Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor

Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh,
Monica S. Lam and Kunle Olukotun

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Thread-level speculation (TLS) makes it possible to parallelize general purpose C programs. This paper proposes software and hardware mechanisms that support speculative thread-level execution on a single-chip multiprocessor. A detailed analysis of programs using the TLS execution model shows a bound on the performance of a TLS machine that is promising. In particular, TLS makes it feasible to find speculative do across parallelism in outer loops that can greatly improve the performance of general-purpose applications. Exploiting speculative thread-level parallelism on a multiprocessor requires the compiler to determine where to speculate, and to generate SPMD (single program multiple data) code.We have developed a fully automatic compiler system that uses profile information to determine the best loops to execute speculatively, and to generate the synchronization code that improves the performance of speculative execution. The hardware mechanisms required to support speculation are simple extensions to the cache hierarchy of a single chip multiprocessor. We show that with our proposed mechanisms, thread-level speculation provides significant performance benefits.

1 Introduction

The desire for higher microprocessor performance on general purpose programs and the increasing gate densities offered by semiconductor technology have driven hardware designers and compiler writers to investigate aggressive techniques for exploiting program parallelism. Current software and hardware techniques focus on exploiting instruction level parallelism from a single execution thread using multiple instruction issue, a combination of static and dynamic instruction scheduling, register renaming and speculative execution. While these techniques are successful at achieving better program performance than a simple CPU pipeline, studies show that the amount of program parallelism that can be exploited using these techniques alone is limited because there is only one thread of control in the model of computation [8]. To overcome this limit we must a consider computational model that includes multiple flows of control. However, this requires parallel applications. While it is possible that applications whose performance is important will be parallelized by hand, automatic parallelization of general purpose codes is required before single-chip multiprocessor architectures become widespread. To date, automatic parallelization has only been successful on numeric FORTRAN programs [3][6] where the data dependencies can be analyzed statically. Parallelizing general purpose non-numeric C programs has not been successful because it requires pointer-disambiguation to statically analyze the dependencies. While a significant amount of progress has been made in this area [13], it is still and unsolved problem. Thus for parallelization of general purpose C programs to succeed we need hardware and software mechanisms that eliminate the need for data dependence analysis to ensure correct program execution.

This paper describes a model of computation called thread-level speculation (TLS) that uses speculative execution across multiple threads of control to overcome the current limitations of software and hardware at uncovering and exploiting parallelism in general purpose programs. The support for TLS can be added to a single-chip multiprocessor that already has low interprocessor communication overhead. This support consists of hardware to keep speculative memory state separate from non-speculative state, a mechanism for committing speculative state under software control, a method of detecting data hazards and flushing the speculative state, and a mechanism for notifying the software when a hazard is detected so that recovery can take place. Hardware support for TLS makes it possible for the compiler to be aggressive instead of conservative in uncovering parallelism. Using the TLS computation model, the compiler takes sequential programs and automatically generates parallel code, ignoring potential data dependencies, the TLS hardware support guarantees correct execution if dependencies actually exist. TLS works best when the data hazards occur infrequently enough so that the performance gained from the increased parallelism is greater than the performance lost from the recovery overhead. To maximize performance, the compiler should determine where speculation is beneficial and insert synchronization between processors where it is not.

Our model of computation is similar to the multiscalar paradigm proposed by Sohi et. al. [11]. The multiscalar paradigm was the first constructive model of a processor with multiple threads and speculation and has influenced this work significantly. Many of the results reported here are also applicable to their proposed system. The primary difference between the multiscalar approach and our proposed system is that the multiscalar machine operates on an annotated sequential program. The architecture is tailored to executing relatively fine-grain speculative execution, with the machine having a single logical register file and special mechanisms to ensure the coherence between the register files. In contrast, we approach speculation with multiple threads as an extension of a multiprocessor that supports fine-grain sharing. Our objective is to build a machine that can cost-effectively execute different programs at the same time, as well as single applications that have been statically or speculatively parallelized. We are interested in exploiting both fine-grain and coarse-grain parallelism. We expect that the compiler is fully aware of the presence of multiple processors, generating code that keeps separate states in their respective register files to improve performance. The machine, intended to be implemented on a single chip, provides efficient generic support for communication between processors, as well as specialized support for speculative execution. Because the software plays a bigger role, our system is more versatile and has lower hardware overheads.

The research reported in this paper has several major components. These include: defining the abstract computation model, studying the model analytically and experimentally using a suite of programs, developing a feedback-driven optimizing compiler for a machine with hardware support for speculation, specifying the processor in enough detail to run operating system code, and demonstrating its functionality through the complete simulation of some representative programs.

A speculative execution model. We first define an abstract machine for speculative execution, and show how this machine can be used to find speculative parallelism across iterations in a loop. We also lay the ground work for subsequent discussion by describing the issues that determine the performance of speculative execution.

Analysis of speculative parallelism in programs. Speculation with multiple threads is a new computation paradigm that has not been well explored, so our intuitions about the performance effects of speculation and when speculation should be employed is not well developed. To gain this intuition we have devised an experiment designed to discover the inherent amounts of speculative parallelism in general purpose programs. We define a *perfect speculation machine* that chooses the best loops to execute speculatively, and issues each load instruction at the optimal time. The simulation of this perfect speculation machine is different from previous limit studies in that it models a realistic form of computation where each individual iteration is executed sequentially and relies on an implementable speculation mechanism. We analyze the results of the simulations in detail to discover characteristics of programs executing under speculation.

Compiler Optimizations for speculation. We explain how minor coding styles can greatly affect the performance of speculation, and describe a simple compiler algorithm that eliminates some of the artificial dependencies in a program. The algorithm is fully implemented in the SUIF compiler system. The information from the simulation study is used by our compiler to make two key optimization decisions: when to speculate and where to synchronize. The compiler generates speculation code for the loops that the perfect speculation machine chooses, and also introduces synchronizations between operations that frequently occur in the simulations. While the simulation is slow compared to normal compilation speeds, the code optimization process is fully automatic, and the time taken is negligible when compared to the manual performance tuning time spent on software products.

A detailed hardware design. Although the computation model is dramatically different from that of existing multiprocessors, we show that the machine can be realized by modifications to the existing single-chip multiprocessor design. We focus on the overall design, addressing all the issues required to support system software, and demonstrate its functionality through full simulations of programs running with a complete operating system. Determining precise memory system parameter values, such as the cache sizes, are outside the scope of this paper.

2 Computation Model

In this section, we present an abstract machine for speculation; discussions on its implementation will be presented in Section 5.

2.1 Speculative Execution Model

Our goal is to support potentially coarse-grain speculative execution, where each unit of code that we speculate across can itself contain other loops and function calls. To fully support this coarse-grain model, each processor must be running an independent thread, with its own stack.

The computation consists of speculative regions which are entered and exited by a processor with explicit *begin-speculation* and *terminate-speculation* operations. A speculative computation is decomposed into sequential tasks, numbered consecutively according to their execution order. At the beginning of the speculation, each of the P processors in the machine are given one of the first P sequential tasks to execute. After a processor completes task t, it proceeds to execute task $t + P$. Each task has a state; the processor executing the task with the lowest ID is said to own the *current state*; all other tasks t have a *speculative state*, which is identical to the speculative state of task t - 1, except that all the memory locations written by task t are replaced with the latest value written by task t.

All read and write operations executed by a processor in speculation mode are directed to the processor's state, which may be current or speculative. There are two flavors of write operations: a normal a *write* operation which may trigger a *RAW hazard* on another processor, and a *synch-write* operation used for synchronization primitives that simply writes to a location. A RAW hazard occurs when a read operation speculatively executes before the write operation that should produce the value that is read. Note that if the value of the write operation was overwritten by the current task before the read in question, then the original write operation does not cause a RAW hazard. When a RAW hazard is detected, the speculative task containing the read and all those tasks following it need to be rolled back. For roll-back, the machine passes the appropriate task number to a software handler so that the task can be restarted.

A task can issue three forms of commit statements. The semantics of each of these commits requires that the task issuing the commit must wait for its speculative state to become current. This requires waiting for all previous tasks to *commit_and_advance*. The simple *commit* operation has the effect of forcing serialization at the commit point. It is used to enforce sequentiality for operations like operating system calls that can not be executed speculatively. At the end of each task, a *commit_and_advance* operation occurs. After waiting for previous tasks to complete, process' task ID is incremented by P and the speculative state of the next task is made the current state. Finally, a *terminate-speculation* operation also has commit semantics and waits for all previous tasks to complete. After waiting, all speculative states of subsequent tasks are flushed and the other processors are given notice to stop operating in speculative mode. All the primitives in our abstract computation model are summarized in Table 2.1 We use p to denote the processor executing the instructions, and t_p to denote its task ID.

Operations	Semantics
Start_Speculation (n)	Start the speculative mode of execution $t_p = n$ Current state = state of processor with task 0.
Read	Read from p's state.
Synch_Write	Write to p's state.
Write	Write to p's state. If task $t > t_p$ has previously read this data, flush speculative states of task t and beyond, and restart the tasks.
Commit	Wait until p's state becomes the current state
Commit_and_Advance	Commit $t_p = t_p + P$ Current state = speculate state of processor executing task t_p+1
Terminate_Speculation	Commit Flush speculative states of subsequent tasks Signal to stop speculation mode for all processors

Table 2.1 Software Primitives in Speculative Execution.

It is clear that this computation model correctly honors the data dependences of the original sequential execution. The machine ensures that all the reads will retrieve the same value as in the original sequential order by rolling back the computation if those dependencies are violated. This speculative execution model is quite powerful, in that a RAW hazard can only occur if a true dependency is violated. WAW and WAR hazards can not occur because a speculative write does not change the state of any preceding task.

2.2 Speculation in Loops

In this paper, we only focus on exploiting parallelism across loop iterations. Loops are a good target for many reasons: loops are where programs spend most of their execution time; there is typically a relatively small amount of state that needs to be transferred between iterations; partitioning loop iterations is a simple way to achieve reasonable load balance across the processors. We are interested not just in finding *do-all* parallelism, which requires iterations in a loop to be data and control independent of each other, but also *do-across* parallelism, where there may exist *loop-carried* data and/or control dependences between iterations. The abstract machine is a complete multiprocessor and can thus be used to support any parallelism that the compiler can extract statically. However, here we concentrate on the subject of exploiting *speculative* parallelism in loops.

Generating correct code for our abstract machine is straightforward. The compiler generates a single program multiple data (SPMD) program with a master thread responsible for executing the sequential portions of the program. All the slaves execute an infinite loop, waiting for the master's order to execute. When the master thread reaches a speculatively parallelized loop, it signals the other slave threads to participate in executing the loop. The master passes to the slaves a small *loop execution context*, which consists of the starting location of the code to execute, a starting iteration number, and the master's frame pointer. The variables local to the master's thread are accessed through the master's frame pointer rather than the thread's own local pointer. Our compiler analyzes the code so that variables that can be privatized, which includes numerous temporary variables, are allocated directly on the slave's stacks. As a source-to-source compiler, our current implementation does not actually pass the frame pointers between the threads, but rather, we include in the loop execution context the values of variables read in the loop, and addresses to variables that may be modified. The code is modified to access the master's variables through the loop execution context.

The SPMD code is parameterized by the processor identification number so that the iterations are cyclically distributed across the processors. That is, on a machine with four processors, thread 0 executes iterations 0, 4, 8,..., etc. It is not necessary that the number of iterations be known at the time the loop execution is started, thus this model is general enough to handle while loops. If the code is a FOR loop with independent iterations, the threads do not need to communicate at all as they can step through their assigned iterations independently.

At the beginning of the loop, every thread issues a *start_speculation(i)* operation, where *i* is the task ID, and starts executing its assigned iteration speculatively. During the course of execution, before a thread executes a system call, it first issues a *commit* operation to ensure that the system calls are made in the original sequential order. If it reaches the end of an iteration, it executes a *commit_and_advance* operation and branches back to the beginning of the loop. On the other hand, if it branches out of the loop, it issues a *terminate_speculation* operation.

Figure 2.1 shows the speculative execution of a loop on three processors. The master wakes up the slaves threads and proceeds to execution 0. The slaves also start executing their respective iterations. The diagram shows that the second slave thread is delayed when it tries to issue a *commit_and_advance* statement at the end of iteration 2 until a *commit_and_advance* statement is issued for iteration 1. In this example, iteration 4 of the loop executes a “break” statement and exits the loop. However, before the break statement is executed, both iterations 5 and 6 have speculatively been completed, with iteration 6 also encountering a break statement in the loop. Since the *commit_and_advance* and the *terminate_speculation* operations are not allowed to proceed until the thread that issued these operations is current, the *terminate_speculation* operation issued by iteration 4 correctly flushes the speculative states of iterations 5 and 6.

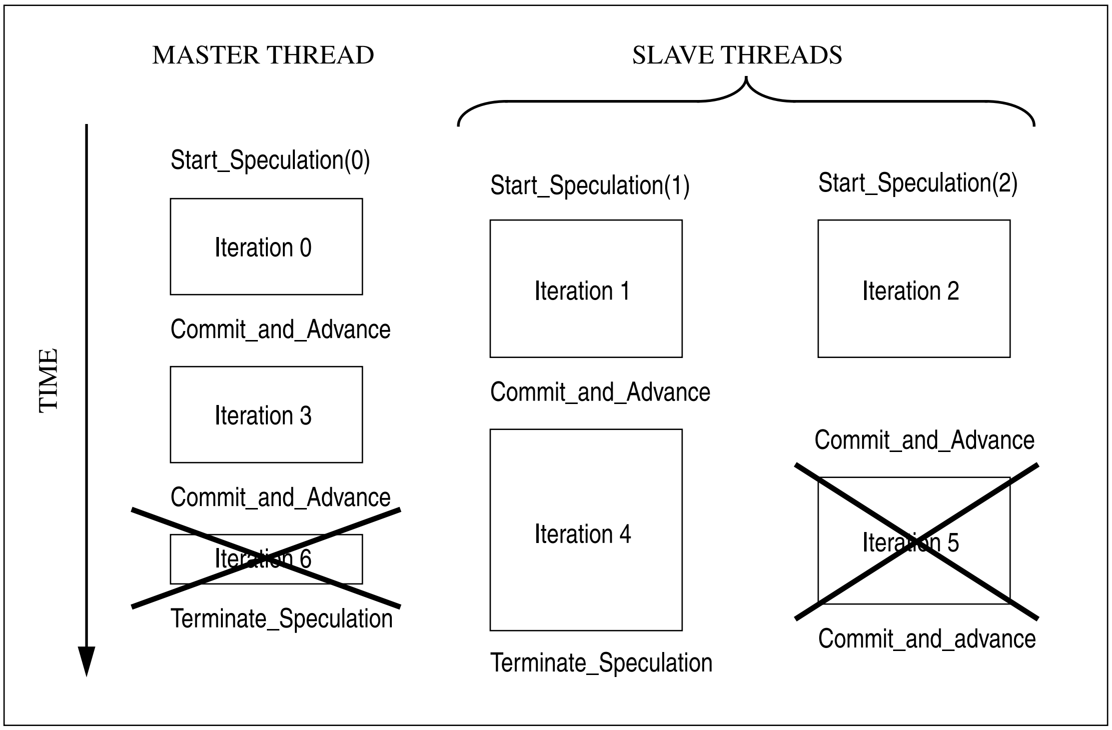

Figure 2.1 Example of a thread-level speculative execution

The compiler provides an exception handler, which is called when the hardware detects a RAW hazard in speculation mode. The exception handler restores the state of the thread to point at the beginning of the iteration. The restoration requires flushing the thread’s speculative state and restoring only the loop execution context, with the starting iteration set to the iteration the thread is speculating on.

In addition to speculative read and write operations, the compiler also uses the *synch_write* operations to enforce synchronization across iterations when there is a definite data dependence between the iterations. The dependent task reads a synchronization variable and does not proceed until the preceding task signals that the data is ready via a *synch_write* operation. By using a *synch_write* operation, the dependent task can read the synchronization variable before the *synch_write* occurs without causing a roll back.

2.3 Analyzing Speculative Parallelism in Loops

In this section, we analyze the performance of speculative parallelism extracted from loops executing on infinite hardware. To simplify our analysis, we assume that the machine can issue a single instruction at a time, and takes only one clock cycle to execute every loop iteration. With infinite hardware, we can initiate all the iterations at the same time.

A read operation is *data dependent* on a write operation if and only if the former reads the value written by the latter; this definition is stricter than its usual meaning which only requires the read and write operations access the same location. Data dependencies that cross iteration boundaries of a loop are said to be *carried* by that loop. Suppose we have a loop-carried data dependency where the first iteration, after executing n_w instructions, writes the value read by the second iteration after executing n_r instructions. If the difference is positive, it is optimal to delay the read operation by n_w-n_r cycles. If we did not delay the read the iteration would roll back when the violation occurs which would delay the read operation by more than n_w-n_r cycles. Thus, we refer to the difference in the number of instructions executed, n_w-n_r, as the *minimum execution delay* of a dependence. The maximum of the execution delay among all the loop-carried dependences defines the total minimum execution delay for the entire iteration.

Loops with loop-carried data dependencies and positive execution delays are referred to as *DoAcross* loops. The execution time can be no shorter than the compounded execution delays across consecutive iterations, and the degree of parallelism in a loop can be estimated by the ratio between the average time to execute an iteration and the average minimum execution delay per iteration. If the iteration does not contain any loops or recursive function calls, the total number of instructions executed is bounded by the number of statements in the loops and functions it calls. Regardless of how small the minimum execution delay is, the degree of parallelism is bounded by a constant which is independent on the number of iterations in a loop. On the other hand, if the iteration contains loops or recursive function calls, the number of instructions in an iteration is unbounded and the degree of parallelism can be quite large if the minimum execution delay is small. We refer to the DoAcross loops that contain inner loops or make recursive calls as coarse-grained, and fine-grained otherwise.

The presence of loop-carried dependences does not necessarily reduce the degree of parallelism in a loop. If the execution delay is negative, it means that the write operation would have taken place before the read operation, had the two iterations been initiated at the same time. Thus dependences with only negative execution delays do not slow down the parallel execution at all. Unfortunately this case only occurs rarely in practice, and when it does, it is sometimes possible for compilers to rearrange the code to eliminate all the loop-carried data dependences.

If there are no loop-carried data dependences in an execution of a loop, the loop is said to have *speculative DoAll parallelism*, and its degree of parallelism that can be exploited is simply the minimum of the number of iterations in the loop and the number of processors in the machine.

There are two classes of DoAll loops. The first class of DoAll loops ar those loops for which the number of iterations is known in advance, then all the iterations can be initiated simultaneously knowing that each will definitely execute. The second class of Doall loops, where the number of iterations are not known in advance, includes WHILE loops or FOR loops with “break” statements. An example of such a loop is one that searches for the first element in an array that matches a key. To execute such a loop speculatively, we can initiate as many iterations as there are processors and then flush any speculative state of iterations that should not have been executed when the loop termination condition is met. The maximum degree of parallelism in the program is again the smaller of the number of iterations in the loop or the number of processors in the machine.

3 Performance with a Perfect Speculation Machine Model

To aid our understanding of speculative execution, we experiment with a perfect speculation machine. This machine has an infinite number of processors, each of which issues and executes a single instruction in each clock cycle. Assuming that only one loop is allowed to execute speculatively at any one time, the perfect speculation machine will choose optimally, for each instance of a loop, whether that loop should be speculatively parallelized or should refrain from speculating in order to allow inner loops to speculate instead. The perfect machine knows how many iterations of each speculative loop will execute, and initiates all those iterations simultaneously. It delays each read operation by the minimum amount of time necessary to satisfy true data dependences. The performance of this perfect machine places an upper bound on the performance of any hardware supporting our speculative execution model.We present an algorithm to simulate the perfect machine, and show the results of simulating a set of programs on this machine. We analyze the results of the experiments and extract information about programs relevant to this speculative execution model.

3.1 Simulation Algorithm

We use a trace-driven simulation algorithm to determine the performance of the perfect machine. The basic engine is similar to many other previous architecture limit studies where we record the latest instruction to write to a memory location so as to determine the earliest time that a read operation can execute. All data references in the user program appear in the trace in order to compute an accurate limit.

It is relatively easy to modify this basic scheme to simulate the perfect speculative execution of a single loop. First, we augment the load/store address trace with the start and end points of each iteration and each loop. To determine the minimum execution of a speculative loop, we reset the time of the simulation to 0 at the beginning of each iteration. Normally, the simulation time advances by one clock cycle per instruction. If the instruction is a load, and the time the location was last written is later than the current time, then the clock is further advanced by the minimum time needed to satisfy the dependence, i.e. the difference between the two times. At the end of an iteration, the clock time is recorded as the completion time for that iteration. The minimum execution time of a speculatively executed loop is then just the maximum completion time over all its iterations.

The complexity of the simulation algorithm lies in determining the best loop to speculate at each point in the program. Our simulation algorithm analyzes the sequential trace only once, but keeps separate simulation times for each currently active loop as if that were the only loop to be speculating. For each loop that an executed instruction is nested in (dynamically), the algorithm tracks the current iteration number and simulation time assuming that loop were executing speculatively. This per-currently-active-loop information is recorded for each location stored to in the simulation. When a load operation is encountered, it compares the current iteration counts with those of the last write to that location. The loop carrying the dependence is the first common loop nest that has different iteration counts. If the time of the store in that loop is later than the current time, we advance the simulation time of the current iteration in that loop (containing the load) by the minimum execution delay. Note that a dependence can only be carried by one loop level, and can only affect the simulation time of that loop. The read will automatically be executed after the write operations regardless of which loop in the current nest we finally choose to execute speculatively.

When an innermost loop terminates, it passes to its surrounding loop the length of its computation and the degree of parallelism it would have achieved had it been speculatively executed. By collecting all such information from its inner loops, an outer loop can determine whether it is more profitable for it speculate or to instead suppress speculation in order to allow its inner loops to speculate. It chooses the option providing the best performance and, upon termination, informs its surrounding loop of the best performance available had speculation been applied to either itself or its inner loops. Eventually this information propagates back to the outermost loop level and determines the best overall performance that loop speculation can provide. Our algorithm handles recursion by assuming that speculation can be applied only to the outermost execution of a loop that recursively invokes itself again.

In addition to the computation described above, we collect statistics deepen our understanding of the program characteristics. We identify the read and write operations that can potentially cause a cross-iteration RAW hazard, and store the information with the loop it affects. We compare the iteration number stored with the current iteration to determine the iteration distance between the dependent operations.

3.2 Simulation Results

Our experiments with the perfect speculation machine, we chose to examine eight benchmarks. None of them have ben shown to have significant amounts of DoAll parallelism in the past. For each of these benchmarks, we simulate a small input set that serves to validate the compilation as well as provide data needed for understanding patterns of parallelism. The benchmarks are summarized in Table 3.1

Table 3.1 Summary of the benchmarks and number of instructions executed

Application	Lines of code	Function	Execution Length
eqntott	3138	Equation solver from Spec92	413M
musicin	6554	Mpeg audio compression	884M
wc	37	Word count utility	3M
grep	430	Pattern matching utility	86M
espresso	15013	Logic minimization	545M
fpppp	2784	Quantum Chemistry benchmark	4,421M
diff	668	File comparison utility	110M
compress	1934	File compression from spec92/95	159M

3.2.1 Analysis of Parallel Behavior

Table 3.2 presents the overall performance of the perfect speculation machine. The programs are sorted in reverse order of their speedups to highlight the range of their behavior. The overall speedup ranges from 1.6 to 16.5, indicating that the potential performance of speculation is highly sensitive to the program.

To understand the overall behavior of a program, we analyze the behavior of the perfect speculation machine and classify how it spends its time in each of the following modes:

- speculative DoAll loops

Table 3.2 Speedups in different modes of speculative parallelism

Program	Overall Speedup	DoAll	DoAll with Early Exits	Coarse DoAcross	Fine DoAcross
eqntott	16.5	--	20.5	80.0	--
musicin	9.4	51.6	--	8.4	15.3
wc	7.8	--	--	--	7.4
grep	6.0	--	11.0	--	4.5
espresso	2.8	3.4	3.6	3.9	5.2
fpppp	2.5	--	--	--	2.6
diff	2.1	--	--	3.5	2.3
compress	1.6	5.8	--	1.5	--

- speculative DoAll loops with potential early exits
- speculative DoAcross loops with coarse-grain parallelism
- speculative DoAcross loops with fine-grain parallelism
- sequential execution.

This breakdown of the execution time enables us to understand the variation of speculative parallelism within the program itself and to correlate the performance with program characteristics. Figure 3.1 shows the breakdown for each program. For each program, we have a bar with five components indicating the percentages of the computation under the different modes. We also show in Table 3.2 the speedup for each component that occupies at least 5% of the computation time.

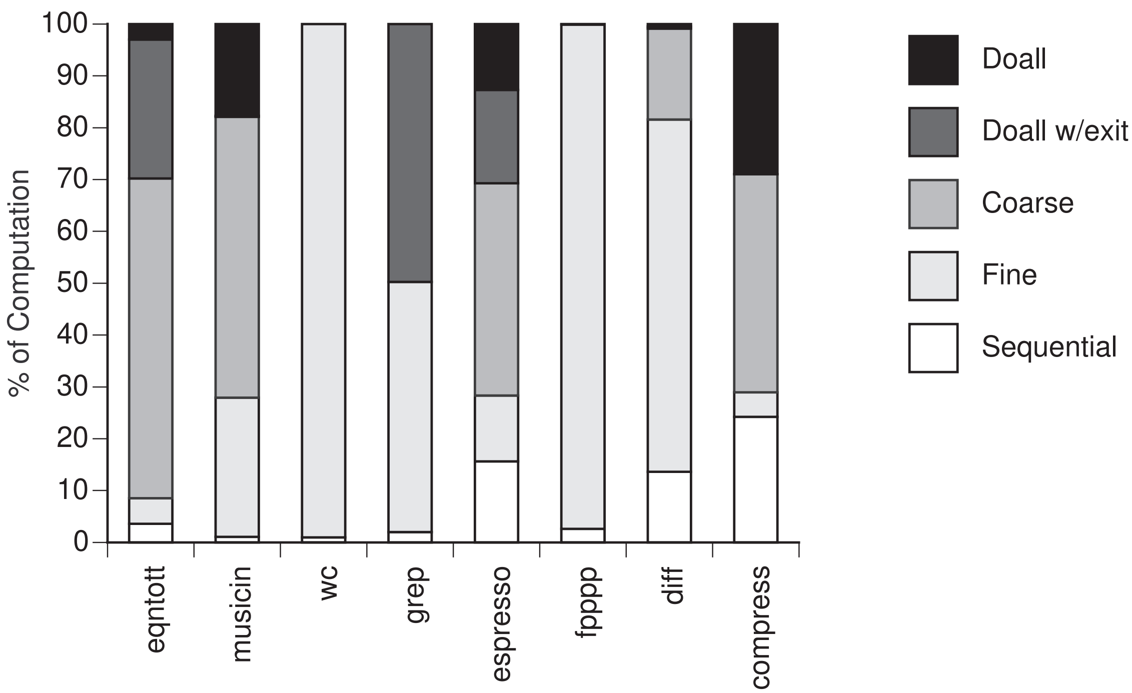

Figure 3.1 Composition of Speculative Execution Modes

The results show that the programs in our collection have very different behaviors. Overall, there is a significant amount of fine- and coarse-grain DoAcross parallelism in all the programs. Several programs also have DoAll parallelism.

The largest speedup, 16.5 times, found in eqntott is achieved by exploiting forms of parallelism that scale with the data set size: DoAll with early exits, and coarse-grain DoAcross parallelism. The medium speedups observed in music, wc and grep (from 6-10) are achieved in different ways: fine-grain and coarse-grain DoAcross parallelism, and DoAll loops with early exits. Espresso, which is one of the largest program in the collection, has a variety of different execution modes, each of which exhibits a reasonable amount of parallelism. Fpppp is a highly unusual program; it spends most if its time executing loops with three or less iterations, each of which consists of thousands of lines of straight-line code. The small number of iterations limit the speedup that can be achieved by exploiting loop-level parallelism. Different techniques such as partitioning the basic block across processors may be more effective for this program. Finally, the program diff and compress have many loop-carried dependences in the program, thus limiting the speedup achieved. We examine each mode of execution more carefully below.

Speculative DoAll loops. Out of the 8 programs examined, there are only three programs that have any measurable speculative DoAll parallelism. These results indicate that this collection of programs is very different from the scientific applications which are often dominated with DoAll parallelism. Only MusicIn, the one program in our collection that has a heavy numerical component, has a significant amount of parallelism in the DoAll loops. These results do not imply that a parallelizing compiler can find this amount of DoAll parallelism in the program. A compiler may be unable to find such parallelism automatically because the analysis is not powerful enough. Furthermore, our speculation machine ignores all anti-dependences, which is not possible with a conventional multiprocessor.

Speculative DoAll loops with early exits. Eqntott, grep and espresso has loops with this form of parallelism. If the dependencies in loops could be analyzed statically, a machine without speculation support could execute the loop as a DoAcross loop, where an iteration would not start until the last iteration determines that it will not terminate prematurely. This could be efficient if the termination condition can be calculated relatively early in the execution of the loop iteration. The use of speculative hardware obviates the dependence analysis by buffering up the side effects of the loop iteration until it is certain that the iteration is executed.

Coarse-grain speculative DoAcross parallelism. Coarse-grain parallelism is usually found in a small number of loops; however, these loops can constitute a large proportion of the program's execution time. Parallelizing outer loops as opposed to inner loops has the advantage that more operations can execute in parallel. On the other hand, in course-grain parallel programs instructions execute simultaneously that were originally much further apart in the sequential program, and thus there is a greater potential for data dependencies. The perfect speculation machine discovered significant effective coarse-grain parallelism in half of the programs. Furthermore, when the machine chooses to speculate on an outer loop, it usually achieves a reasonable speed up. The only exception is compress; in this case, even though the speedup on the outer loop is small, the speedup from speculating in the inner loops is even smaller. Overall, these results suggest that it is important to identify coarse-grain DoAcross loops in a program.

Fine-grain Speculative DoAcross parallelism. While the presence of coarse-grain DoAcross parallelism is more dependent on the algorithms used in a program, fine-grain DoAcross parallelism is more universal. Note that programs with coarse-grain speculative parallelism may also contain fine-grain parallelism that, is deemed to be less profitable. The range of the speedup is quite large, from 1.9 to 15.3. The success of exploiting this form of parallelism lies in a hardware organization that can support speculation and fine-grain communication very efficiently.

Sequential Execution. Without any parallelization overhead, the perfect speculation machine will employ speculation on loops even if the gain in performance is very small. As a result, the machine is speculating almost all the time. We observe that several programs (espresso, diff and compress) have a non-negligible sequential component. The reason for this is that the perfect speculation machine decides that it is more profitable to exploit speculative parallelism in the inner loops, which leaves the remaining code in the outer loops to execute sequentially. For example, espresso executes in speculation mode only 76% speculation, but manages to achieve a reasonable performance of 2.8 by speeding up the speculative portion by 4 times.

3.2.2 The Frequency of Roll Backs

The speculative execution hardware serves two important functions. First, it guarantees the serial semantics, so it is not necessary to prove the absence of a dependence at compile time. The second important function is that it can extract more parallelism out of loops with dynamically varying dependences than any non-speculative scheme. For example, in the program below:

```
for (i =0; i<n; i++) {
        f(a);
        ...
        if (exception_condition)
                a = a+1;
        }
```

The speculative execution hardware allows an iteration to proceed before the previous iteration determines if the `exception_condition` holds in the preceding iteration.

To understand how the speculative hardware is used Figure 3.2 further breaks down the computation into different components, highlighting whether the dependences are uniform across all iterations. DoAll loops with early exits obviously have irregular dependencies, requiring the hardware to squash those iterations that are speculatively initiated incorrectly. We also partition the coarse-grain and fine-grain speculative executions into those where the same data dependence shows up every iteration and those that do not. Clearly, most of the programs have dynamic dependences, with the exception of eqntott, and the ability of the hardware to roll back plays a significant role.

3.2.3 Size of the Speculative State

The perfect machine can keep an infinite amount of speculative state, whereas a real machine have a fixed limit to the amount of speculative state. In order to show the amount of speculative state required for each task, we keep track of the maximum volume of data written by a single iteration of any particular loop. Multiple writes to the same location during the same iteration are counted only once. in Table 3.3 shows the degree of parallelism obtained if a small amount of state (less than 300 bytes) is kept. In addition, if an infinite volume were possible, we give the maximum volume needed to execute any loops contributing to more than 0.5% of the parallel coverage. Note that in all but 2 of the programs, 300 bytes or speculative state is sufficient to obtain the desired parallelism. In fpppp, there is a threshold of about 4k bytes that are needed before any significant parallelism is attained. This correlates well to the size of the loop that is speculated across in fpppp. For musicin, a similar amount of state is required for maximum parallelism.

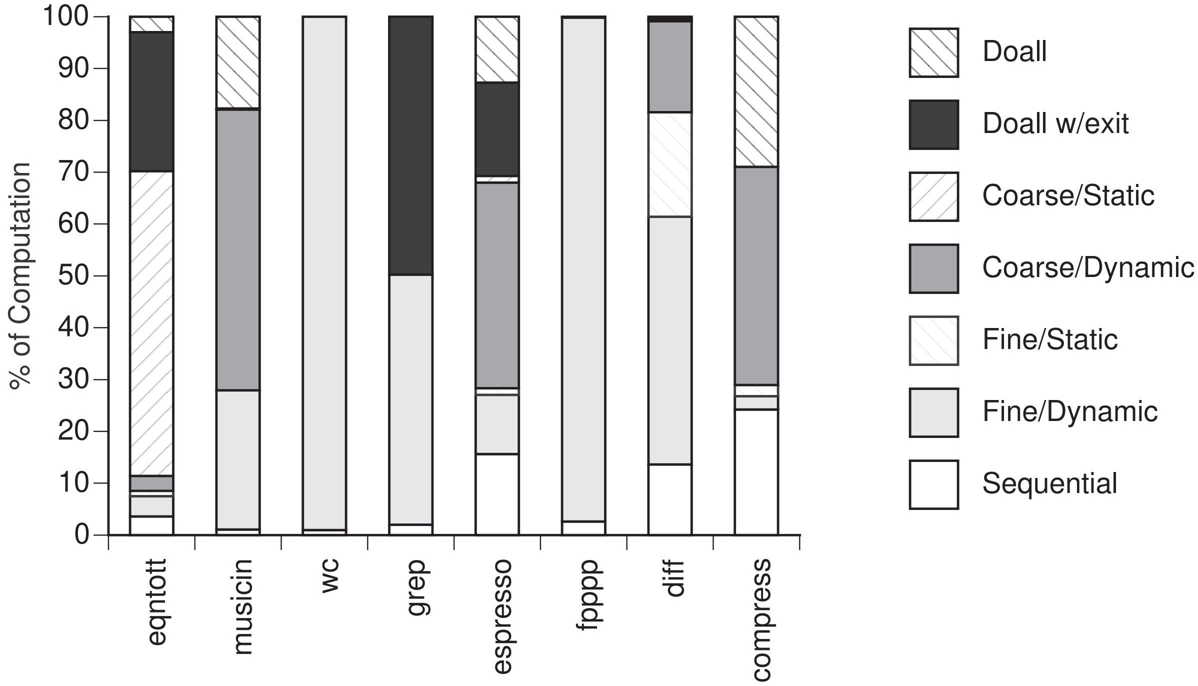

Figure 3.2 Speculative Parallelism with Static or Dynamic Dependencies

Table 3.3 Volume of Data vs. Parallelism Exposed

Program	< 300 bytes Parallelism	Infinite Volume Parallelism	Infinite Volume Bytes required
eqntott	16.5	16.5	97
musicin	7.1	9.5	4284
grep	6.0	6.0	44
espresso	2.5	2.8	1112
fpppp	1.0	2.5	3868
diff	2.1	2.1	160
compress	1.6	1.6	1036

4 Compiler Optimizations for Speculation

The abstract machine makes it easy to generate TLS code, but the that the resulting performance is highly sensitive to how the code is generated. We show how the compiler can improve the speculative code by using code reordering and by using statistics gathered from the perfect speculation machine to choose the best loops to execute speculatively and to insert synchronization code to reduce the frequency of roll backs.

4.1 Sensitivity of Performance to Code Ordering

The performance of a speculatively parallelized loop is very sensitive to the actual code sequence in each iteration. If, for example the first operation of every iteration depends on the last operation of the previous iteration, then no amount of cross-processor speculative hardware will deliver any parallelism on this code.

Loop-carried dependences impose limits on the parallelism available to the machine. Recurrences create loop-carried dependences that cannot be eliminated. A common form of recurrences found in programs is calculations on loop induction variables. We have implemented an aggressive induction variable recognition algorithm that eliminates these recurrences.

Loop carried dependences can also show up in code that does not participate in recurrences. We reduce these occurrences with a heuristic algorithm that greedily finds opportunities for converting them into intra-iteration dependences. By shifting the loop boundary for these loops, we can expose more parallelism to the speculative hardware.

4.2 Where to Speculate

In the perfect machine each individual execution of the same loop can decide whether or not to speculate for this execution. Thus, the overall optimal speedup of a loop may involve speculating on it some of the time, while instead allowing its inner loops to speculate at other times. However, when we annotate the program for real execution, for each loop we must make a static decision to speculate or not to speculate which will uniformly apply to all the dynamic executions of that loop. To guide the compiler, we have augmented our simulation of the perfect speculation machine to calculate for each loop its speed up had that loop been chosen execute speculatively for all executions. This information is compared with the speed ups of its inner loops to make the speculation decision.

We have modified our perfect speculation machine to accept static external decisions on which loops to speculate and evaluate the performance of the resulting execution. Table 4.1 shows the speculation results using the static decisions made using the heuristic described above. Most program speedups are largely unchanged, apart from musicin, which has a static loop nest in which the perfect machine chooses to speculate at different levels at different times, making a static speculation policy clearly suboptimal to the perfect dynamic one.

Table 4.1 Performance under Static Speculation

Program	Perfect Speedup	Static Speculation Speedup
eqntott	16.5	14.1
musicin	9.4	6.3
wc	7.8	7.8
grep	6.0	6.0
espresso	2.8	2.3
fpppp	2.5	2.5
diff	2.1	2.1
compress	1.6	1.6

4.3 Insertion of Synchronization Code

In a real speculative execution machine where backup costs are significant, it is worthwhile to introduce synchronizations into the program to delay a read operation if executing it early would only cause the machine to roll back and lose performance.

Figure 4.1 shows the frequencies of the cross-iteration dependences. The frequency is computed by dividing the number of times the dependence was observed by the number of iterations of the loop it was observed in, minus 1 (a loop with 1 iteration can have no cross-iteration dependences). Note that only the first of multiple reads from a single write is counted, as the first synchronization or rollback is presumed to eliminate the hazard for subsequent reads of the same data.

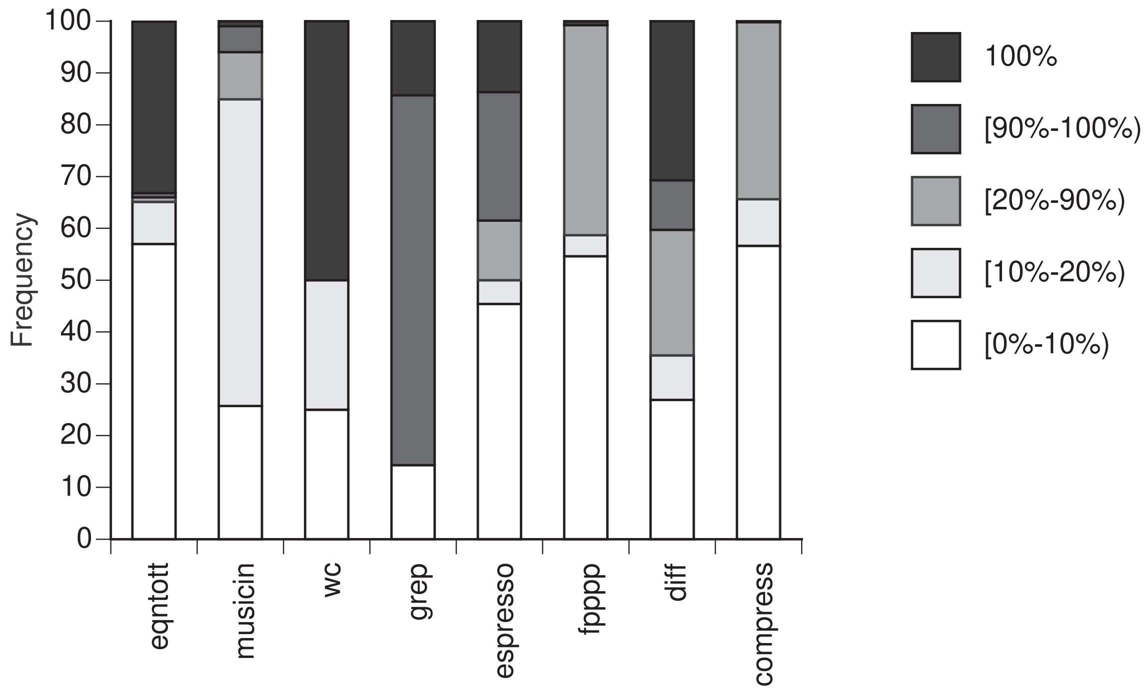

Figure 4.1 Frequency of dependences observed

The figure suggests a bimodal frequency distribution to the dependences with many being very close to 100% and many that rarely occur. Furthermore, examining the dependencies reveals that almost all loop-carried dependencies are detected between consecutive iterations. The only exception is compress, with 40% of the dependencies spanning the boundaries of 2 to 5 iterations. The results suggest a simple strategy of inserting synchronizations between operations whose dependences have a 100% frequency and synchronizing only between consecutive iterations.

Note that if we decide to synchronize a read or write that is inside a conditional control flow structure, we move the synchronization outside the control flow so that a consumer never waits for a producer synchronization that never occurs. Furthermore, we analyze the control flow to determine if enforcing the desired ordering of a pair of dependent operations may enforce the order of another; if so, we remove the redundant synchronization to improve the efficiency of the SPMD code.

4.4 Reducing Commits for DoAll loops

Commits operations must execute in order, and an iteration may not proceed until the preceding iteration has already committed. If the iterations have high variance in their execution times, serializing the commits may waste many cycles. It is possible to improve the performance of speculative DoAll loops by assigning multiple iterations to the same processor to minimize the overhead of the commit operations.

5 Hardware Support and Speculative System Performance

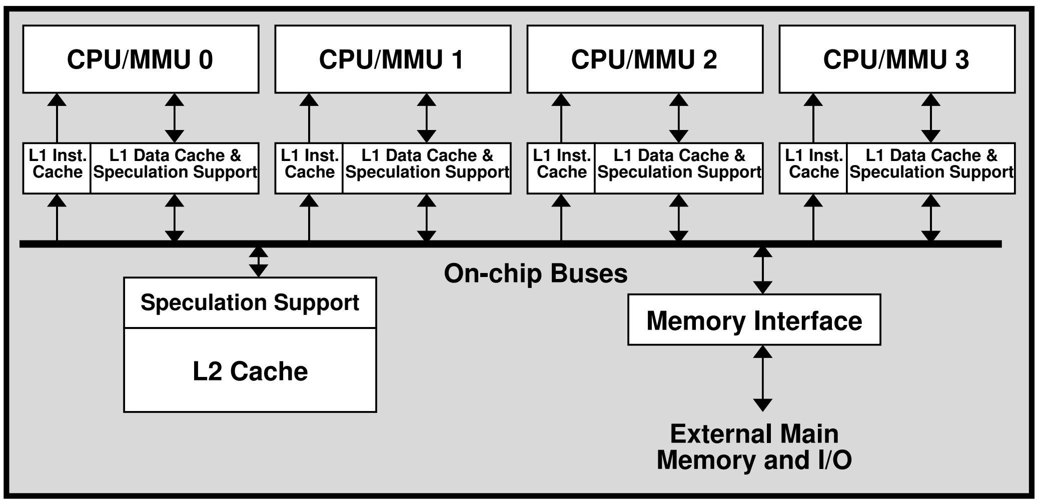

Figure 5.1 A single-chip multiprocessor with speculation support.

In this section we describe a high-performance single chip multiprocessor with hardware support for thread-level speculation. Figure 5.1 shows the architecture of a four CPU single-chip multiprocessor designed to exploit fine-grained speculative parallelism. At the heart of the system are four dual-issue processors. Each of the processors has a set of tightly coupled primary caches that makes it possible to achieve single-cycle cache access at very high processor clock frequencies. The on-chip memory system that interconnects the processor-primary cache subsystems provides interprocessor communication and supplies data from the lower levels of the memory hierarchy. It is designed for high-bandwidth with extensive pipelining and buffering. Key components of the memory system design are a pair of read and write buses. These buses provide the connection between the processors and to the rest of the levels of the memory hierarchy. In addition, the buses are the primary points of synchronization and communication between the processors in the system. To prevent the buses becoming a performance bottleneck they are designed for very low occupancy. Each transaction occupies a bus for a only one cycle; this is accomplished through the use of a pipelined centralized bus-arbitration mechanism that allocates cycles on the buses on every cycle, a cycle before the bus is actually used.

To maintain cache coherency, we use a write-invalidate protocol with write-through primary data caches. A write buffer is placed between the processor and the write bus to allow the processor to continue executing while the write waits for the bus. The write bus “sees” every write made by any of the processors because the data caches are write-through. The order in which the write bus is allocated to the processors imposes a total ordering on the writes in the system. A write is complete from the point of view of the other processors when the write transaction completes on the bus. The write bus is also the key coherency point and is responsible for invalidating the L1 caches that may be sharing the cache line. The write bus has a direct connection to the secondary cache, which must be updated on every write.

The shared secondary cache provides a low latency mechanism for the processors to share data. It is designed with enough bandwidth to accommodate the write-through traffic and the read-miss traffic from the four processors. The hardware support for speculation is distributed among the primary caches and the secondary cache. This hardware makes it possible to back out of memory references when a data hazard is detected. We describe the speculation support in detail in the next section.

5.1 Hardware Support for Speculation

Hardware support for speculation makes it possible for the processor to back-out of memory operations and restore the memory system to a previous state. This implies a way of keeping temporary memory values and a way of detecting when it is necessary to back out of memory operations. First, we describe an ideal model for hardware speculation and then we describe how the mechanisms for speculation are actually implemented.

Ideally, the memory system hardware support for speculation consists of a fully associative, infinite-size primary cache attached to each processor. This cache holds the speculative state and operates in a write-back mode so that no writes change the sequential state in the secondary cache until a commit point is reached. During speculative mode the behavior of the memory operations made by a processor executing task t_i are as follows. When a processor executing iteration t_i performs a read operation the speculative hardware returns the most recent value of the data. This data is read from the processor's primary cache if it is found there because this version of the data is the most recent. If it is not in the processor's cache then the most recent version of the data is read from a processor executing a task less than t_i. If the data is not found in any of the caches of processors executing a task less than t_i the data is read from the secondary cache. The fact that the processor has read and the memory address of the data is recorded in its primary cache.

It is important to note that because each processor has a separate speculative buffer the only data hazards that we detect are RAW hazards that are to due to true data dependencies in the program. Write after read (WAR) and write after write (WAW) hazards which are due to named dependencies are eliminated by the automatic renaming of memory locations provided by having a separate speculative buffer for each processor.

As Figure 5.2 shows, speculation support is distributed between the primary caches and a set of speculation write buffers. Associated with each processors primary cache are a set of bits for tracking RAW hazards and speculative state. There is one *read* bit per 32 bit word in the cache and two bits (*modified, pre-invalidate*) per line in the cache. The read bit is used to indicate that the word has been speculatively read by the processor. This bit is set when the processor reads the word into the cache. The word bit is used to detect RAW hazards when writes, which are broadcasting with the *taskID* of the processor that performed the write, appear on the write bus. The modified bit is used to indicate that the line contains speculative write state. It is set if the processor has written to the line or reads data that was written speculatively by another processor. The pre-invalidate bit is used as a pending invalidation bit. When a write appears on the write-bus associated with task *t*, it causes all matching cache lines for processors executing tasks less than t to set their pending invalidate bit. For processor executing tasks greater than *t* writes cause invalidations or violations. When a task commits all lines with the pending invalidate bit set are invalidated. Doing this ensures that when a processor that received the pending invalidate is reassigned to a task greater than *t* the state of its cache is consistent with the newly assigned iteration.

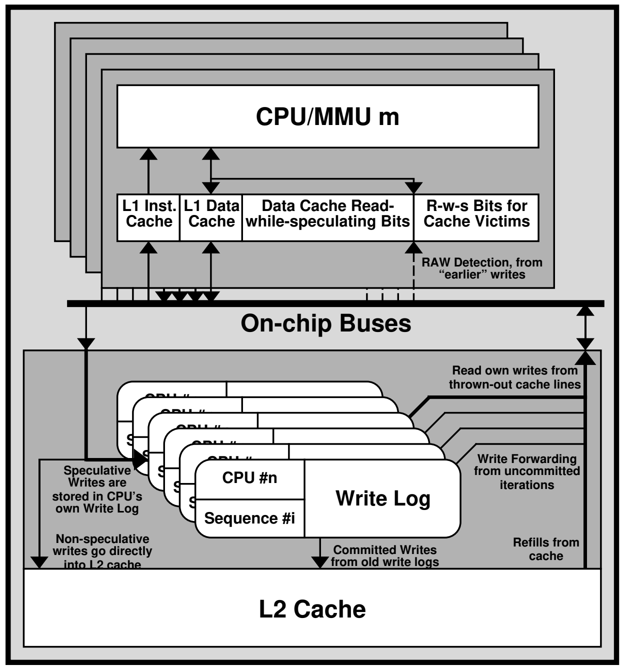

Figure 5.2 Implementation of speculation support.

With the ideal speculation support the primary caches are infinite and so there are no replacement misses. With finite sized primary caches there is the possibility of replacement misses. To maintain correctness, when a replacement happens, a processor executing iteration *i* must stall until all processors executing iterations less than *i* commit. At this point the processor can continue because there are no processors behind it in sequential program order and so it is no longer executing in speculative mode. To lessen the impact of replacement misses, when a line is replaced its speculative tag state is kept in a victim cache. When replacements in this victim cache occur the processor stalls.

A design goal for the support speculation is to minimize the time it takes to commit or discard speculative state and to ensure that the in-order state is updated correctly. A set of fully associative write buffers that hold speculative state are used to attain this goal. There are twice as many buffers as there are processors. During speculation mode a buffer is assigned to each processor. When a processor completes a commit operation the contents of the buffer are transferred to the secondary cache. To ensure a consistent view of the in-order state, the contents of the write buffers must be completely emptied one at a time. To allow a processor to work on a new iteration during the data transfer a new buffer is assigned to the processor. This double buffering minimizes the latency of the commit operation and maximizes the parallelism that is exploited in speculative mode. When a processor must discard speculative state the contents of the current write buffer are invalidated along with those lines in the primary cache that have the modified bit set. This should only take one or two processor cycles.

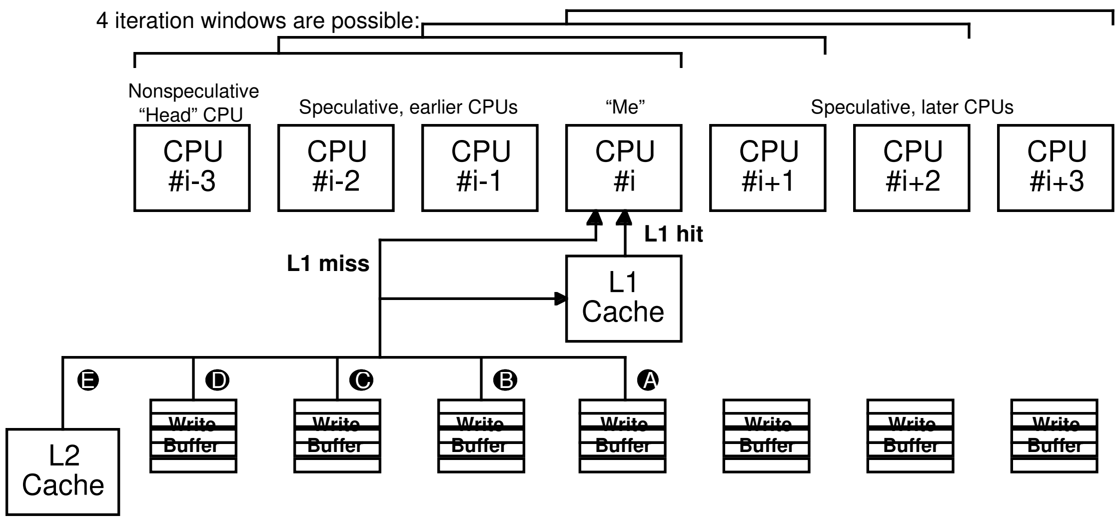

Figure 5.3 Speculative reads.

Figure 5.3 shows a diagram of how reads work in our multiprocessor with speculative support. The processor labeled *CPU #i* is shown in four possible iteration windows. In the first window *CPU #i* is the most speculative processor and in the last window it is the non-speculative (in-order) processor. Assuming that we are in the first iteration window speculative reads will work in the following way. A read hit fetches data from the primary cache and updates the appropriate speculative read bit. A read miss cause the line to be fetched from the secondary cache. The data from the secondary cache is merged with the data from the write buffers so that bytes from the buffer labeled *A* has the highest priority and bytes from the secondary cache labeled *E* have the lowest priority. This data merging ensures that *CPU #i* always sees the most recent data written by *CPU #i-3*, *CPU #i-2* and *CPU #i-1*.

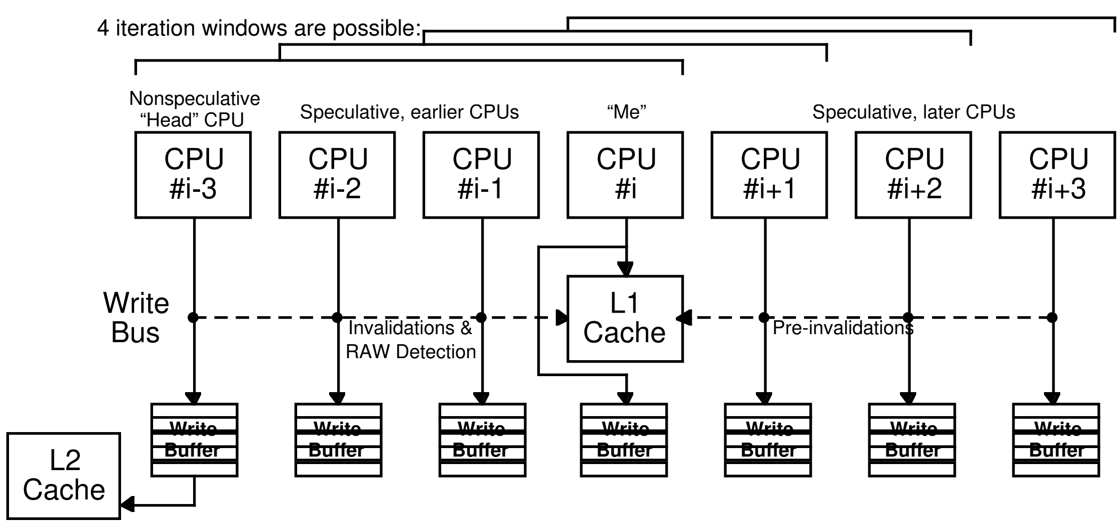

Figure 5.4 Speculative writes.

Figure 5.4 shows a diagram of how writes work. Again, four iteration windows are possible for *CPU #i*. Writes from processors (*CPU #i-3*, *CPU #i-2*, *CPU #i-1)* with taskIDs less than *CPU #i* cause the lines in *CPU #is* cache to be invalidated or may cause RAW hazard detection if *CPU #i* has already read the word that is being written. Writes from processors (*CPU #i+1*, *CPU #i+2*, *CPU #i+3)* with a taskID greater than *CPU #i* may cause pre-invalidations in *CPU #is* cache. The pre-invalidated lines will actually be invalidated when *CPU #i* commits.

5.2 Simulation Model

All of the results presented in this section are collected on a simulator that models the idealized hardware. We think that the single cycle commits and flushes will be possible to achieve in a real implementation, but of course infinite primary caches are not feasible. We will use the results presented later to show that the hardware requirements for a practical implementation of speculation support are actually quite modest.

The simulator for ideal speculation support is implemented in the SimOS environment[10]. SimOS models the hardware of a multiprocessor in enough detail to boot and run a commercial version of unix (SGI IRIX 5.3). This makes it possible for us to simulate binaries that have been compiled for existing SGI multiprocessor systems. SimOS has a set of CPU models with differing levels of detail and simulation speed. For this study we use a CPU model that models a simple five stage pipeline (IF, ID, EX, MEM, WB) [7], in which all instructions take a single cycle in the execute (EX) stage. This CPU model is coupled to a memory system model that includes ideal speculation support. In the memory system model all memory accesses take one cycle; this assumes a perfect infinite cache and is consistent with the description of ideal speculation support described in Section 5.1. We use a perfect memory system because we are primarily interested in an evaluation of the performance of speculation that is not influenced by the performance artifacts from other parts of the memory system design.

5.3 Speculation and the OS

The advantage of using the SimOS simulation methodology is that it provides a much more realistic environment in which to evaluate speculative parallelism. In this environment the CPU spends part of the time in kernel mode, handling system calls, I/O request interrupts and page faults. Therefore, the memory system sees all the memory references that would occur in a real machine. This presents extra challenges that are not present in a simulator that only directly executes user-level code. The challenges arise because the kernel should never be allowed to execute in speculation mode. There are two reasons for this. First, the kernel has not been analyzed by our compiler so there is no guarantee that it will work correctly in speculation mode. Second, the kernel interfaces with I/O devices such as the disk subsystem that have no facility for backing up.

To control speculation in the kernel there are two broad classes of interrupts we need to consider. Those that are coerced such as page faults, timer interrupts and disk I/O requests and those that are user requested such as a *syscall* made by the user program. Coerced interrupts may or may not be related to the user process and must be responded to by the processor immediately to avoid a potential dead-lock situation. In speculation mode these interrupts are handled by turning off speculation so that reads and writes bypass the speculation support and operate directly on sequential state. Any writes that occur while the processor is executing in the kernel will be checked for RAW hazards against all the processors. If a hazard is detected, the processor is restarted. However, in our simulations we have never observed this situation. User requested interrupts such as a syscall do not have to be handled immediately, but these interrupts must occur in the same order as the sequential program and must be serialized. To enforce these conditions the processor must execute a *commit* operation before executing the syscall. This ensures that the processor is in a sequential state before the kernel is entered and that no speculative syscalls are executed.

5.4 Speculation System Performance Results

To demonstrate the software and hardware for speculation working together in a complete system we present some preliminary results. We show the performance of our hardware model running four applications, namely: *wc*, *eqntott*, *grep* and *diff* that have been compiled automatically using our feedback-based parallelizing compiler. Table 4.1 shows the speedup and the percentage of speculative tasks that must restart for the four applications. The speedup achieved by the speculative multiprocessor on the applications is highly correlated to the number of restarts. W*c* achieves a very good speedup of 3.6 on four processors. This is consistent with the single loop with fine-grain speculative do-across parallelism which can be exploited with speculative hardware support. The percentage of speculative tasks that are restarted is low and the restarts happen at the beginning of the execution, after which the processors operate in a pipelined fashion without interfering with each other. The performance of *eqntott* is not as good as *wc* even though perfect speculation machine results predict that *eqntott* should achieve better performance than *wc*. The reason for this is that large fraction of the tasks in the speculative do-across parallelism in *eqntott* are restarted. These restarts waste time and lower overall performance. Recall that the perfect speculation machine does not need to restart tasks. *Grep* contains both do-all with early exit parallelism and fine-grain do-across parallelism. The results indicate that most of the parallel speedup is due to the do-all parallelism; the application spends most of the time in the do-across loop restarting. The result is that a high percentage of the speculative tasks are restarted. The restarts are due to a loop-carried dependencies between the loop iterations that result in a RAW hazards. The performance of *diff* is not very good because all the speculative parallelism is in fine-grained do-across loops that. These loops restart constantly which indicates that there is not much true parallelism in the application.

Program	Speedup	% Restart
wc	3.6	9.4
eqntott	2.7	40.7
grep	2.2	53.7
diff	1.2	85.5

Table 5.1 Overall hardware performance on four processors

An important result that is shown in Figure Figure 5.4 is the size of the speculative write state for the four applications. The results indicate that the write state is relatively small; under 100 bytes for 99% of the speculative tasks. We found that the speculative read state was also under 100 bytes for the majority of speculative tasks. The implication of these results is that replacements in primary cache of reasonable size (greater than 8 Kbytes) are unlikely to cause significant performance losses in speculative mode. More importantly, the small size of the write state indicates that the speculative state can be placed in write buffers constructed from a modest amount of SRAM. These results closely correlate with the size of the speculative read and write state measured with the perfect machine described in Section 3.1

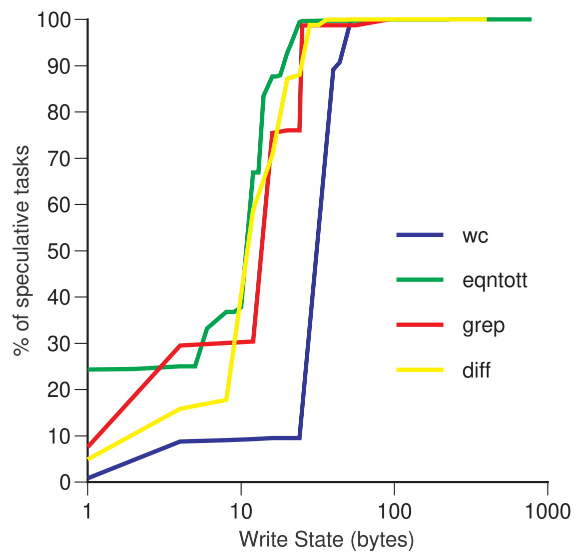

Figure 5.5 Cumulative distribution of speculative write state.

6 Related Work

The goal of exploiting the maximum amount of parallelism in general purpose programs is the driving force of modern processor design. The use of multiple instruction issue, speculative execution and dynamic scheduling to exploit instruction level parallelism (ILP) has been extensively studied. Rau and Fisher [9] present a comprehensive overview of these approaches and Wall studies suggest that the ILP limits with reasonable window size are not very high [12]. To overcome the limitations of conventional superscalar approaches to ILP, Franklin and Sohi have proposed the expandable split window approach [5] which has been further refined in the Multiscalar paradigm [11]. The multiscalar paradigm combines multiple threads and speculative execution. The developers of this approach present several arguments for why this paradigm will be better than superscalar approaches at exploiting parallelism in general purpose programs. Sohi et. al. also argue that speculation will enable the multiscalar paradigm to exploit parallelism that is not available to a multiprocessor. There have been some attempts to exploit do-across parallelism on a multiprocessor [4]. These attempts have not been successful due to the high cost of synchronization and communication latency in commercial available multiprocessors. Our approach to speculation with multiple threads combines the ideas from the Multiscalar paradigm with much more emphasis on compiler technology and a focus on hardware mechanisms that lead to efficient VLSI implementation. This makes it possible to use speculative parallelism to improve performance with the addition of a modest amount of hardware support to a single chip multiprocessor architecture.

7 Conclusions

This paper describes the first detailed analysis on a speculative computation model that uses multiple flows of control. The results show that speculative execution is a promising technique to the very difficult and important problem of improving the performance of general-purpose code. In particular, the results point out that exploiting speculative parallelism in outer loops can lead to a significant performance gain in these codes.

This paper also presents an automatic strategy of generating efficient speculative software. While the speculative hardware makes generating legal code easy, generating efficient code is difficult. We have developed an algorithm to reduce loop-carried data dependences in a program. We have developed a simulation and feedback system to determine the best loops to speculate and the synchronizations to introduce. Our analysis shows that having to make static decisions on where to speculate only degrades the performance slightly over ideal decisions that can vary the choice every instance. Furthermore, we show that the frequencies of the dependences are bimodal, allowing a simple and highly effective strategy of placing explicit synchronizations on those dependences with close to 100% frequency.

We have proposed hardware support for speculative parallelism. We investigated both ideal hardware support and a realistic implementation the used a primary cache and speculative write buffers. We incorporated a model of the ideal version of the speculation support in to a simulation environment which closely resembles a real machine. This realism raised a number of issues, which we addressed, concerning the interaction of speculation with the operating system. Our results demonstrate the complete software and hardware system for speculation in multiprocessors. So far, we have achieved good speedups on small integer programs. The measurements of the speculative read and write state suggest that the small speculative write buffers can take complete advantage of speculative parallelism in integer programs.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng, "An overview of the SUIF compiler for scalable parallel machines," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Compiler*, San Francisco, 1995.

[2] S. Amarasinghe et.al., "Hot compilers for future hot chips," *presented at Hot Chips VII*, Stanford, CA, 1995.

[3] W. Blume, et. al., "Parallel programming with Polaris," *Computer,* vol. 29, pp. 78–83, December, 1996.

[4] R. Cytron, "Doacross: beyond vectorization for multiprocessors," *Proceedings of Int. Conf. Parallel Processing*, pp. 836–844, 1986.

[5] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 58-67, Gold Coast, Australia, May,1992.

[6] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, pp. 84–88, December, 1996.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach 2nd Edition*. San Francisco, California: Morgan Kaufman Publishers, Inc., 1996.

[8] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proceedings of 19th Annual International Symposium on Computer Architecture*, pp. 46-57, Gold Coast, Australia, May, 1992.

[9] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, 1993.

[10] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "The SimOS approach," *IEEE Parallel and Distributed Technology*, vol. 4, no. 3, 1995.

[11] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proceedings of 22nd Annual Intl. Symp. Computer Architecture*, pp. 414–425, Ligure, Italy, June 1995.

[12] D. W. Wall, "Limits of Instruction-Level Parallelism," Digital Western Research Laboratory, WRL Research Report 93/6, November 1993.

[13] R. Wilson and M. Lam, "Efficient context-sensitive pointer analysis for C programs," *Proceedings of Prog. Lang. Design and Implementation*, pp. 1–12, June 1995.