McGill University
School of Computer Science

Speculative Execution and Branch Prediction on Parallel Machines

ACAPS Technical Memo 57
December 21, 1992

Kevin B. Theobald
Guang R. Gao
Laurie J. Hendren

A version of this paper appears in the Proceedings of the
7th ACM Intl. Conf. on Supercomputing, Tokyo, July 1993

e-mail: author's-surname*@cs.mcgill.ca*

Advanced Compilers, Architectures and Parallel Systems Group

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Several recent studies on the *limits of parallelism* have reported that the use of speculative execution can lead to large increases in the amount of exploitable parallelism in a program, especially non-numerical programs. This is true even for parallel machines models which allow multiple flows of control. However, most architectural techniques for speculative execution and branch prediction are geared toward conventional computers with a single flow of control, and little has been done in studying speculative execution models and techniques which are applicable to parallel machines with multiple threads of control.

This paper presents a model of speculative execution for parallel machines. We define two different types of speculation (conservative and aggressive), and define the level of speculation (how far ahead the speculation can go). Conventional techniques for branch and jump prediction must be altered in order to work with this model, so we show how this is done for a variety of the common static and dynamic prediction strategies.

This paper presents a comprehensive quantitative study of: (1) how parallelism is affected by the speculative execution and branch prediction techniques under a parallel model of execution, and (2) what speculation depth is required to get a large portion of available parallelism.

We measure the parallelism limits of 5 long benchmarks (4 non-numerical) with different speculation models and branch prediction methods, and compare the results.

Keywords: branch prediction, control dependence, instruction-level parallelism, parallel architectures, speculative execution.

Contents

List of Figures

List of Tables

1 Introduction

For those designing highly-parallel computers, an important question to ask is how much parallelism is inherent in the applications, independent of any particular architecture. There have been many studies over the years attempting to answer this question [1, 2, 6, 7, 8, 9, 11, 13, 16, 18, 19]. Though the methods and models vary from paper to paper, all of these studies simulate real benchmark programs on idealized models that attempt to remove specific architectural limitations, e.g., by allowing for unlimited hardware-renaming of registers, assuming perfect alias analysis by the compiler, etc. Thus, the authors hope that the results give upper bounds on the parallelism achievable by real architectures. Such studies can give important insights into what is needed at both the architectural level and the algorithmic level for future generation parallel supercomputers.

One fact which is shown in some of the studies is that in most programs, particularly non-numeric applications, conditional branches and jumps significantly reduce the amount of parallelism which can be exploited in such programs. This is because a conditional branch or jump creates uncertainty about whether or not a particular piece of code will actually be executed, so that code can't be executed until the branch outcome is known. Opportunities for parallelism can be increased by performing *control-dependence analysis*, in which a program trace is analyzed to decide on exactly *which* branch each block of code is dependent. Even so, many non-numeric programs have low parallelism limits [9]. The same programs usually have dramatically higher parallelisms when the effects of branches are completely ignored, e.g., by using an *oracle* model [11].

Consequently, there has been increasing interest in getting more parallelism from programs through *speculative execution.* This means executing the code at one or more destinations of a branch before the branch outcome is known. This has primarily been used to try to prevent branches from disrupting long execution pipelines. For instance, some machines issue instructions at the "not-taken" destination of a branch through the pipeline, cancelling them if necessary. Such architectures may also prefetch the instructions at the other destination, just in case. A more ambitious approach is to use the past history of branches to try to predict the most likely destination, and to prefetch along that path. Various algorithms for *branch prediction* are given in [10, 14]. There has been a lot of effort toward finding branch prediction methods that increase the rate of successful predictions. Some are looking at doing more than improving prefetching. One superscalar uses *boosting* [15] to execute down the most likely path many instructions before a branch is resolved, using "shadow registers" in order to maintain two program states which are made consistent once the branch has been resolved.

Therefore, some researchers on instruction-level parallelism have incorporated speculative execution into their idealized models, in order to get an idea of how much parallelism may be obtained through this method. The work most relevant to this paper includes Wall [19], Lam and Wilson [9], and Theobald, Gao, and Hendren [16].

This paper ties these two lines of research (speculative execution and branch prediction) together. We wish to study how branch prediction and speculation interact, and what impact this has on program parallelism. In particular, we would like to know what kinds of speculative execution and which branch prediction strategies lead to the highest potential amounts of parallelism.

We also want to understand how branch prediction will work on a parallel machine. The existing dynamic branch prediction algorithms predict branches using the previous history of branches. Unfortunately, since, on a parallel machine, many instructions may be executed concurrently, the "branch history" won't have the same meaning as it does on a sequential machine. Therefore, we must modify these algorithms to a form appropriate for multiprocessors.

As well as providing the framework for models of speculation and branch prediction, we also present a quantitative study on the effects on parallelism by speculative execution under parallel machine models. We have defined a parallel model of speculative execution based on the concept of *Dynamic Control Dependence Tree* (DCDT). In our experiments, for each branch the machine only performs speculation on one path (outcome) — we call this the *one-path-per-branch* scheme (Note, of course, that there may be many concurrent branches in such a one-path-per-branch speculative execution!). The following is a summary of the major results.

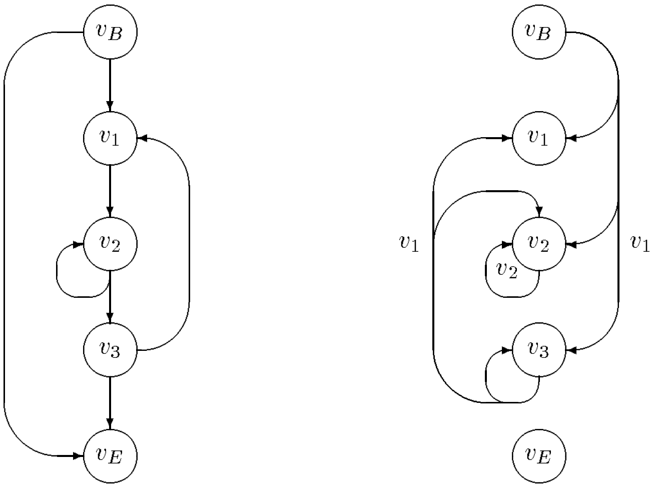
- In all benchmark programs studied, speculative execution has a significant effects on parallelism limits. Moreover, a machine performing one-path-per-branch speculative execution could get impressive performance improvements over similar machines without speculation.
- In most cases, only a limited amount of speculation is needed to get a large fraction of the parallelism which would be available if the machine could speculate to an infinite depth (around 5 levels deep to get 50%, and around 10 levels deep to get 90%).
- In most cases, static branch prediction works as well as dynamic prediction (based on branch history). In only one case did dynamic prediction lead to substantially better performance.
- The studies show that branch prediction success rates do not always correlate well with overall parallelism. This should not come as a complete surprise, as some similar observations on the relation between success rate and overall program performance has been reported for sequential machines [5].

In the next section, we present a parallel model of speculative execution based on *Dynamic Control Dependence* relationships between instances of basic code blocks, using the notion of a *Dynamic Control Dependence Tree* (DCDT). Based on this relationship, we define *levels of speculation.* Two types of speculation, called *conservative* and *aggressive,* are presented. In Section 3, some of the well-known static and dynamic methods of branch and jump prediction are modified to work with the execution model in Section 2. Section 4 presents results of an empirical study measuring the parallelism limits of 5 long benchmarks (4 non-numerical) with different speculation models and branch prediction methods. Related work is discussed in Section 5. The final section, Section 6 summarizes the paper and discusses the conclusions that may be drawn from this study.

2 Speculative Execution

The basic purpose of *speculative execution* is to speed up program execution by running some code segments before it is known whether or not they are actually reached. To illustrate the concept, consider the simple program represented in Figure 1. Part (a) shows a *Control Flow Graph* (CFG) [4] showing all the basic blocks of a program as vertices, and the possible paths from one block to another as directed edges. The diagram might represent, for instance, a 2-dimensional *for* loop. Basic block 2 (v_2) is the inner loop, while v_3 is the end of the outer loop. The nodes v_B and v_E represent the beginning and end of the program, and don't correspond to any real code. An arc connects them to represent the decision of whether or not to run a program.

Without any speculation, a machine can't execute beyond v_2 until it knows which way the branch at v_2 is taken. To run faster, the machine could assume that the branch will be taken (to the start of v_2) and begin another iteration of v_2 before the first branch is resolved (assuming this is not prevented by data dependencies). Since the branch might be taken the other way, it may be necessary to eliminate the



(a) (b)

Figure 1: Example of Control Flow Graph and Static Control Dependence

state changes caused by the extra execution of v_2. This can be done by backtracking, or by maintaining two machine states which are coalesced once the branch outcome is known. The latter is done in an existing superscalar [15].

It might appear that v_3 can't execute until all iterations of v_2 have been completed. However, since v_2 inevitably leads to v_3, it is possible to execute both concurrently, assuming that all other dependencies are met. In fact, the execution of v_3 is certain as soon as the program begins. To get more parallelism out of a program, a computer can start executing v_1, v_2 and v_3 simultaneously, even without any speculation. These opportunities for additional parallelism are uncovered through *control-dependence analysis*, covered in the next section.

Speculation can be combined with control-dependence analysis to achieve even more parallelism. For instance, if v_1, v_2, and v_3 are executed simultaneously, the machine could speculate along the branch at v_2 and begin another iteration of v_2. Simultaneously, it could also speculate along the branch at v_3 and begin another instance of v_1.

In this section, we develop models of speculative execution for parallel machines capable of simultaneously following multiple flows of control. In Section 2.1, we introduce the concept of *dynamic control dependence*. Section 2.2 develops a model based on this concept, beginning with a parallel, non-speculative execution model. Speculation is added in Section 2.2.1. In Section 2.2.2, we define the notion of "levels" of speculation based on the dynamic control dependence. In Section 2.2.3, we describe two types of speculation models: conservative and aggressive.

2.1 Dynamic Control Dependence

In the example of the 2-D loop represented by the CFG in Figure 1(a), it was shown that v_3 doesn't need to wait until all instances of v_2 have completed. *Control dependence* [4] explicitly identifies this relationship. This form of dependence captures the idea that the choice of whether or not to execute a particular block is decided by a specific conditional branch or jump. Intuitively, block v_j is control-dependent on v_i if v_i has more than one exit, if some of the exits from v_i always eventually lead to v_j, and if, by taking other exits from v_i, it is possible to avoid v_j. The following definitions are adapted from [4]:

Definition 1 *A* Control Flow Graph *(CFG) is a connected directed graph, i.e., a pair (V, E) in which V is a set of vertices corresponding to the basic blocks of a program, and $E : V \times V$ is a set of edges. Edge (v_i, v_j) is a member of E iff v_j can follow v_i in the execution of a program. V contains special nodes v_B and v_E where*

1. *v_B has no predecessors and two successors, one of which is v_E;*
2. *v_E has no successors, and has v_i as a predecessor iff the program can terminate* immediately *after executing v_i;*
3. *For any vertex v_i, there is a path from v_B to v_i and a path from v_i to v_E;*

Definition 2 *A block v_i in G is* post-dominated *by a block v_j (written $postdom(v_j, v_i)$) if every path from v_i to v_E (not including v_i) includes v_j.*

Definition 3 *A block v_j in G is* control-dependent *on block v_i iff*

1. *There exists a directed path P from v_i to v_j such that any v_z in P (excluding v_i and v_j) is post-dominated by v_j;*
2. *v_i is not post-dominated by v_j.*

For example, in Figure 1(b), directed edges show the control dependence relations among the elements of the CFG in Figure 1(a). For instance, v_3 is control-dependent on both itself and v_B. The *first* occurrence of v_3 is predetermined as soon as the program begins execution. Subsequently, each instance of v_3 is determined by the execution of the branch at the end of the previous instance of v_3.

Note that v_3 only gets repeated if the branch in v_3 is *taken*; the program terminates if the branch is *not* taken. Therefore, each control-dependence relation is associated with the conditions necessary for the dependent block to be executed. This would be T or F for a conditional branch (a transfer to a fixed destination, which is either taken or not taken), and one or more destinations for a jump (a transfer in which the destination is computed at run-time[1]). We can unify the two formats by representing conditions with the actual destinations in the CFG. Thus, a conditional branch will have two destinations, while the jump will have two or more destinations. Therefore, each edge is labeled with the branch or jump outcome(s) which lead(s) to the eventual execution of the destination vertex, as illustrated in Figure 1(b).

This notion of control dependence with associated conditions can be formalized by a "dependence function" $D_{V,E}$ which is specific to a particular CFG (V, E). Each edge in the CFG represents the act of taking a branch or jump to a particular destination. For each edge (v_i, v_j), $D_{V,E}((v_i, v_j))$ is the set of all nodes whose execution becomes imminent when the conditional branch or jump at v_i is taken to destination block v_j. A block is in this set if it doesn't post-dominate v_i, but either post-dominates v_j or *is* v_j. The statement that a block is the same as v_j or post-dominates v_j is equivalent to condition 1 in Definition 3. (Similar information is contained in the *Control Dependence Subgraph* [4], where a region node groups together all nodes control-dependent on the same outcome of a particular branch or jump.) The following is a definition of the dependence function.

[1] A return to a location which was stored on a stack isn't considered a jump

Definition 4 *A* dependence function *(or D-function for short)* $D_{V,E}$ *is a function* $E \longrightarrow V^*$ *where*

1. $v_k \in D_{V,E}((v_i, v_j)) \iff (v_i, v_j) \in E \wedge \neg postdom(v_k, v_i) \wedge (v_k = v_j \vee postdom(v_k, v_j))$
2. $(v_i, v_j) \notin E \Rightarrow D_{V,E}((v_i, v_j)) = \emptyset$

For example, in the CFG of Figure 1, $D_{V,E}((v_2, v_2)) = \{v_2\}$ and $D_{V,E}((v_3, v_1)) = \{v_1, v_2, v_3\}$.

Control dependence as shown is a static relationship among basic blocks in a CFG. Each of these blocks can correspond to many instances in the actual execution of the code. In a CDG, a block may be control-dependent on more than one other block. For instance, block v_2 in Figure 1 is control-dependent on itself, v_3, and v_B. However, the execution of each particular *instance* of v_2 is determined by a different block instance which is unique. As soon as the program begins, it is predetermined that the first execution of v_2 will occur. Each succeeding iteration of the inner loop is determined by the previous instance of v_2. The first iteration of the inner loop within the second iteration of the outer loop is dependent on the conditional branch at v_3.

To charcterize such "dynamic" control dependence between block instances in the execution, we introduce the *Dynamic Control Dependence Tree* (DCDT). This generalizes the idea illustrated above, where the execution of different instances of a basic block can be determined by different conditional branches. It does this by explicitly identifying the unique branch on which each instance is control-dependent. Intuitively, the CFG is "unrolled" (using its control-dependence function D) into a tree corresponding to a specific instance of a program execution. The following operational model will produce a DCDT from a given CFG and its function D, according to the following *control-firing rules*:

- Initially, the start node v_B is *control-enabled*, and its execution instance becomes the root of the DCDT.
- Assume that a node (block) v_i corresponding to instance s_j in the DCDT is control-enabled. Then this block can be "fired" (executed), provided that all relevant *data* dependencies have been satisfied. If v_i only has one successor in the CFG, then s_j is a leaf; nothing is control-dependent on it (it doesn't contain a branch or jump). If v_i has more than one successor in the CFG, then it is a branch or jump. Since a particular firing of a node represents a *single* instance of a block, the branch or jump has a particular outcome. If that outcome leads to block v_k, then all nodes in $D_{V,E}((v_i, v_k))$ become control-enabled. Corresponding nodes are created in the DCDT as the children of s_j, and the arcs leading to the children are labeled with v_j.

For instance, Figure 2 shows a DCDT corresponding to a particular (parallel) execution of the 2-D loop. Readers are encouraged to generate the tree from the CFG following the rules descibed above.

Thus, a DCDT represents the partial ordering of block instances necessitated by all the true control dependencies for an execution of the program. Two unrelated blocks which happen to be separated by a conditional branch can run in parallel if neither is control-dependent on the other. Note that we only consider control dependence here. There may be data, anti, and output dependencies as well. For instance, v_3 may increment an index variable which is used by v_2, which is an anti-dependency, so that instruction in v_3 would have to wait. We assume that some kind of interlocks or dataflow scheduling are supported by the machine model which can prevent hazards arising from data dependencies, while register and memory renaming or a single-assignment rule can prevent hazards from anti-dependencies.

Just as a dynamic program trace is a useful way to describe the relations between instances of instructions executed in a program run, a DCDT is conceptually useful to characterize the partially-ordered history of branch/jump operations of the corresponding program execution. What is the relation

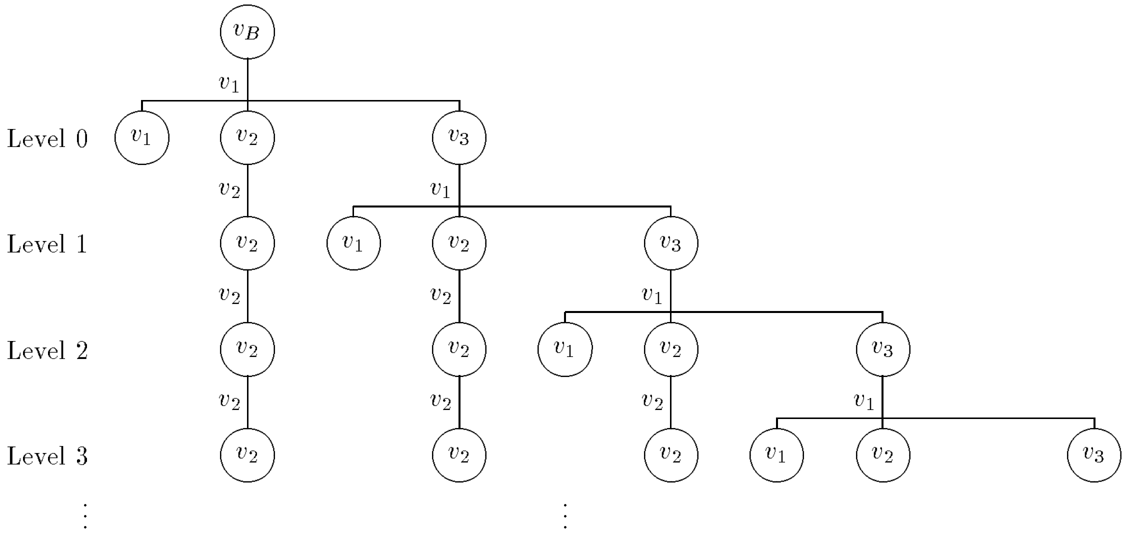


Figure 2: A DCDT Corresponding a Parallel Execution of the 2-D Loop

between a DCDT and a dynamic trace? A dynamic trace of a sequential execution of a program corresponds to a preorder depth-first traversal of the DCDT, assuming that corresponding branch/jump outcomes are the same. Although a formal proof is beyond the scope of this paper, there is a simple intuitive explanation of this result:

If $D_{V,E}((v_i, v_j))$ is non-empty, then one of the members of that set must be the destination v_j. That node will obviously immediately follow v_i. The eventual execution of every other member of that set is also inevitable, by definition, and they will occur in a particular order (since they are being executed sequentially). Now suppose v_x and v_y are members of $D_{V,E}((v_i, v_j))$, and v_y follows v_x. Then every block which is directly or indirectly dynamically control-dependent on that instance of v_x (i.e., a descendent of the corresponding instance in the DCDT) *must* be executed before the instance of v_y. Otherwise, we would have the absurd sitation of a block which is dynamically control-dependent on an instance of v_x *following* an instance of block v_y, even though the execution of that instance of v_y *can't* be affected by the outcome of the branch or jump in the instance of v_x.

For instance, It is clear from the drawing of the DCDT that the order of blocks in a sequential trace of an execution of the 2-D loop would follow a preorder, left-to-right, depth-first traversal of the DCDT.

2.2 A Parallel Execution Model

The CFG and the corresponding DCDT can be used to create a model of parallel execution in which the effects of control dependence are minimized. In the previous section, as a CFG was "unrolled" into a DCDT, vertices in the CFG were successively control-enabled, and each new "enabling" of a node corresponded to a node in the DCDT. Time can be added to this model. We can say that a node begins execution as soon as it is control-enabled, and that its children (if any) are control-enabled some time later, after its branch or jump has been completed. We say that an instance of a block is *resolved* at the time that its branch or jump has been computed. (This could occur before all instructions in that block have completed, if the instructions are reordered so that the branch predicate is computed before the end of the block.)

Definition 5 0-Level Speculation *is a form of execution in which a node in a DCDT does not begin execution until its parent has been resolved.*[2]

2.2.1 Adding Speculation to the Model

Suppose a machine begins running according to the model, and starts executing v_1, v_2, and v_3 at the same time. If the branch in block v_2 is taken (destination is v_2), then another instance of v_2 will be initiated. A 0-Level Speculator would have to wait until the first instance of v_2 is resolved before starting another iteration.

However, a speculative machine could assume that the branch at v_2 will be taken, and start executing the next iteration of the inner loop before the *first* branch is resolved. Simultaneously, it may start new occurrences of v_1, v_2, and v_3 before the first occurrence of v_3 has finished. This would be an example of *1-Level Speculation*. The speculator would use some algorithm for predicting the outcome of each branch or jump (these will be covered in Section 3).

With speculation, the execution of the program would correspond exactly to the DCDT only if all branches and jumps were correctly predicted. That is, the DCDT only corresponds to the true dynamic control dependencies of the program. If the speculator makes a wrong prediction and begins executing a block at the wrong destination, it will have to cancel the state changes made by that block once the true branch outcome is known.

2.2.2 Levels of Speculation

It is possible to speculate down more than one level. For instance, the machine could start another occurrence of v_2 (on the assumption that the first speculated instance of v_2 ends with a branch to repeat the loop) and another set of $\{v_1, v_2, v_3\}$ on the assumption that the outer loop will be repeated as well. Figure 2, the DCDT, can be used to view how the machine may speculate down many levels. In this case, it is assumed in the diagram that program execution has just begun. The labels at the left show the blocks associated with each level of speculation. In this instance, each new level of speculation involves starting another pass through each active inner loop and initiating another pass through the outer loop.

Note that at each branch, the machine only speculates down one path. A fully speculative machine could speculate down both paths of each branch, but since the program states would grow exponentially, *we only consider machines that speculate down one branch in this paper*. It is important to note that under a parallel execution model, even speculating down one destination can lead to an exponential growth in program states, since taking a single branch can lead to the concurrent execution of several basic blocks, each ending in its own branch. However, a higher percentage of the program states will be valid, since only the most likely paths are followed.

Clearly, the further down the machine speculates, the less certain it is about the blocks it is executing. The *speculation depth* represents the degree of certainty:

Definition 6 *The* speculation depth *of an instance of a basic block is the number of unresolved branches upon which that block instance is dynamically control dependent.*

[2] This is similar to the CD–MF model in [9].

This is a dynamic quantity which decreases with time. A block with depth 0 is definitely supposed to execute, because its parent (in the DCDT) has been resolved. A finite speculator can speculatively execute all blocks within a limited distance of a depth-0 block. Some of these blocks will have high speculation depths, which will either decrease over time as their ancestors are resolved one-by-one, or will disappear entirely when it is discovered that an ancestor's branch was incorrectly predicted.

Definition 7 n-level speculation *means the speculative concurrent execution of all block instances whose speculation depth is less than or equal to* n.

2.2.3 Types of Speculation

What happens when the branch or jump at the end of a block is resolved? This depends on whether the actual outcome of the branch or jump is the same as the predicted outcome, or different. It also depends on the current speculation depth of the branch.

Suppose that the resolved branch has a current speculation depth of 0. This means that there is no question that this block is supposed to be executed. Therefore, when this branch is resolved, whatever immediately follows this branch is determined (speculation depth 0). The machine should check the actual outcome of the branch against the prediction it made when speculating past that branch. If the resolved branch had been correctly predicted, the states which are the children of the branch in the DCDT, which formerly had a speculation depth of 1, have their speculation depth changed to 0. All descendents of this branch have their speculation depths decremented. Since the blocks with depth n have their depths reduced to $n-1$, the speculator can now speculate one level past those blocks. If the resolved branch had been incorrectly predicted, all states which are descendents of the branch are invalid, and must be removed. Computation may resume by re-traversing the CDG from the block in which the branch occurs, this time applying the D function to the *correct* destination.

What happens if the resolved branch has a current speculation depth greater than 0? This means that this branch is dynamically control dependent on branches which have not been resolved yet. Therefore, all we can say is that this branch *will* be taken, provided all preceding branches were predicted correctly. We define two types of speculation, which differ in their response to this case:

Conservative Speculation disallows speculating any new blocks until the branches in all ancestors in the DCDT have been resolved. If the branch had been mispredicted, all descendent blocks are immediately halted and their side effects canceled. However, no new execution may begin until the ancestor branches have been resolved. If the branch had been correctly predicted, all speculated descendents may continue executing, and their speculation depths are decreased, but speculation can't progress beyond the current limit until the ancestor branches are resolved.

Aggressive Speculation doesn't make this restriction. If the branch prediction was wrong, the machine traverses down the correct path, assigning a speculation depth to each child which is the same as the speculation depth of the resolved branch (since the branch is now resolved). From there, speculation continues until depth n is reached. If the branch prediction was right, the speculation depths of all descendents are decremented, and speculation progresses beyond the current limits by 1 level.

3 Branch Prediction

The previous section described different models of speculative execution for multiprocessor machines. The execution models assumed the existence of an algorithm for predicting branches. This section develops several algorithms, both static and dynamic, which are appropriate for the kind of speculative machine presented in the previous section.

3.1 Static Branch and Jump Prediction

Static prediction assigns a most likely outcome to each branch or jump instruction in the object code. Every time this instruction is encountered, the same prediction is made, and it never changes during program execution.

In this paper, three kinds of static branch prediction are considered (a "branch" is a conditional transfer which is either taken or not taken):

Take All: Predict that branches are always taken.

Take Back: Predict that backward-branches are always taken, while forward-branches are never taken. (This is based on the assumption that backward-branches mostly occur in *while* and *for* loops that are generally repeated more than once.)

Table: For each branch, predict the outcome by looking up the particular branch instruction in a table. The table contains the most frequent branch outcome of each instruction as seen in a previous execution of the same program. This is often based on statistics gathered by profiling information.

For the prediction of jumps, there are no methods corresponding to the first two branch prediction schemes above. Table-based prediction, however, can be done for jump prediction as well.

3.2 Dynamic Prediction

Dynamic prediction modifies the predicted outcome according to the history of actual outcomes of previous branches. It is hoped that the predictor can learn from and adapt to changing branching patterns. One general approach is to record the outcomes of the previous n occurrences of *each* branch instruction in the object code, and to base the prediction of a branch on the history specific to that instruction [10].

These dynamic methods must be modified for execution on the kind of speculative multiprocessor envisioned in this paper. In their original form, they assume a uniprocessor with a single thread of control. If the program only follows a single thread of control, then branches will occur in a deterministic order which can be recorded in the dynamic branch history states. This breaks down when processing is made concurrent.

For instance, on a uniprocessor, if a recursive procedure calls itself twice on different data, the outcomes of branches in the first recursive call will update the dynamic branch history, which will affect the predictions made for the same branches in the second recursive call. On a parallel machine, the two recursive calls may be executed concurrently. With no deterministic ordering between the instructions in the two procedure invocations, how can the branch outcomes in one procedure call affect the predictions in the other? The outcomes in the "first" invocation may be resolved at the same time, or even later, then the time of the predictions in the "second" invocation.

In the speculative models we have been discussing, the only definite ordering of conditionals are the orderings created by the dynamic control dependence relation. In other words, the only deterministic history of branches and jumps available to the prediction algorithm when trying to predict the conditional at a given instance of a block is that block's direct ancestors in the DCDT. All other branches may occur concurrently with these key branches, so the order of their occurrence, relative to each other and to the direct ancestors, is indeterminite.

Therefore, for the speculative machine described in Section 2, a dynamic branch prediction algorithm will base its prediction for a given instance of a block only on a subset of all the block instances which are ancestors (in the DCDT) of the block being predicted.

Most prediction strategies don't look at the entire history of branches or jumps from the beginning of a program, but only encapsulate this history in a fixed number of state bits. On a uniprocessor following a single thread of control, only a single state need be maintained and updated because of the total ordering of all branches in the program. On a parallel speculative machine, because different instances of blocks will have different sets of ancestors in the DCDT, they can't all work with the same state. Rather, a separate state is associated with each currently executing block instance which contains a branch or jump. When the machine speculates past one of these blocks, and initiates one or more dynamically-control-dependent blocks, it copies the branch history state into the program state of each of the dependent blocks, and updates these copies according to the way the branch was predicted.

If that branch is later resolved, and a misprediction is detected, all the states corresponding to blocks descendent from the mispredicted branch, including the relevant copies of the branch history state, are incorrect, and must be removed. If the branch is resolved and correct prediction is verified, then the copy of the history state in that branch can be deleted, as it is now known that all children have valid states.

3.2.1 Branch Prediction Strategies

In this study, four basic types of dynamic branch prediction are studied. These are taken from existing methods used for conventional, sequential machines. Each assumes a state (a set of counters) which is updated by successive branches. These techniques are adapted for the speculative machine as described above.

The methods are:

1-bit: For each branch, a single bit records the outcome of the most recent occurrence of that branch which is an ancestor (in the DCDT) of the current block, and that bit is used to predict the next branch.

2-bit: For each branch, two bits record an abbreviated history of previous occurrences of that branch. When a branch must be predicted, the prediction is based on the 2-bit state. The copied state is updated according to the predicted branch outcome. If it is later found that the branch was mispredicted, then the copied state is re-created according to the actual branch outcome. Three specific 2-bit prediction schemes are presented in [10] and shown in Figure 3. In each state diagram, a state represents one of the possible values of the branch history. Its label represents the prediction for the next branch based on this state. The arcs are labeled with the transitions, which represent either the predicted outcomes (if branches are being predicted) or the actual branch outcomes (if they are being corrected).

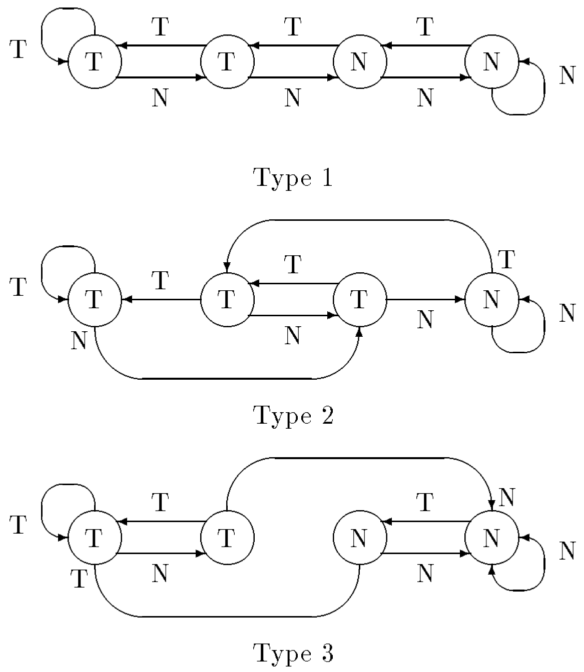


Figure 3: 2-Bit Branch Prediction State Diagrams

For each of these algorithms, several variations are possible. First, what is the initial value of each state value? In other words, what outcome should be predicted if, when a prediction must be made for a particular branch, there are no instances of that branch among the ancestors of that branch in the DCDT? We consider using the three alternatives for static prediction presented in Section 3.1 (take all, take back, table) as the default prediction or initial value for the state maintained for each branch. For the 2-bit schemes, there are four possible choices for the initial value but only two possible predictions (take or not take), so the initial value chosen is the "weakest" value, i.e., the value which corresponds to the statically-predicted outcome, yet is closest to being overturned by a wrong prediction.

Another variation involves the scope of the history states relative to procedure boundaries. Our experiments involve two alternatives:

Total Scope: We assume that history states are passed to called procedures, so that branch prediction within a procedure is affected by the history of branch outcomes prior to the procedure call. When a procedure returns, the branch history state as it was prior to the procedure call is restored.

Procedure Only: We can hypothesize that branch histories won't have any meaningful effect across procedure boundaries, and decide instead that history states are reinitialized (using one of the three defaults) whenever a new procedure is entered. (The original states are restored when the procedure returns.)

Thus, there are 6 ($= 2 \times 3$) variations of each of the four basic dynamic prediction techniques. We performed experiments involving all 24 variations, plus the 3 static techniques in Section 3.1. These are described in Section 4.

For some selected algorithms, we performed experiments involving an additional variation, which is based on a recent paper [12] which reported that branch prediction rates could be improved by correlating the branch histories based on specific branch address with the history of recent branches (regardless of address). Instead of associating a single 1- or 2-bit state with each branch address, the machine keeps 2^n such states *for each branch address*, and also keeps a record of the most recent n branches taken. Then, when predicting the outcome of a particular branch, the machine must first look up the n-bit pattern representing to the recent branch history, then look up the corresponding 1- or 2-bit history state for the particular branch.

3.2.2 Jump Prediction Strategies

Jump prediction is more complex than branch prediction, because there may be more than two destinations for a particular jump, and because it is not generally possible to tell from the object code what the possible destinations are. Therefore, the only dynamic jump prediction algorithm we consider in this study is a straightforward one. The predicted destination of a jump instruction is the destination of the previous occurrence of the same jump instruction.

Variants on this method are similar to the variants on branch prediction. The default initial state is either to make no prediction at all, or to predict the destination which was the most frequently occurring destination of that jump instruction in a previous run of the code. Also, like the branch prediction, the jump prediction can carry the "previous destination" history state across procedure boundaries, or else reset the history state to the default destination (none or table-lookup) whenever a new procedure invocation is entered.

4 Experiments

Our experiments can be categorized into three main groups. The first group of experiments measure the lower and upper limits of parallelism assuming no speculation for the lower limit, and assuming an oracle model for the the upper limit. These results are given in Section 4.2. We then narrowed our focus to interesting cases where there is a large difference between the lower and upper limits, which presumably means a wide range of possible results for the speculative models. The second group of experiments measure the theoretical limits of parallelism achievable by two speculation models and the wide variety of branch and jump prediction algorithms presented in the previous section. The thrust of these experiments was to determine which branch and jump prediction schemes work best when infinite speculation is allowed, and to compare conservative and aggressive speculation. These experimental results are given in Section 4.3. The third group of experiments were designed to study the effect of a finite depth of speculation, and to measure what finite depth of speculation is required to get close to the parallelism obtained for the models allowing infinite depth speculation. These experimental results are given in Section 4.4.

4.1 Experimental Framework

To perform these experiments, we used a tool called SITA (Sparc Instruction Trace Analyzer) which we developed to study program parallelism [17, 16]. The tool executes a benchmark program on a Sun SparcStation, generates a dynamic trace of all the operations performed in the execution, and then attempts to schedule these operations into "parallel instructions" according to how an idealized parallel architecture may try to execute them. This basic approach is used in many of the other parallelism studies (using different machines) [1, 9, 19].

The scheduler makes certain ideal assumptions when scheduling the operations. In some cases, these assumptions may appear to correspond to unrealistically ambitious hardware implementations, but the purpose of these studies is to measure the parallelism inherent in the program and algorithm itself, not the parallelism possible in a given architecture. All operations (including floating-point calculations and memory accesses) have 1-cycle latency. Anti and output dependencies are ignored, on the assumption that the effects of these can be eliminated through means such as register renaming and adhering to a single-assignment rule. Operations from arbitrary places in the dynamic stream can be scheduled into the same parallel instruction, and a parallel instruction can be filled with an unbounded number of operations. Further details as to how we configured SITA to perform the experiments are given in Appendix A. Basic implementation details are provided in [17].

We ran our tool on several benchmarks, each consisting of at least 100 million instructions. We looked for programs large enough to give figures suggestive of what can be achieved with "grand challenge" problems, without being so large as to overwhelm our analyzer. We ran all programs to completion, giving the analyzer enough opportunity to extract parallelism from all parts of the code.

Analysis of a program trace requires two passes. In the first pass, the instruction trace is broken down into its constituent basic blocks, and the frequency of each branch or jump is tabulated. This forms a CFG for the program in which the edges are labeled with their relative frequencies. The control dependence relations are derived from this CFG using an algorithm presented in [3].

4.2 The Range of Speculation

To set the stage for our complete experiments, we first ran some some preliminary tests on seven benchmarks (Table 1). The benchmarks consist of three regular FP-intensive scientific programs and four symbolic applications. All are from the SPEC benchmark test suite, except for tex, which is from the DLX suite, and a speech-recognition program taken from an actual industrial application.

Two preliminary tests were run to measure the upper and lower bounds of parallelism when control dependence is a factor. The upper bound was provided by the *oracle* model, which ignores control dependences completely. The oracle model measures only the effects of true data dependences. It shows what would be achieved by a machine that could speculate down *all* destinations of a branch or jump though an infinite number of levels, or, alternately, what would happen on an ideal machine if all branches were predicted correctly. Thus, an imperfect speculator will have a lower resulting parallelism. The lower bound was found by configuring SITA to model a 0-Level Speculator (a model that does not support any speculation at all, yet gets as much parallelism as possible through control dependence analysis).

Since the 0-Level Speculator is constrained by all control dependences, and the Oracle ignores these dependences, these two models represent the extremes of the effects of control dependences. An n-Level Speculator ($n > 0$) will be affected by those control dependences arising from branches and jumps which are mispredicted, and those which are deeper than the machine's ability to speculate. Therefore, the parallelism for a Speculator will lie somewhere between these two extremes.

The last two benchmarks, dodoc and tomcatv, show almost no difference between the 0-Level

Source	DLX	industry	SPEC				
Benchmark	tex	speech	espresso	eqntott	fpppp	doduc	tomcatv
Input	draft		bca.in	int_pri_3.eqn	4	small	257
Language	C				FORTRAN		
Compiler	Sparc	gcc 1.42.1	gcc 2.2.2	gcc 2.2.2	Sparc	Sparc	Sparc
Dyn. Ops (mil.)	107	544	427	1124	277	522	3018
Dyn. Blocks (mil.)	20	138	98	418	2.8	24	43
FP Ops (%)	.04	4.8	<.01	0	19	14	17
Loads (%)	16	15	24	18	43	36	48
Stores (%)	8.1	2.8	2.1	1.1	10	10	13
Oracle	189	7991	1363	28661	4974	615	8417
0-Level Speculator	5.31	45.4	2.81	3.28	1178	566	6014

Table 1: Parallelism Upper and Lower Bounds

Speculator and the Oracle. This is due to the regularity of their control structures. In both cases, there are a lot of loops in which the loop predicate is not data-dependent on the body of the loop (e.g., iteration across a fixed range). In such a case, a 0-Level Speculator could iterate and test the loop counter as soon as the loop body begins, so one loop iteration could begin every few cycles. Speculating past the test would save only a few cycles, because the initiation of loop iterations would still be constrained by the data-dependences involved in the incrementing of the loop counter. Therefore, only the first five benchmarks were selected for further study.

4.3 Infinite Depth Speculation and Branch Prediction

The next phase of the experiment was to make a complete sweep of the basic algorithms discussed in Section 3. SITA was run on the five interesting bencmarks: tex, espresso, speech, eqntott, and fpppp. Tables 2–6 summarize the parallism results for 27 different algorithms on both the conservative and aggressive models. The tables are divided into 12 rows. The rows are organized first by speculation type (conservative or aggressive; see Section 2.2.3), then by the sort of scope used for passing the history of branch prediction states (total or procedure-only; see Section 3.2.1), and then by the sort of default branch prediction scheme used (take-all, take-back, table; see Section 3.1). For each row we give the parallelism measure for the four different dynamic branch prediction schemes and the static branch prediction scheme.

For the static schemes, no jump prediction was used in conjunction with the take-all and take-back methods, while table-based jump prediction was used along with the table-based branch prediction. Dynamic jump prediction (always predicting the previous destination) is used with all the dynamic branch prediction schemes. The default jump prediction is none if the default branch prediction is take-all or take-backward, and from the table if the default branch prediction is from the table.

Let us first examine the overall success of speculation. Table 7 compares the parallelism results for no speculation, infinite speculation, and the oracle machine. For the infinite speculation we take the maximum of all entries in the table for each benchmark (Tables 2 – 6). From Table 7 it is clear that speculation results in substantial increases in parallelism over the 0-level speculation results (6.93 to 800-fold increase). However, as expected, the parallelism with speculation falls short of the parallelism for the oracle machine (the oracle has 1.19 to 35.8-fold increase over the best infinite speculation). This is particularly evident for the first four benchmarks which exhibit irregular control flow.

Speculation Type	Scope	Default Pred.	Dynamic				Static
			1-bit	2-bit			
				Type 1	Type 2	Type 3	
Conservative	Total	Take All	12.6	13.1	12.7	13.1	8.55
		Take Back	10.8	10.8	13.2	10.8	9.43
		Table	25.8	26.9	12.7	25.6	25.5
	Proc. Only	Take All	12.6	13.1	12.6	13.1	N/A
		Take Back	10.8	10.8	13.2	10.8	
		Table	25.8	26.1	12.7	25.8	
Aggressive	Total	Take All	19.5	19.5	19.1	19.5	12.1
		Take Back	15.8	15.8	19.8	15.8	11.6
		Table	35.9	36.0	20.1	36.0	36.8
	Proc. Only	Take All	19.3	19.3	19.0	19.3	N/A
		Take Back	15.8	15.8	19.8	15.8	
		Table	36.3	36.5	19.9	36.5	

Table 2: Tex

Speculation Type	Scope	Default Pred.	Dynamic				Static
			1-bit	2-bit			
				Type 1	Type 2	Type 3	
Conservative	Total	Take All	68.3	68.3	68.3	68.3	67.9
		Take Back	69.1	69.1	68.3	69.1	69.1
		Table	134	134	68.3	134	134
	Proc. Only	Take All	69.3	68.3	68.3	68.3	N/A
		Take Back	69.1	69.1	68.3	69.1	
		Table	134	134	68.3	134	
Aggressive	Total	Take All	70.5	70.5	70.5	70.5	316
		Take Back	1192	1192	70.5	1192	1191
		Table	141	141	70.5	141	141
	Proc. Only	Take All	70.5	70.5	70.5	70.5	N/A
		Take Back	1192	1192	70.5	1192	
		Table	141	141	70.5	141	

Table 3: Speech

Given that speculation seems like a beneficial idea, let us now examine the data to determine which kind of speculation leads to the best parallelism results. First, compare the results for the conservative models versus the results for the agressive models. It is clear by examining the tables that in all cases the aggressive results are better than the correspoding conservative results. This is to be expected. In order to summarize the benefit we give a comparison of the best conservative result and the best aggressive result for each benchmark in Table 8. Note the added benefit of the aggressive approach is quite significant, particularly for the speech and eqntott benchmarks.

Now compare the results for the models using the total scope for passing histories of branch predictions relative to the equivalent models that consider on the procedure level scope. Comparing the entries for the total and procedure scopes in Tables 2–6, it is quite clear that there is very little difference between the results, both strategies seem to work equally well. This suggests that there is little correlation between branch histories in different levels of a recursive procedure.

Given that aggressive speculation is significantly better than conservative speculation, and that

Speculation Type	Scope	Default Pred.	Dynamic 1-bit	Dynamic 2-bit Type 1	Dynamic 2-bit Type 2	Dynamic 2-bit Type 3	Static
Conservative	Total	Take All	9.21	9.24	8.32	9.24	7.44
		Take Back	12.7	12.7	9.27	12.7	11.7
		Table	19.7	19.7	8.28	19.7	19.8
	Proc. Only	Take All	8.14	8.27	8.32	9.25	N/A
		Take Back	12.7	12.8	9.27	12.7	
		Table	19.8	19.8	8.29	19.8	
Aggressive	Total	Take All	12.9	13.0	12.7	13.0	7.66
		Take Back	35.6	35.6	13.0	35.6	33.3
		Table	52.9	52.9	13.0	52.9	53.0
	Proc. Only	Take All	12.9	13.0	12.7	13.0	N/A
		Take Back	35.6	35.6	13.0	35.6	
		Table	53.0	53.0	13.0	52.9	

Table 4: espresso

Speculation Type	Scope	Default Pred.	Dynamic 1-bit	Dynamic 2-bit Type 1	Dynamic 2-bit Type 2	Dynamic 2-bit Type 3	Static
Conservative	Total	Take All	13.8	13.8	13.8	13.8	4.64
		Take Back	25.4	25.4	14.8	25.4	6.81
		Table	307	307	14.8	307	306
	Proc. Only	Take All	13.8	13.8	13.8	13.8	N/A
		Take Back	25.4	25.4	14.8	25.4	
		Table	307	307	14.8	307	
Aggressive	Total	Take All	1124	1124	1123	1124	236
		Take Back	2624	2624	1125	2624	1721
		Table	1149	1149	1125	1149	1149
	Proc. Only	Take All	1124	1124	1123	1124	N/A
		Take Back	2625	2625	1125	2625	
		Table	1149	1149	1125	1149	

Table 5: eqntott

total scope is basically the same as procedure-only scope, we now focus on the effect of different branch prediction schemes for the aggressive/total cases (rows 7-9 in the tables).

First let us determine the best dynamic scheme. In all benchmarks the 1-bit scheme performs about the same as the best 2-bit scheme. Within the 2-bit schemes type 1 and type 3 are the best, type 2 doesn't perform as well. We suspect that this is because the type 2 scheme is more biased toward taking branches (the type 2 scheme takes branches in 3/4 of the states, while the other two schemes take branches for 2/4 of the cases). It is likely that this bias towards taking branches does not work so well for non-numeric programs.

Now let us compare the static strategies versus the dynamic strategies. In table 9 we give the comparison between the best dyanmic strategy and the best static strategy. In most cases the best static prediction is as good as the best dynamic prediction. Only in the case of eqntott is the dynamic strategy significantly better.

Speculation Type	Scope	Default Pred.	Dynamic				Static
			1-bit	2-bit			
				Type 1	Type 2	Type 3	
Conservative	Total	Take All	1659	1683	1683	2054	1463
		Take Back	2283	2274	2054	2283	2274
		Table	2162	2269	1683	2269	2043
	Proc. Only	Take All	1659	1683	1683	2054	N/A
		Take Back	2283	2283	2054	2283	
		Table	2162	2269	1683	2269	
Aggressive	Total	Take All	2120	2120	2015	2120	4114
		Take Back	4166	4166	2120	4166	4166
		Table	4024	4024	2069	4024	3363
	Proc. Only	Take All	2120	2120	2069	2120	N/A
		Take Back	4166	4166	2120	4166	
		Table	4024	4024	2069	4024	

Table 6: fpppp

	0-Level Speculator	Infinite Speculation	Infinite/ 0-level	Oracle	Oracle/ Infinite
tex	5.31	36.8	6.93	189	5.14
espresso	2.81	53.0	18.9	1363	25.7
speech	45.4	1192	26.3	7991	6.7
eqntott	3.28	2625	800	28661	35.8
fpppp	1178	4166	3.54	4974	1.19

Table 7: Speculation Success

This could indicate that with this model of parallel execution, the branch histories are much shorter to begin with. Also, the patterns that enable a dynamic predictor to "learn" aren't as strong. Patterns that are established on a sequential machine by many repetitions of the same branch instruction won't appear on a parallel machine if those repetitions are not control-dependent on one another. If branches that are related by control-dependence are essentially random, then dynamic predictions will tend toward the most frequently-occurring branch outcome. In other words, the dynamic predictor will tend to make the same prediction as a table-based static predictor.

However, we must also consider that our static estimates are based on an execution history for the same data, and thus give only an upper bound on how well the static methods can perform. Further experimentation will be required to determine how well the static methods perform when the branch

	best conservative	best aggressive	conservative/ aggressive	aggressive / conservative
tex	26.9	36.8	0.73	1.37
espresso	19.8	53.0	0.37	2.68
speech	134	1192	0.11	8.90
eqntott	307	2625	0.12	8.55
fppp	2283	4166	0.55	1.82

Table 8: Conservative versus Aggressive

	best static	best dynamic	static/ dynamic
tex	36.8	36.5	1.01
espresso	53.0	53.0	1.00
speech	1191	1192	1.00
eqntott	1721	2625	0.66
fppp	4166	4166	1.00

Table 9: Summary of Static Prediction versus Dynamic Prediction

		tex	espresso	speech	eqntott	fpppp
Dynamic (Type 1)	Take Back	54.5%	76.9%	50.6%	91.5%	51.0%
		15.8	35.6	**1192**	**2624**	**4166**
	Table	83.2%	87.1%	83.1%	92.9%	88.1%
		36.0	**52.9**	141	1149	4024
Static	Take Back	52.1%	68.0%	50.6%	68.3%	50.9%
		11.6	33.3	**1191**	**1721**	**4166**
	Table	84.3%	87.1%	85.0%	92.9%	88.1%
		36.8	**53.0**	141	1149	3363

Table 10: Summary of Branch Prediction Success Rate and Parallelism

prediction estimates come from executions on different data sets.

Now let us look at the branch prediction success rate and the resulting parallelism. In Table 10 we give the results for four different branch prediction schemes, two dynamic schemes and two static schemes. For each scheme we give the percent of branches correctly predicted, and the parallelism result. The entries in bold give the winning dynamic strategy, and the winning static strategy. Note that for tex and espresso, the table mechanism is superior to the take back mechanism. Both of these applications have relatively low parallelism, and the branch predicition mechanism with a higher success rate seems to do better. For speech, eqntott and fpppp, the take back mechanism is better even though it has a much lower branch prediction success rate. This is particularly noticable in the case of speech and fpppp. For example, in the speech benchmark, the take back mechanism gives only a 50.6% branch prediction success, but led to an impressive parallelism of 1192. In contrast, the table mechanism has a much higher success rate of 83.1%, but results in a speedup of only 141.

This illustrates that a better branch prediction rate does not necessarily give better parallelism results. It seems that for some branches in these cases, while the branch is usually taken one way, the less-frequent destination is on a critical path, and shortening that critical path increases parallelism.

A few additional experiments were performed with infinite speculation. In the previous set of tests, we compared take-back and table-based static prediction. But the former experiment also included jump prediction, which was not in the latter experiment. To separate these two effects, we ran a single experiment with static take-back branch prediction and table-based jump prediction. The results were virtually identical to take-back branch prediction without jump prediction, showing that for those cases where table-based results were higher, the improvement was due to the table-based branch prediction, *not* the addition of jump prediction.

We also ran some experiments to measure the effects of branch correlation, discussed at the end of Section 3.2.1. We found that branch correlation had virtually no impact on parallelism or on prediction rates (what little impact it had was usually negative), so we didn't persue this path any further.

4.4 Finite Speculation

So far, comparisons of prediction strategies have been based on the values obtained from infinite speculation. But this may not be the best way to compare prediction techniques, since infinite speculation would be impossible to implement. It is possible that one algorithm is good for deep speculation, while another is better when speculation is more limted. Therefore, we ran some experiments where speculation only went to a specified level.

We chose the four prediction algorithms listed in Table 10 since, based on their previous results, they seemed to be representive of the different classes of algorithms. A summary of the experimental results are listed in Table 11. We found that, in 4 of the 5 cases, the results for infinite speculation were an accurate predictor of the relative performances of the finite speculation. That is, the best branch predication mechanism for the infinite case, was also the best branch prediction mechanism for the finite cases. Therefore, in those cases, we only listed the results for the best speculation algorithm. There were anomolous results for finite speculation of eqntott, so three algorithms are listed for that case.

Each entry in Table 11 consists of two numbers. The upper number gives the parallelism measured for the specified model, while the bottom number gives the percent measurement as compared to the parallelism for infinite speculation depth. For example, for the tex benchmark using a static table branch predictor, and a speculation depth of 1, we get a parallelism measurement of 18.2 which is 49.5% of the parallelism measurement for infinite depth speculation.

We note that for all benchmarks but eqntott, a speculation depth of 8 is sufficient to get a large percentage of the parallelism availabe with infinite depth. In the case of eqntott, even a speculation depth 256 gives less than 50% of the infinite speculation depth. For the case of eqntott we can also notice that even though the static take back scheme was better for the infinite depth speculation, the static table scheme is better for smaller speculation depths.

Table 12 gives the speculation depth required to achieve, 10, 20, 50, and 100 percent of the parallelism acheived by infinite depth speculation. To acheive 10% of the infinite case, four of the five

		Speculation Depth										
		0	1	2	4	8	16	32	64	128	256	∞
tex	static,	5.31	18.2	27.6	35.0	36.7	36.8	-	-	-	-	36.8
	table	*14.4%*	*49.5%*	*75.0%*	*95.1%*	*99.7%*	*100%*					*100%*
espresso	static,	2.81	8.19	14.1	21.8	45.2	49.8	50.9	51.3	52.3	52.4	53.0
	table	*5.30%*	*15.5%*	*26.6%*	*41.1%*	*85.3%*	*94.0%*	*96.0%*	*96.8%*	*98.7%*	*98.9%*	*100%*
speech	static,	45.4	325	332	402	876	1191	-	-	-	-	1191
	take back	*3.81%*	*27.3%*	*27.9%*	*33.8%*	*75.6%*	*100%*					*100%*
eqntott	static,	3.28	8.27	15.5	29.0	55.1	105	195	350	570	750	1721
	take back	*0.19%*	*0.48%*	*0.90%*	*1.69%*	*3.20%*	*6.10%*	*11.3%*	*20.3%*	*33.1%*	*43.6%*	*100%*
	static,	3.28	12.4	21.2	56.2	146	296	529	757	770	790	1149
	table	*0.28%*	*1.07%*	*1.85%*	*4.89%*	*12.7%*	*25.8%*	*46.0%*	*65.9%*	*67.0%*	*68.8%*	*100%*
	dynamic,	3.28	12.5	21.4	56.9	147	302	541	822	872	933	2624
	take back	*0.13%*	*0.48%*	*0.82%*	*2.17%*	*5.60%*	*11.5%*	*20.6%*	*31.3%*	*33.2%*	*35.6%*	*100%*
fpppp	static,	1178	1695	3137	3724	4155	-	-	-	-	-	4166
	take back	*28.3%*	*40.7%*	*75.3%*	*89.4%*	*99.7%*						*100%*

Table 11: Summary of Speculation Depth and Parallelism

	tex	espresso	speech	eqntott	fpppp
10%	0	1	1	16	0
50%	1	5	5	> 256	2
90%	3	10	10	$\gg 256$	4
100%	8	> 256	11	$\gg 256$	10

Table 12: Speculation Depth Required for % Parallelism of Infinite Depth

benchmarks (tex, espresso,speech, and fpppp) needed only 0 or 1 levels of speculation. However, the benchmark eqntott required 16 levels of speculation to achieve even 10%. Similar results hold for higher percentages. For tex, espresso, speech, and fppp, a maximum of 5 levels gives 50%, and a maximum of 10 levels gives 90%. This indicates that for many benchmarks an acceptable portion of the benefit of speculation can be realized with about 10 levels of speculation. However, as illustrated by the eqntott benchmark, some programs require much deeper speculation to acheive even a small portion of the potential benefit.

5 Related Work

Most studies on the limits of instruction-level parallelism have taken one of two approaches. One approach is to analyze the selected benchmark at either the source-code or object-code level, usually by executing the program with a special interpreter that is based on a certain parallel machine model [7, 18, 11, 8]. The other approach [9, 19, 16], also used in our study, is to schedule machine instructions from a trace generated by an actual execution of the object code.

Wall [19] developed a series of models representing various levels of optimism about what the hardware and compiler could do. Using trace driven simulation, he analyzed benchmarks (SPEC programs plus some real programs) and found that speculative execution is an important factor in exploiting the parallelism and achieving its limit. Recently, Wilson and Lam reported their study of the way control flow limits parallelism. They demonstrated that substantially higher parallelism can be achieved by peforming control-dependence analysis, executing multiple flows of control simultaneously, and speculative execution [9]. Theobald, Gao and Hendren have also reported their study of instruction-level parallelism and its smoothability. Their study also found the potential for much improvement using speculation.

None of these previouse studies studied the interaction of speculation and branch prediction, as we did in this paper. Instead, the speculation in each of these studies corresonds to an isolated point in our set of models. Wall's study did not emphasize the role of multiple flow of control, due to a limited scheduling window. In terms of level speculation, it appears that his speculation corresponded to our infinite aggressive model. In terms of branch prediction, he assumes the dynamic scheme using a 2-bit state for each branch address, equivalent to our Type 1.

Lam and Wilson included control depedence and multiple flow of control as an important part of their study. However, in terms of speculative execution under multiple flow of contrl, their study appears focused on only two models: their CD-MF model, corresponding to our 0-Level Speculator, and their SP-CD-MF model, corresponding to our infinite aggressive speculation. their experiments used static branch prediction, based on profile information collected from running the same benchmark programs with the same inputs used in the simulation (i.e., static table-based prediction).

All three studies used a model in which a machine speculates down only one branch. Riseman and Foster [13] studied a model in which the machine could execute down both paths through a finite number of levels. They found that up to a point, potential parallelism was roughly proportional to the square root of the number of levels.

6 Conclusions

In this paper, we defined a family of parallel models for speculative execution based on the notion of dynamic control dependence (DCDT). Based on these models we presented a comprehensive experimental study of the limit of parallelism for a number of benchmark programs assuming a variety speculative execution models and different branch prediction strategies.

Our results provide strong evidence that aggressive speculative execution is required to achieve reasonable upper bounds on parallelism, particularly for non-numerical applications.

In studying the interaction between branch prediction and parallelism for the speculative models, we found that the dynamic and static schemes gave similiar results, and that a default prediction based on the take back and table schemes worked the best. Somewhat suprisingly, we found that branch prediction success rate did not correlate with parallelism results. In fact, in some cases a scheme with a much lower success rate gave much better parallelism results. In addition, we found that the parallelism measurements were relatively insensitive to: (1) the scope of branch prediction, (2) the method used for jump prediction, and (3) branch correlation.

On the positive side, our experiments indicate that even though concurrent speculation on multiple flow of control is essential, speculation down only one path (destination) per branch execution seems to be enough to achieve a reasonable degree of parallelism. Furthermore, for the majority of programs we studied, the speculation depth needed to achieve a high portion of available parallelism is bounded by a small number (around 10).

References

[1] T. M. Austin and G. S. Sohi. Dynamic dependence analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 1992.

[2] M. Butler, T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, Toronto, Ontario, May 1991.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single-assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[5] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Massachusetts, October 1992.

[6] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, Massachusetts, April 1989.

[7] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[8] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, C-37(9):1088–1098, September 1988.

[9] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992.

[10] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[11] 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, November 1984.

[12] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, October 1992.

[13] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.

[14] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, May 1981.

[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, Seattle, Washington, May 1990.

[16] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Portland, Oregon, December 1992.

[17] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. ACAPS Technical Memo 40, School of Computer Science, McGill University, Montréal, Québec, June 1992.

[18] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.

[19] 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, Santa Clara, California, April 1991.

A Implementation Details

Since a sequential trace of a program is equivalent to a preorder depth-first traversal of the corresponding DCDT (see Section 2.1), and since a block is only dynamically control-dependent on its ancestors in the DCDT, it is only necessary to keep a *stack* of basic block identities, and the number of the parallel instruction in which each block's branch is executed. This represents the time at which a block is resolved, so we use the term "resolution time" to refer to the number of the parallel instruction containing the branch.

When a new block is encountered in the trace, SITA uses the D function to find the set of blocks on which the new block is control-dependent, and then pops blocks off the stack until it finds a block which is in that set. (The blocks which were popped off the stack represent parts of the DCDT which have already been traversed, and hence don't need to be saved anymore.) SITA then looks at the resolution time of the block at the new top of stack, and makes it a "floor" for scheduling the current basic block: all operations in the current block must be scheduled after that time. Of course, the floor is only one constraint; each operation in the new block may be further delayed by unsatisfied data dependencies.

Like 0-Level Speculation, Inifininte-Level Speculation is simulated by maintaining a stack with both the identity of each basic block and its resolution time. In addition, if dynamic prediction is used, then each entry in the stack contains the current branch history state.

When a new block is encountered in the trace, SITA pops blocks off the stack until it finds a block which is in the set defined by D. SITA then looks back through the stack until finding the first mis-predicted branch or jump, and uses the resolution time of that branch or jump as the floor for scheduling the current basic block.

When SITA encounters a branch or jump, it predicts the jump according to the selected algorithm. If prediction is static, the appropriate method is applied and the block at the top of the stack is either marked "predicted" or "mis-predicted." If prediction is dynamic, SITA looks back through the stack to find the most recent branch or jump with the same address (which is an ancestor in the DCDT) and copies the history state into the new top of stack, updating it according to the algorithm chosen. The current block is either marked "predicted" or "mis-predicted," and its resolution time is saved. If the previous block on the stack (representing the parent in the DCD) was resolved *after* the current block, and the speculation is conservative, then the new resolution time recorded is the same as the one in the previous block.

The method of determining the floor must be modified if speculation is only to a depth of n. After popping back to the appropriate block in the stack, SITA then looks back through the stack until finding the first mis-predicted branch. So far, this is the same as infinite speculation. But whereas with infinite speculation, the resolution time of that mis-predicted branch becomes the "floor" for scheduling instructions in the current basic block, with *finite* speculation the current block may have to be scheduled later. This is because there may be many unresolved branches between the current block and the mis-predicted branch, more than the speculation limit. Therefore, if n-level aggressive speculation is used, the floor for the current basic block is the minumum of

1. The resolution time of the mis-predicted branch, and
2. The resolution time of the $(n+1)^{st}$-most-recently resolved block in the stack. (1 is added because 0-Level Speculation corresponds to the resolution of the most-recently-resolved block.)