Limits of Control Flow on Parallelism

Monica S. Lam and Robert P. Wilson
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper discusses three techniques useful in relaxing the constraints imposed by control flow on parallelism: control dependence analysis, executing multiple flows of control simultaneously, and speculative execution. We evaluate these techniques by using trace simulations to find the limits of parallelism for machines that employ different combinations of these techniques. We have three major results. First, local regions of code have limited parallelism, and control dependence analysis is useful in extracting global parallelism from different parts of a program. Second, a superscalar processor is fundamentally limited because it cannot execute independent regions of code concurrently. Higher performance can be obtained with machines, such as multiprocessors and dataflow machines, that can simultaneously follow multiple flows of control. Finally, without speculative execution to allow instructions to execute before their control dependences are resolved, only modest amounts of parallelism can be obtained for programs with complex control flow.

1 Introduction

The potential for parallelism can be evaluated from two different perspectives. The first is to propose an actual system design; the performance of such a system is a lower bound on the amount of available parallelism. The second is to perform studies on the limits of parallelism for a particular approach by enforcing only the constraints associated with the approach and relaxing all other constraints. If the upper bound of performance is found to be unacceptably low, then the approach is inadequate. On the other hand, if the upper bound is within expectations, the only conclusion is that the approach may be sufficient. It is only from low limits that we can draw interesting conclusions.

A recent study by Wall [17] contains a surprising result that suggests a severe limitation in current approaches. One of the experiments in that study examines the performance of a processor that uses an aggressive hardware algorithm to predict the outcomes of branches. Instructions along the predicted path of a branch are *speculatively* executed before the branch is resolved. The processor has perfect memory disambiguation, perfect register renaming, unlimited instruction fetching, and a very large number of functional units. The reported speedups for this processor on a set of non-numeric programs range from 4.1 to 7.4. Given the assumptions of perfect memory disambiguation and a large number of functional units, these speedups are quite small.

Wall's result contrasts sharply with experiments that assume perfect branch prediction [10, 12]. Since a machine with perfect branch prediction requires foreknowledge, we refer to such a machine as an *oracle*. The effects of control flow on parallelism are essentially eliminated on an oracle machine because all of the branch outcomes are known in advance. Much more parallelism is available on an oracle machine, suggesting that the bottleneck in Wall's experiment is due to control flow.

The ultimate goal of our study is to discover ways to increase parallelism by an order of magnitude beyond current approaches. In this paper, we focus on control flow. Through a study of the limits of parallelism, we hope to establish the inadequacy of current approaches in handling control flow and identify promising new directions.

This paper offers evidence that the limit of parallelism discussed in Wall's paper is likely to apply to many more non-numeric programs than those measured. A processor that uses speculation to only exploit local parallelism found between mispredicted branches is fundamentally limited. A compiler can locate more global parallelism in the code by extracting the true *control dependence* of the program. However, this increased opportunity for parallelization cannot be fully exploited unless the machine can simultaneously pursue multiple flows of control. The degree of parallelism is fundamentally limited by the von Neumann architecture. Higher performance can be obtained with machines that can simultaneously follow multiple flows of control.

In this paper, we analyze the techniques of speculative execution, control dependence analysis, and following multiple flows of control. We evaluate these techniques empirically by computing the limits of parallelism for a set of programs on machines that employ different combinations of these techniques. So far, research on superscalar and multiproces-

This research was supported in part by DARPA contract N00039-91-C-0138 and by an NSF graduate fellowship.

In *Proceedings of the 19th Annual International Symposium on Computer Architecture* Gold Coast, Australia, May 19–21, 1992, pp. 46–57.

sor architectures has mostly been conducted independently. This work studies how instruction and processor levels of parallelism interact. Our study also suggests that a useful characteristic for predicting the parallelism in a program is whether the control flow is data dependent.

We first introduce the major concepts of the paper in Section 2. We explain speculative execution, control dependence analysis, and pursuing multiple flows of control. Section 3 presents a set of abstract machine models that use these three techniques in various combinations. In Section 4, we describe the experimental framework used to evaluate the limits of parallelism. Section 5 presents the results of these experiments and analyzes the parallelism for each machine model. Finally, Section 6 presents the conclusions of our study and describes its implications for some specific architectures that have been proposed and implemented.

2 Relaxing Control Flow Constraints

The perfect branch prediction of an oracle machine is an upper bound of all other techniques to relax control flow constraints. Unfortunately, it is also an unrealistic upper bound. Because branch outcomes are known in advance on an oracle machine, no instructions have to wait for branches to be resolved. To achieve the same performance, a machine must speculatively execute all possible paths through a program. This requires hardware resources exponential in the number of outstanding conditional branches. Since perfect branch prediction is not realistic, more practical techniques are required to handle control flow. This section examines three techniques: speculation with branch prediction, control dependence analysis, and following multiple flows of control.

2.1 Speculation with Branch Prediction

While speculation on all possible paths is infeasible in practice, limited speculation is commonly used throughout computer systems to improve performance. For example, almost all modern processors prefetch the instructions following a conditional branch. Some machines such as the IBM 360/91 pursue both execution paths by also prefetching the instructions at the branch target.

A common technique to improve the efficiency of speculation is to only speculate on instructions from the most likely execution path. The success of this approach depends on the accuracy of branch prediction. Even for non-numeric code, both hardware and software prediction algorithms have been shown to be accurate over 85% of the time [6, 8].

Extending speculative instruction fetching to speculative execution creates additional parallelism. However, unlike instruction fetching, speculatively executing an instruction may generate unwanted side effects. These side effects must be discarded if the branch prediction is incorrect. Both hardware and software techniques can be used to implement speculative execution.

Various hardware structures have been proposed to support speculative execution [7, 11, 13, 16]. These structures store the results of the speculative instructions until the branch direction is determined. If the branch prediction was correct the results are committed, otherwise they are discarded. Hardware scheduling, however, is limited by the fact that an instruction simply cannot execute before it is fetched. It is difficult to fetch instructions from far ahead along the predicted execution path, especially for programs with complex control flow.

Software techniques overcome instruction fetch limitations by reordering instructions. Trace scheduling identifies the most commonly executed trace and schedules the instructions within the trace as if they belong to one large basic block [2, 4]. However, implemented trace scheduling algorithms only employ very limited forms of speculation. Smith et al. extend software scheduling with hardware support for speculative execution [15]. Instructions to be speculatively executed are *boosted* before a conditional branch. These instructions are labeled so that their results are discarded or committed when the branch condition is determined. This combines the ability of software to eliminate fetch limitations and the ability of hardware to speculatively execute instructions with side effects.

2.2 Control Dependence Analysis

For most hardware instruction schedulers, all speculatively executed instructions must be discarded when a conditional branch is mispredicted. This constraint is, however, unnecessarily strict. Consider the following simple example:

```
if (a < 0)
    b = 1;
c = 2;
```

While the assignment `b = 1` is executed only if `a < 0`, the assignment `c = 2` is always executed regardless of the value of `a`. We say that `b = 1` is *control dependent* on the condition `a < 0` and that `c = 2` is *control independent*. We refer to the branch on which an instruction is control dependent as its *control dependence branch*.

A hardware scheduler generally cannot determine which instructions are control independent. In this example, suppose the scheduler speculatively executes `c = 2` before the condition is resolved. If the branch is mispredicted, the hardware must discard the assignment to `c` only so that it can repeat the identical assignment later. A compiler can compute the control dependence and eliminate this inefficiency. More generally, control dependence analysis allows instructions to be moved across many branches. By expanding the range of code from which parallelism can be extracted, control dependence analysis increases the available parallelism.

Control dependence is an intuitively simple concept. For block-structured programs, the immediate control dependence for an instruction is simply the condition in the closest enclosing control construct. For example, in the following code,

```
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```

the call of the `foo` function is control dependent on the condition `A[i] > 0`, which in turn is control dependent on the loop exit condition `i < 100`. The `bar` function is not control dependent on any part of the loop and can execute in parallel with the loop, assuming there is no data dependence between `foo` and `bar`.

Control dependences in programs with arbitrary control flow can easily be computed in a compiler using the *reverse dominance frontier* algorithm [3]. Hardware techniques for analyzing control dependences have also been considered [9], but they can only detect a small subset of the control independent instructions and require complex hardware.

2.3 Executing Multiple Flows of Control

Control dependence analysis discovers parallelism from across different regions of code, each of which may have its own flow of control. In the example above, control dependence analysis shows that the `bar` function can run concurrently with the preceding loop. However, a uniprocessor can typically only follow one flow of control at any time. A uniprocessor cannot fetch and execute the instructions from within the loop while following the arbitrary control flow that may be present in the `bar` function.

Support for following multiple flows of control is necessary to fully exploit the parallelism uncovered by control dependence analysis. Multiprocessor architectures are a general means of providing this support. Each processor of a MIMD multiprocessor can follow an independent flow of control. At the other extreme, it is sometimes possible to generate uniprocessor code that corresponds to pursuing multiple flows of control in the original computation. A small number of code segments can be packed together by generating different versions of the code for every possible combination of control flow. However, in general, combining independent control flows can lead to significant code expansion.

3 Abstract Machine Models

To establish the fundamental limits of these three techniques for relaxing control flow constraints, we define a set of abstract machine models and analyze the parallelism for each machine under ideal conditions. Each machine model uses a different combination of the three techniques. Our approach is to examine a set of instruction traces from real programs

BASE An instruction cannot execute until the immediately preceding branch in the trace is resolved. This implies that branch instructions must execute in order, one per cycle.

CD An instruction cannot execute until its control dependence branches are resolved. In addition, branch instructions must execute in order, one per cycle, to reflect the inability to pursue multiple flows of control simultaneously.

CD-MF An instruction cannot execute until its control dependence branches are resolved. Multiple branch instructions can execute in parallel and need not be ordered.

SP An instruction cannot execute until the immediately preceding mispredicted branch in the trace is resolved. This implies that a branch instruction must wait for all preceding mispredicted branches.

SP-CD An instruction cannot execute until its mispredicted control dependence branches are resolved. In addition, a branch instruction must wait for all preceding mispredicted branches, not just the ones it is control dependent on.

SP-CD-MF An instruction cannot execute until its mispredicted control dependence branches are resolved. There are no additional constraints on branches.

ORACLE There are no constraints due to control flow.

Figure 1: Control Flow Constraints of Abstract Machines

and compute the available parallelism by simply enforcing true data dependence constraints and the control flow constraints associated with each abstract machine model. Other constraints due to imperfect memory disambiguation, reuse of variables, and limited resources are ignored.

We define seven abstract machines. The BASE machine uses none of the three techniques and provides a baseline for comparison. The ORACLE machine has perfect branch prediction, and its performance represents an upper bound of parallelism given the assumptions in our experiments. The other five machines use combinations of control dependence analysis (CD), following multiple flows of control (MF), and speculative execution with branch prediction (SP). There are only five interesting combinations because we cannot recognize independent flows of control without control dependence analysis.

Each machine is distinguished by its ability to handle control flow. We model these abilities by imposing the sequencing constraints shown in Figure 1 on the execution of instructions in dynamic traces. More aggressive machines have less restrictive constraints. Since the BASE machine uses none of the three techniques, its control flow constraint is the most severe; it prevents instructions from executing before any preceding branches. The CD machine has perfect control dependence information, and thus, instructions that are not dependent upon a branch need not wait for it to be resolved. To reflect the limitation of following one flow of control, we also impose an ordering on the branch instructions. To model the behavior of current compilers [1], this ordering requires all branches in the program to execute in the original sequential execution order. The CD-MF machine may follow an unlimited number of flows of control and does not require a branch ordering constraint.

All of the machines with speculative execution in this study only speculatively execute instructions on the most likely execution path. Simultaneously executing instructions on alternate paths would require that some instructions be cancelled regardless of the branch outcomes. However, in the various SP machines, all of the speculative instructions are potentially useful, making these machines more realistic than the ORACLE machine.

The SP machine can speculate on an infinite number of consecutive branch outcomes. This essentially creates an infinitely long path of predicted instructions through a program. Instructions from anywhere along this path may execute in parallel. However, when a branch is mispredicted, all of the instructions on the predicted path following the branch must be cancelled. Only those instructions that are not cancelled appear in the traces that we analyze. Therefore, the control flow constraint on an instruction in a trace is that it cannot execute until all of the preceding mispredicted branches are resolved. As long as the branch predictions are correct, the flow of control does not change and the branch instructions can execute in any order. A mispredicted branch requires that the flow of control transfer to the unpredicted branch path, and thus, only one mispredicted branch can execute in each cycle.

The SP-CD machine differs from the SP machine in its treatment of mispredicted branches. Instead of cancelling all of the instructions along the predicted path, only those instructions that are actually control dependent on the mispredicted branch are cancelled. Due to following a single flow of control, the mispredicted branches must still be executed in order. The SP-CD-MF machine can follow multiple flows of control simultaneously, and thus mispredicted branches can execute in parallel.

To illustrate the power of the different abstract machine models, consider the flow graph in Figure 2. Assume that there are no data dependences in this program. Each node in the graph consists of a single instruction. Next to each node is the set of branch instructions on which it is immediately control dependent. The edges represent the possible flow of control, with the more likely branch outcomes highlighted by bold arcs. One possible trace through this graph is also

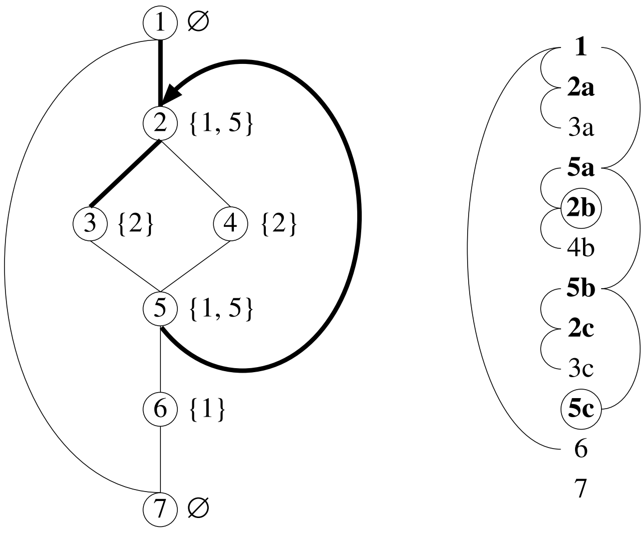

Figure 2: Example Flow Graph and Trace

shown in Figure 2. Each instruction in the trace is identified by the instruction node number and a letter to distinguish the specific instance. Branch instructions are written in boldface and circled if they are mispredicted. The edges in the trace represent the control dependence relationships.

Since there are no data dependences in the program, the ORACLE machine simply executes all of the instructions in one cycle. The executions of the other machine models are shown in Figure 3, where the edges represent the dependences due to control flow and instructions at the same level execute at the same time.

The BASE machine executes all of the branches sequentially and each non-branch instruction one cycle after the preceding branch. The CD machine executes instructions `6` and `7` earlier because they are not control dependent on the immediately preceding branches. The CD-MF machine need not execute the branches in order; the edges are simply the control dependence edges from the trace in Figure 2. The SP machine executes all of the instructions between mispredicted branches in parallel. The SP-CD machine behaves similarly but executes instructions `3c`, `6`, and `7` earlier because they are not control dependent on the mispredicted branches. Finally, the SP-CD-MF machine executes all but one of the instructions in one cycle. Since instruction `4b` is not on the predicted path, the machine does not speculatively execute the instruction. The instruction is therefore not executed until the processor discovers that it has mispredicted the branch `2b`. In contrast, to achieve the performance of the ORACLE machine, both instructions `3` and `4` must be executed in parallel on every iteration. This illustrates the fundamental difference between SP-CD-MF and ORACLE: the SP-CD-MF machine does not pursue alternate paths simultaneously.

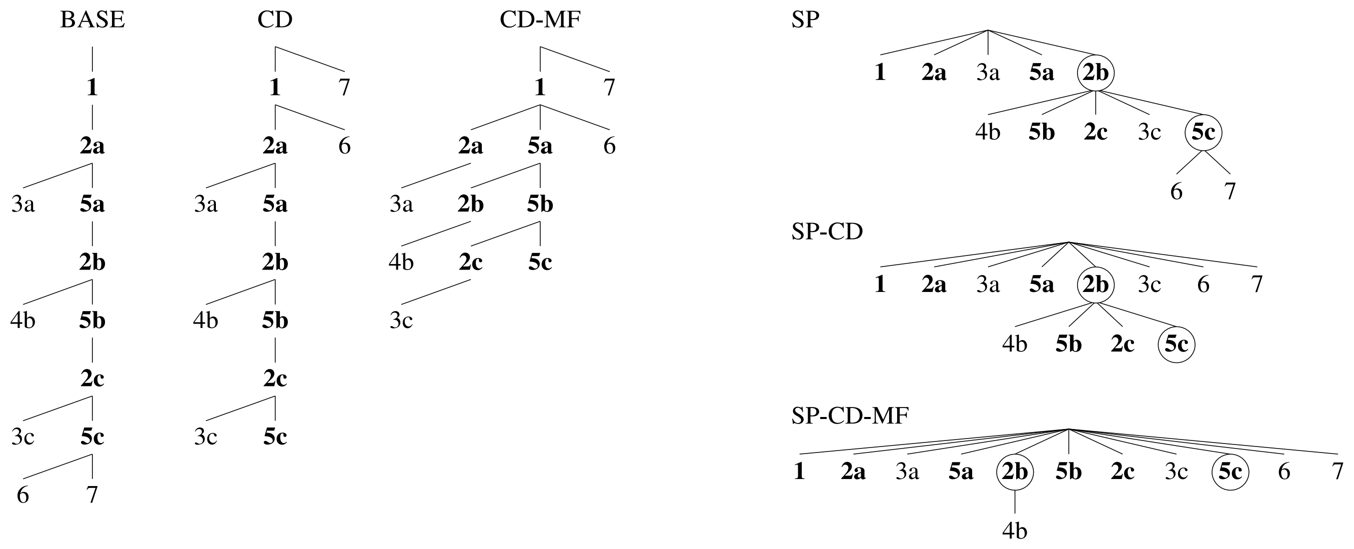

Figure 3: Execution of the Abstract Machines

4 Experimental Framework

This section describes the methodology used to analyze the parallelism for each machine model. Since we are only interested in evaluating techniques for handling control flow, we limit the scope of the experiment to include only the fundamental constraints related to control flow.

4.1 Data Dependence

We assume an idealistic approach to handling data dependences. The only data dependence constraint enforced is that a read operation in the trace must only wait for the immediately preceding write operation to the same location. This relaxes many constraints in reality.

First, since we are enforcing only true data dependences (reads after writes), we have eliminated all the anti-dependences (writes after reads) and output dependences (writes after writes) for all register as well as memory accesses. In practice, various renaming techniques can be used to remove some of these spurious data dependences.

Second, we also assume that memory references are perfectly disambiguated; that is, the machines can determine if two addresses are identical even before they have been computed. This perfect disambiguation is an upper bound for what can be achieved by compiler analysis and by speculative hardware disambiguation.

Third, an instruction must wait until it can be determined that no preceding instructions will alter the instruction's operands. Different data dependences may occur along different control flow paths, as shown in the following example:

```
b = 0;
if (a < 0)
   b = 1;
c = b;
```

Depending on the outcome of the `a < 0` condition, the `b = 1` assignment may or may not be executed. Therefore, the value assigned to `c` depends on the outcome of the condition. The potential data dependence between `c = b` and `b = 1` delays the assignment to `c` at least until the condition `a < 0` is resolved.

Since potential data dependence constraints force instructions to wait for branches, speculation with branch prediction can relax these constraints. For example, if we predict that `b = 1` will not execute, `c = b` can execute speculatively with the assumption that `b` is `0`. If the prediction was incorrect, the assignment must be re-executed with the correct value of `b`.

We are unable to include the potential data dependence constraints in our experiment because only one execution path is captured in a trace. These constraints must be analyzed statically. Static analysis, however, cannot be completely accurate in the presence of indirect memory references because perfect disambiguation is only possible when analyzing a trace. To ensure that our results are upper bounds, we must avoid introducing constraints based on imperfect disambiguation, and so we ignore the potential data dependence constraints.

4.2 Program Transformations

Procedure calls and loops are two control constructs that are commonly implemented in ways that introduce unnecessarily serializing constraints. These constraints can be eliminated by alternate implementation techniques and simple transformations such as procedure inlining and loop unrolling. Our study tries to model an upper bound of these techniques.

Procedure calls and returns introduce unnecessary control flow and cause problems for branch prediction. Furthermore, a new stack frame is typically allocated and deallocated in every procedure by incrementing and decrementing the stack pointer. Since there is a true data dependence between every increment and decrement, these stack pointer manipulations must be executed sequentially. For programs with many small procedures, this may limit the parallelism. To simulate the optimal case of inlining all procedures, including recursive procedures, we ignore all call and return instructions in a trace, as well as all instructions that manipulate the position of the stack pointer.

Parallelism in many loops is inhibited by dependences on the loop index variables. Loop index variables and induction variables are incremented in every loop iteration[1], creating true data dependences between iterations. However, these dependences are only a result of the way code is generated for scalar processors and are not an inherent part of the loop semantics. Loop unrolling is a simple technique that compilers use to reduce such effects.

In our experiment, we simulate perfect and complete unrolling. First, we analyze the object code to discover the loops in the program. We then use iterative data flow analysis to identify registers that are incremented by a constant exactly once per loop iteration. We only check the registers and not memory locations because we assume that the compiler has allocated the index variables in registers. If the index variables could not be register allocated, the loop index update is unlikely to be in the critical path of the loop execution. Finally, the analysis marks all instructions that increment loop index and induction variables, comparisons of loop indices with loop invariant values, and branches based on the results of such comparisons. These instructions are ignored when they occur in the trace.

Admittedly, many more transformations could be applied. A sophisticated compiler may be able to translate a program into an equivalent one that is more amenable to parallelization. This experiment does not, by any means, establish the limit of parallelism for the problem solved by a program. Finding such a limit is an undecidable problem. This experiment only establishes the limits of parallelism for the particular code generated by the compiler and subject to all of the assumptions in our experiment. If the resulting limits are considered inadequate, other techniques not considered in this study must be used.

4.3 Benchmark Programs

Our benchmark suite consists of the six SPEC programs and the four other benchmarks shown in Table 1. We have included three of the FORTRAN SPEC programs for comparison with the non-numeric programs. The standard inputs were used for the SPEC benchmarks. The programs were compiled for a MIPS R3000 processor by the MIPS C and FORTRAN compilers with full optimization. The traces were obtained using the MIPS pixie tool, and each program was simulated for up to 100 million instructions. Our instruction traces include library routines but not system calls, so we ignore dependences that occur within the operating system.

Program	Language	Description
awk	C	pattern scanning
ccom	C	C compiler front-end
eqntott	C	truth table generation
espresso	C	logic minimization
gcc (cc1)	C	Gnu C compiler
irsim	C	VLSI layout simulator
latex	C	document preparation
matrix300	FORTRAN	matrix multiplication
spice2g6	FORTRAN	circuit simulation
tomcatv	FORTRAN	mesh generation

Table 1: Benchmark Programs

4.4 Simulation Algorithm

The basic simulation algorithm is to determine the execution time of each instruction in a trace. The completion time of the last instruction to execute is the total execution time for the trace. The resulting parallelism is the ratio of the sequential execution time to the parallel execution time. Since we want to measure the actual parallelism and not the speedup for a realistic machine, we use one clock cycle latencies for all instructions. With non-unit latencies, some of the parallelism would be used to fill pipeline bubbles. The instructions removed because of our perfect inlining and perfect unrolling do not contribute to the sequential time, so the parallelism does not include any speedup due to removing those instructions.

Our experiment allows instructions arbitrarily far apart in the program trace to execute in parallel. This is necessary to detect parallelism that could be exposed by aggressive compiler transformations, such as loop interchange. Since the simulator cannot record the data dependences in a limited scheduling window, it records the time of the most recent write to each register and memory location. A large hash table is used to record writes to memory. Each bucket in this table may contain entries for several different locations, and additional space is allocated if a bucket overflows.

For each instruction, the simulator first determines when the operands are available. This is straightforward for register operands. For memory operands, we read the actual address from the trace and check the hash table for the time of the last write to that address. Next the simulator determines when the control flow constraints are satisfied. For the BASE machine, the execution time of the most recent branch in the trace is

[1]This assumes that the compiler has identified the induction variables and performed strength-reduction.

recorded, and all subsequent instructions must wait until that time. The implementations of the control flow constraints for the other machine models are described below. Given the constraints of control flow and operand availability, the instruction execution time is the minimum time satisfying all of these constraints. Finally, the simulator records when the instruction result is written. For store instructions, the actual address of the destination is read from the trace, and the time is entered into the hash table.

4.4.1 Control Dependence Analysis

For the CD and CD-MF machines, the control flow constraint is that an instruction cannot execute until after its immediate control dependence branch has executed. Control dependence analysis is performed in two stages. We first compute the control dependences within each procedure by analyzing the object code. Interprocedural control dependences are handled dynamically as the traces are analyzed.

To analyze the control dependences within a procedure, we must first build a control flow graph. We use the MIPS pixie tool to identify the basic block boundaries. We then decode and analyze the instructions from the object file to determine the successors of each basic block. Using the flow graph, the analysis computes all of the immediate control dependences by finding the reverse dominance frontier of each basic block [3]. All of the instructions within a basic block are immediately control dependent on the branches in the reverse dominance frontier of the block.

An instruction may be immediately control dependent on multiple branches. However, each dynamic instance of an instruction only depends immediately on one of these branches. For example, instruction 2 in Figure 2 is control dependent upon both instructions 1 and 5. If control flows from instruction 1 to instruction 2, we need to consider only the dependence on instruction 1. This is accomplished by sequentially numbering each basic block in the trace, and as we analyze the trace, the simulator records for each basic block in the code the sequence number of its most recent instance. The immediate control dependence of an instance of an instruction is simply the branch within its reverse dominance frontier with the latest sequence number.

Interprocedural control dependences are handled by maintaining a stack that contains the control dependence information for each active procedure. This stack records the control dependence for each calling instruction and the sequence number at the start of each procedure. Each procedure simply inherits the control dependence of the instruction that calls that procedure. Without recursion, the control dependence for an instance of an instruction is either the control dependence on the top of the stack or an instance of a branch in its reverse dominance frontier, whichever is most recent.

With recursion, the control dependence for an instruction is either the dependence on the top of the stack or an instance of a branch in its reverse dominance frontier *from the same procedure invocation*. For expediency, our simulations do not keep track of all the necessary information to accurately compute the control dependence in a recursive procedure. For each basic block in the code, along with the sequence number of its most recent instance, the simulator also remembers the sequence number at the start of its procedure. Recursion is detected when any branch in an instruction's reverse dominance frontier has a procedure sequence number greater than the current procedure. At that point, to provide an upper bound on the control dependence, we simply ignore the control dependence for that instance of the instruction.

4.4.2 Speculative Execution

Our simulations of speculative execution use static branch predictions based on profile information. These statistics were collected from running the benchmarks with the same inputs used in the simulations. Our prediction rates are therefore an upper bound for static branch prediction techniques. Dynamic techniques provide similar performance [8]. Table 2 shows the branch prediction rates for conditional branches in each benchmark. We do not attempt to predict computed jumps.

A mispredicted branch in a trace is easily identified by comparing the actual branch outcome with the predicted outcome. The simulator for the SP machine simply remembers the execution time of the most recent mispredicted branch, and no subsequent instructions in the trace can execute before that time. For the SP-CD and SP-CD-MF machines, an instruction must wait for its last mispredicted control dependence branch. By recording for each branch in the code whether its most recent instance was mispredicted, the simulator can compute this control flow constraint.

Program	Prediction Rate	Dynamic Instructions Between Branches
awk	93.48	6.8
ccom	92.02	7.5
eqntott	91.92	3.4
espresso	85.64	6.0
gcc (cc1)	89.29	7.9
irsim	87.71	6.7
latex	87.11	9.4
matrix300	99.02	20.0
spice2g6	97.66	13.1
tomcatv	99.09	58.8

Table 2: Branch Statistics

5 Evaluation and Analysis

The parallelism for each machine model is shown in Table 3. The following sections examine these results for the non-numeric benchmarks and discuss the implications. The results for the numeric benchmarks are presented in Section 5.3, and Section 5.4 discusses the effects of perfect loop unrolling.

The BASE machine provides a standard for comparison by determining the amount of parallelism when no special effort is made to reduce the control flow constraints. For the non-numeric benchmarks, the BASE machine has a harmonic mean parallelism of 2.14, which is larger than in other studies of similar machines because our assumptions are very different. First, the BASE machine allows some instruction scheduling across basic blocks. An instruction can execute as soon as the previous conditional branch is resolved, even if other instructions before the branch have not completed. In addition to this overlap, basic blocks that are not separated by conditional branches may also be executed in parallel. Second, whereas all operations in our machines execute in only one clock cycle, some previous studies used realistic operation latencies. Since non-unit latency operations consume some parallelism to fill pipeline bubbles, the reported speedups do not measure all of the parallelism. Finally, we do not include any limitations on fetching instructions [14].

At the other extreme is the upper bound of the ORACLE machine. As expected, the amount of parallelism is quite large and varies significantly between benchmarks. This reflects the different types of algorithms in the programs. For example, `eqntott` primarily executes a quicksort function which contains few data dependences. On the other hand, `ccom` and `latex` are composed of algorithms with much less inherent parallelism. Several factors cause our numbers to be considerably larger than in previous experiments with perfect branch prediction. Our unlimited scheduling window exposes parallelism across the entire program trace. Anti-dependences and output data dependences are not considered. Our simulation of procedure inlining also removes the instructions that adjust the stack pointer at the entry and exit of most procedures. This is significant in the case of the ORACLE machine because the stack pointer increments and decrements often lengthen the critical path of a program.

5.1 Control Dependence Analysis

The CD machine has more parallelism than the BASE machine because basic blocks with the same control dependences can be executed in parallel. However, the harmonic mean parallelism of 2.39 for the CD machine is only slightly better than for the BASE machine. Figure 4 shows the parallelism for each benchmark compared to the BASE machine. The parallelism for the CD machine is primarily limited by the constraint that branches must be executed in order. Since conditional branches occur frequently in our benchmarks, executing one branch at a time is a serious bottleneck. Table 2 shows the average number of dynamic instructions between conditional branch instructions in the program traces. For the non-numeric programs, a branch instruction occurs about every six instructions in a trace. When all of these branches are ordered, it is difficult to find much parallelism.

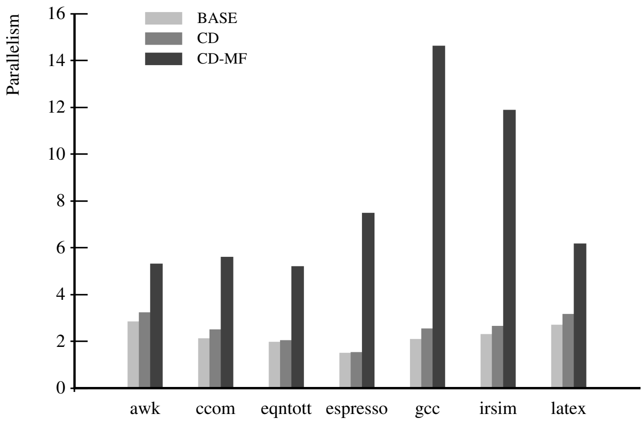

Figure 4: Parallelism with Control Dependence Analysis

When the constraint on branches is removed in the CD-MF machine, only the true control and data dependences must be observed. The parallelism for each benchmark is shown in comparison to the parallelism for the CD machine in Figure 4. The parallelism increases for all of the programs, and especially for `gcc`, `irsim`, and `espresso`. However, there is still not a massive amount of parallelism. This is really not too surprising when one considers the types of benchmarks that we are analyzing. There may be some parallelism within individual components of these programs, but the overall algorithms are simply not very parallel.

Since the constraints for the CD-MF machine only require that true data and control dependences be observed, the parallelism for this machine is a limit for all systems without speculative execution. Dataflow architectures, for example, are able to execute programs with only these essential dependences. Since there are not massive amounts of parallelism, any machine attempting to exploit parallelism in non-numeric programs without speculative execution must have low overhead to be effective.

5.2 Speculative Execution

The parallelism for the SP machine ranges from 4.16 to 9.22, with a harmonic mean of 6.80. Figure 5 shows the parallelism for each benchmark compared to the BASE machine. These results are comparable to Wall's results for a similar machine [17]. The differences can be attributed to procedure inlining, perfect loop unrolling, and the unlimited scheduling window in our simulator. The parallelism for the SP machine is fairly consistent across the different benchmarks. The following measurements offer an explanation for the consistency and suggest that this limit of parallelism will probably apply to many more non-numeric applications.

In the SP machine, a misprediction cancels the execution of all instructions following the branch, so mispredic-

	BASE	CD	CD-MF	SP	SP-CD	SP-CD-MF	ORACLE
awk	2.85	3.24	5.32	9.22	12.89	41.88	242.77
ccom	2.13	2.51	5.61	6.92	9.83	18.05	46.80
eqntott	1.98	2.05	5.21	6.40	18.09	225.90	3282.91
espresso	1.51	1.54	7.49	4.16	19.55	402.85	742.30
gcc (cc1)	2.10	2.55	14.63	7.76	13.18	66.29	174.50
irsim	2.31	2.66	11.89	8.40	15.82	45.86	265.42
latex	2.71	3.17	6.18	7.60	9.72	18.65	131.69
Harmonic Mean	2.14	2.39	6.96	6.80	13.27	39.62	158.26
matrix300	293	432	68324	36192	108575	180632	188470
spice2g6	2.14	2.29	16.80	8.11	25.28	196.76	843.60
tomcatv	22.23	42.77	3237	124	1881	3918	3918

Table 3: Parallelism for each Machine Model

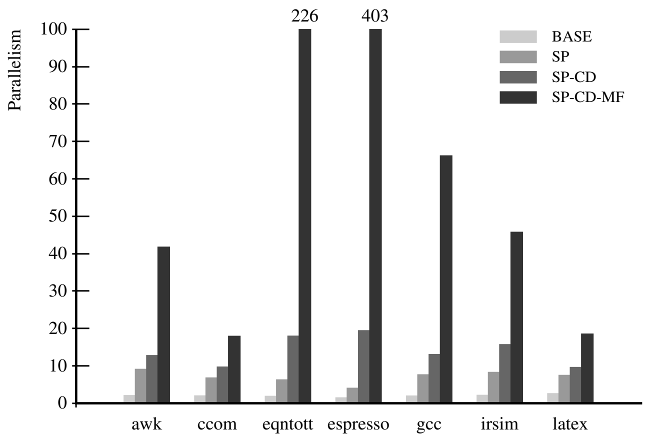

Figure 5: Parallelism with Speculative Execution

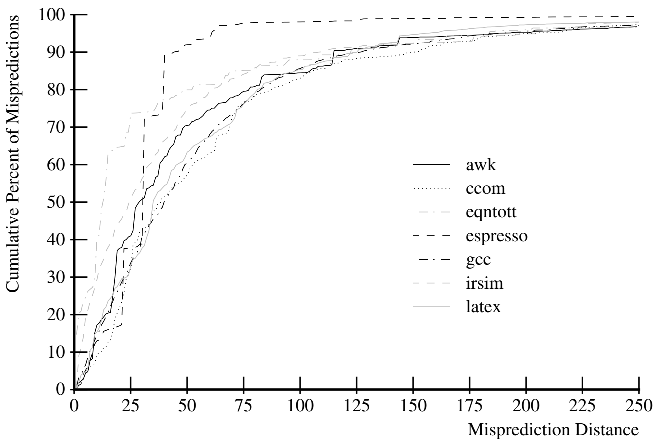

Figure 6: Cumulative Distribution of Misprediction Distances

tions are barriers to instruction scheduling. Parallelism can only be found among the instructions between mispredicted branches. Therefore, the overall limit of parallelism for the SP machine is actually an average over many discrete segments of code separated by mispredicted branches. Each of these segments has two vital characteristics: the degree of parallelism and the *misprediction distance*, that is, the number of instructions in the segment. In our experiments, we recorded the number of occurrences of each misprediction distance. The cumulative distributions of these misprediction distances for each program are shown in Figure 6. These distributions are quite consistent across the different benchmarks, with over 80% of the mispredictions occurring within a distance of 100 instructions. We expect other non-numeric programs to have similar distributions.

We also recorded the degree of parallelism for each segment of code between two mispredicted branches and found that the relationship between the degree of parallelism and the misprediction distance is similar for all of the benchmarks. Figure 7 is a combination of the statistics for all of the programs. For each misprediction distance, we plot the harmonic mean of the parallelism for all segments of that size. To reflect differences in the significance of these numbers, the bars for frequently occurring misprediction distances are shaded darker. For short misprediction distances, there is little parallelism. Instructions within these short segments tend to be closely related and have many data dependences between them, and thus the parallelism is limited. For longer misprediction distances, there is a greater chance of having unrelated instructions within the segments and more parallelism can be found. However, as shown by the distributions, long misprediction distances do not occur very frequently. Therefore, non-numeric programs with predominantly short misprediction distances have limited parallelism on the SP machine due to the data dependence in short segments of instructions.

The SP-CD machine does not need to discard all instructions following a mispredicted branch, and thus it can exploit parallelism across mispredicted branches. As a result, the harmonic mean parallelism for this machine increases to

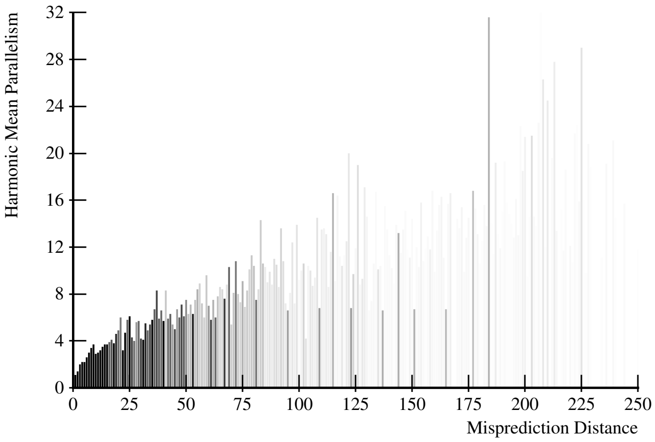

Figure 7: Parallelism vs. Misprediction Distance

13.27. Figure 5 compares the parallelism for each benchmark to the parallelism for the SP machine. The branch constraint for this machine requires that a branch cannot execute before a preceding misprediction. This is much less restrictive than the branch constraint for the CD machine. The flow of control only changes when a branch is mispredicted, and since mispredictions are relatively infrequent, the branch constraint is not a bottleneck.

Finally, the parallelism for the SP-CD-MF machine is much larger. Figure 5 illustrates the parallelism for each benchmark. The `eqntott` and `espresso` programs have especially large amounts of parallelism, and there is a large increase for all of the benchmarks.

The SP-CD-MF model provides us with an interesting data point. It is more aggressive than the SP machine but also more realistic than the ORACLE machine. To achieve the performance of the ORACLE machine, instructions from alternate paths must be executed simultaneously, and only the instructions from one of the paths will be useful. On the other hand, as long as branches are correctly predicted, the SP-CD-MF machine does not have to cancel any instructions.

5.3 Numeric Applications

Numeric programs, programs which operate on floating-point data and which are commonly written in FORTRAN, are generally considered to contain more parallelism than non-numeric programs. This section examines the parallelism for our three FORTRAN benchmarks and shows that the type of control flow in a program is a more useful indication of the available parallelism. The parallelism measured by our experiments for the FORTRAN benchmarks is shown in Table 3.

The `matrix300` and `tomcatv` programs have much higher parallelism for all of the machine models. From the ORACLE machine, we observe that there is less data dependence in these programs. The CD-MF machine achieves a large fraction of the ORACLE machine parallelism. Because the control flow in these programs is not dependent on the results of the computation, the control dependence analysis exposes parallelism across different levels of nested loops and across outer loops. As a result, speculation is not as important. In comparison to the CD-MF machine, the SP-CD and SP-CD-MF machines only compress the critical path of each inner loop by a small constant factor. Thus it is the data independent control flow that sets these programs apart from the non-numeric programs.

Among the FORTRAN benchmarks, the behavior of `spice2g6` is clearly different. The control flow in `spice2g6` is highly data dependent, thus causing it to behave like the non-numeric programs in this study. As numeric programs evolve to model more complex phenomena, they are likely to have increasingly complex control flow and data structures. Distinctions based on the source language and type of arithmetic will become less meaningful. Our study suggests that a more relevant characteristic for predicting the parallelism in a program is whether the control flow is data dependent.

5.4 Effects of Perfect Loop Unrolling

Previous studies on limits of parallelism did not remove all of the dependences on induction variables [17]. A question often raised is whether the induction variable dependences significantly affect the results of these studies. We performed two experiments, one with and one without removing the induction variable dependences. Table 4 shows the percent change in parallelism compared to the case when perfect loop unrolling is not performed. That is, a positive percent change means that removing the induction variable dependences improves parallelism. We discovered that removing these data dependences and the associated control dependences has mixed effects.

Although our simulation of perfect loop unrolling always decreases the program execution times, this does not necessarily imply that the parallelism increases. In fact, perfect unrolling has two competing effects on parallelism. By removing index variable dependences and loop branches, more parallelism is exposed. However, at the same time, removing the loop overhead instructions decreases the opportunities for overlapping those instructions with the rest of the computation in the loop, thereby decreasing the parallelism. Either one of these effects may dominate depending on the benchmark and the machine model.

For most of the non-numeric programs, unrolling has little effect. For example, `ccom` and `latex` have almost no change at all. These programs primarily contain loops with a lot of control and data dependences, so the dependences removed by unrolling are not very significant. The loops in these programs also tend to iterate a small number of times.

The CD-MF machine is most sensitive to perfect unrolling. Removing induction variable dependences allows

	BASE	CD	CD-MF	SP	SP-CD	SP-CD-MF	ORACLE
awk	30	36	10	48	52	41	-22
ccom	0	1	2	3	2	-2	-2
eqntott	-1	-1	-54	11	11	-4	3
espresso	-6	-6	134	-2	-16	15	-21
gcc (cc1)	0	2	-2	14	18	-3	-4
irsim	2	3	9	17	4	-9	-9
latex	0	0	-1	0	0	0	29
matrix300	2911	4317	16	182136	5488	2	0
spice2g6	12	12	35	21	23	0	-1
tomcatv	47	126	-9	149	13	-12	-12

Table 4: Percent Change in Parallelism due to Perfect Loop Unrolling

multiple iterations with arbitrary control flow to execute in parallel. This can improve parallelism, as in the case of `espresso`. However, the loop overhead constitutes much of the parallelism found in some loops. Thus, parallelism decreases when we remove such instructions, as in the case of `eqntott`.

Perfect unrolling has the biggest impact on `matrix300` and, to a lesser extent, `tomcatv`. These programs primarily execute simple loops where index variable dependences limit the parallelism. For these programs, the SP machine benefits the most from perfect unrolling. For nested loops, each iteration of an outer loop is separated by a mispredicted branch from the end of the inner loop. This prevents the outer loop iterations from executing in parallel. Perfect unrolling removes the loop branches, essentially coalescing the loops, so that these serializing mispredictions do not occur.

In general, the effects of loop index and induction variable dependences on parallelism vary depending on the application program and the machine model. As expected, the matrix-oriented numeric programs benefit significantly from perfect loop unrolling. For programs with complex control flow, unrolling often makes no significant difference.

6 Conclusion

This paper shows that control flow in a program can severely limit the available parallelism. The control flow of many non-numeric programs and also some numeric programs is complex and highly data dependent. To increase the available parallelism beyond the current level, the constraints imposed by control flow must be relaxed.

This paper discusses three basic techniques for handling control flow: speculative execution, control dependence analysis, and following multiple flows of control. Through a study of abstract machines that utilize different combinations of these techniques, we have established the importance of each technique. These basic techniques also form a useful set of criteria with which to evaluate real architectures.

This study suggests that some of the current highly parallel architectures lack adequate support for control flow. For example, a VLIW (Very Long Instruction Word) machine can only follow one flow of control. It cannot find sufficient parallelism in programs whose control flow is highly data dependent. In contrast, a dataflow machine can execute from many different parts of a program simultaneously. However, our study shows that even if instructions are executed as soon as all their data and control dependences are satisfied, the parallelism is still quite limited. Speculation is necessary to find sufficient parallelism in these programs.

This study of abstract machines also helps to identify useful architectural features. The concept of boosting [15], which relies on software for scheduling and a small degree of hardware to support speculative execution, appears particularly promising. Another interesting concept is *guarded instructions* [5]. A guarded instruction is conditionally executed based on a value stored in a general register. This allows a compiler to specify some amount of control dependence information, that only the action is control dependent on the guard. Furthermore, using guarded instructions, a basic block can contain code from different conditional statements by simply capturing their conditions in the guards. Guarded instructions are particularly interesting when combined with support for speculative execution, since they help increase the distance between mispredicted branches. Though guarded instructions give the processor some ability to execute from different regions of the source code, they are inefficient for following multiple complex flows of control simultaneously. If higher performance is desired, a small-scale multiprocessor system with guarded instructions and speculative execution support would be an interesting possibility.

References

[1] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming*

Language Design and Implementation, pages 241–255, June 1991.

[2] R. P. Colwell, R. P. Nix, J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, C-37(8):967–979, Aug. 1988.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

[4] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[5] P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395, June 1986.

[6] W. W. Hwu, T. M. Conte, and P. P. Chang. Comparing Software and Hardware Schemes For Reducing the Cost of Branches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 224–233, May 1989.

[7] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[8] S. McFarling and J. Hennessy. Reducing the Cost of Branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–404, June 1986.

[9] K. Murakami, N. Irie, M. Kuga, and S. Tomita. SIMP (Single Instruction stream/Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 78–85, May 1989.

[10] A. Nicolau and J. A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11):968–976, Nov. 1984.

[11] Y. N. Patt, S. W. Melvin, W. Hwu, and M. Shebanow. Critical Issues Regarding HPS, A High Performance Microarchitecture. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 109–116, Dec. 1985.

[12] E. M. Riseman and C. C. Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, Dec. 1972.

[13] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[14] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Apr. 1989.

[15] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, May 1990.

[16] G. S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.

[17] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Apr. 1991.