hardware checkpointing is an optimistic form of synchronization — a form of "wait-free" synchronization. As a result, software synchronization just provides additional overhead, slowing down the system while supplying little or no benefit, as it is redundant.

Table 6 summarizes the gains from these improvements for each of the major application classes using the harmonic means

7. CONCLUSIONS

In this paper, we have shown that a few simple optimization techniques can dramatically improve the performance of the Hydra CMP on sequential programs executed as a sequence of speculative threads. We achieved a 60 percent overall performance improvement on the eleven benchmarks and a 75 percent performance improvement on general integer benchmarks.

We began with our baseline architecture from [7], which extracts speculative parallelism from procedures and loops. This architecture is effective at exploiting speculative parallelism when large amounts of parallelism are present in the application, as our results for the floating point benchmarks demonstrate. However, it has difficulty extracting parallelism from conventional integer applications, sometimes slowing the applications down. A key problem with the baseline Hydra speculation implementation is the high overhead incurred by the speculation software to properly handle both procedures and loops. This problem is especially acute with integer applications because the number of overhead-incurring restarts is higher due to a larger number of dependencies between loop iterations and among procedures. Lastly, procedures typically degrade performance because too much time is spent starting speculative threads for procedures that turn out to have little parallelism.

As an initial optimization technique, we eliminated procedures and focused on making loop iteration speculation run as quickly as possible, with much lower speculation software overheads. This effort was very successful and resulted in significant speedups on most of our applications. Violation statistic gathering mechanisms allowed us to identify data-dependent pairs of loads and stores in the benchmarks that frequently resulted in violations. On half of our applications, we were able to use this information to guide several software optimization techniques that modified parts of the original source code. These modifications also improved performance significantly. Finally, we implemented a hardware checkpointing scheme that provides nearly optimal performance by eliminating most of the delay caused by frequently violating loads. The performance gains that we were able to achieve with hardware checkpointing would probably not justify the investment in hardware required to implement this technique.

Eqntott was the only benchmark that whose performance was not improved by the software optimization techniques. Much of the execution time in eqntott is dominated by a recursive quicksort procedure call instead of an iterative loop of reasonable size, and so our loop-based techniques were of little use. However, by manually limiting procedure speculation to the key quicksort routine, instead of speculating throughout the program, we were able to get a significant improvement. This strongly indicates that procedure speculation is still a viable alternative for environments where it can be used judiciously (Java is one such environment [1]). This will be especially true if the overheads can be reduced through the addition of specialized hardware to accelerate the procedure speculation software.

Overall, our results demonstrate that there is a promising migration path from our current, simple sequential-to-speculatively parallel conversion tools, such as hydracat, to more sophisticated compilation tools that can optimize code for a speculatively parallel architecture. Today, these optimizations still require programmer intervention, but it should be possible to automate them significantly, allowing a speculatively parallel CMP to be competitive with a conventional wide-issue out-of-order machine on many integer applications [14]. At the same time, the CMP should be able to achieve higher performance on inherently parallel applications, such as floating point numerical code and multimedia applications, than a comparable cost superscalar architecture.

8. ACKNOWLEDGMENTS

This work was supported by DARPA contracts DABT 63-95-C-0089 and MDA 904-98-C-A933.

9. REFERENCES

[1] M. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded Java programs," *Proceedings of Parallel Architectures and Compilation Techniques (PACT 98)*, pp. 176–184, Paris, France, October 1998.

[2] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," *Proceedings of 25th Annual*

International Symposium of Computer Architecture, pp. 142–153, Barcelona, Spain, June 1998.

[3] M. Franklin and G. S. Sohi, “The expandable split window paradigm for exploiting fine-grain parallelism,” *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58–67, Gold Coast, Australia, May 1992.

[4] M. Franklin and G. Sohi, “ARB: A hardware mechanism for dynamic reordering of memory references,” *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[5] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, “Speculative versioning cache,” *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.

[6] L. Hammond and K. Olukotun, *Considerations in the Design of Hydra: a Multiprocessor-on-a-Chip Microarchitecture*, Stanford University Computer Systems Laboratory, Technical Report No. CSL-TR-98-749, Stanford University, February 1998.

[7] L. Hammond, M. Willey, and K. Olukotun, “Data speculation support for a chip multiprocessor,” *Proceedings of Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pp. 58–69, San Jose CA, October 1998.

[8] B. Kernighan and R. Pike, The Practice of Programming. Reading, Massachusetts: Addison-Wesley, 1999.

[9] E. Iwata and K. Olukotun, *Exploiting coarse-grain parallelism in the MPEG-2 Algorithm*, Stanford University Computer Systems Laboratory, Technical Report CSL-TR-98-771, September 1998.

[10] T. Knight, “An architecture for mostly functional languages,” *Proceedings of the ACM Lisp and Functional Programming Conference*, pp. 500–519, August 1986.

[11] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, “Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor,” *Proceedings of 25th Annual International Symposium on Computer Architecture*, pp. 306–317, Barcelona, Spain, June 1998.

[12] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A tool for evaluating and synthesizing Multimedia and communications systems,” *Proceedings of 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[13] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic speculation and synchronization of data dependences,” *Proceedings of 24th Annual Int. Symp. Computer Architecture*, pp. 181–193, Denver, CO, June 1997.

[14] K. Olukotun, K. Chang, L. Hammond, B. Nayfeh, and K. Wilson, “The case for a single chip multiprocessor,” *Proceedings of the 7th Int. Conf. for Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 2–11, Cambridge, MA, 1996.

[15] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, *Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors*, Stanford University Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University, February 1997.

[16] J. Oplinger, D. Heine, M. Lam, and K. Olukotun, *In Search of Speculative Thread-Level Parallelism*, Stanford University Computer Systems Laboratory Technical Report CSL-TR-98-765, July 1998.

[17] W. H. Press, S. A. Teulosky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992.

[18] G. Sohi, S. Breach, and T. Vijaykumar, “Multiscalar processors,” *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, Ligure, Italy, June 1995.

[19] J. G. Steffan and T. Mowry, “The potential for using thread-level data speculation to facilitate automatic parallelization,” *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.