Submitted to the 24th Int. Symp. on Comp. Arch.

Selective Dual Path Execution

Timothy H. Heil, James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI 53706

Abstract

Selective Dual Path Execution (SDPE) reduces branch misprediction penalties by selectively forking a second path and executing instructions from both paths following a conditional branch instruction. SDPE restricts the number of simultaneously executed paths to two, and uses a branch prediction confidence mechanism to fork selectively only for branches that are more likely to be mispredicted. A branch forking policy defines the behavior of SDPE when a low confidence branch prediction is encountered while two paths are already being executed.

Trace driven simulation is used to evaluate the effectiveness of SDPE with three different forking policies. SDPE can reduce the cycles lost to branch mispredictions by 34 to 50%, resulting in an approximate 10% reduction in overall execution time. However, it is shown that both branch mispredictions and low confidence predictions tend to occur in clusters, limiting the effectiveness of SDPE.

A number of design parameters are studied via simulation. These include prediction and confidence table sizes. Finally, a number of implementation issues are discussed, with emphasis on instruction fetching mechanisms and register renaming.

1.0 Introduction

Conditional branch instructions disrupt smooth pipeline flow and are widely known to pose performance problems in modern microprocessors. The method of choice for dealing with conditional branches is to predict their outcomes and speculatively execute instructions down the predicted path. If the prediction is correct, useful work is done, and performance losses are minimized. If the branch prediction is incorrect, however, cycles are wasted on instructions that must be discarded. And there may be an additional penalty for restarting instruction fetches down the correct branch path.

Another approach that is sometimes suggested, but often dismissed, is to fork a second execution path and execute instructions down both paths following a conditional branch. When the conditional branch is eventually resolved, one of the two paths is valid and the

other must be discarded. Fig. 1 illustrates conventional branch prediction and multiple path execution.

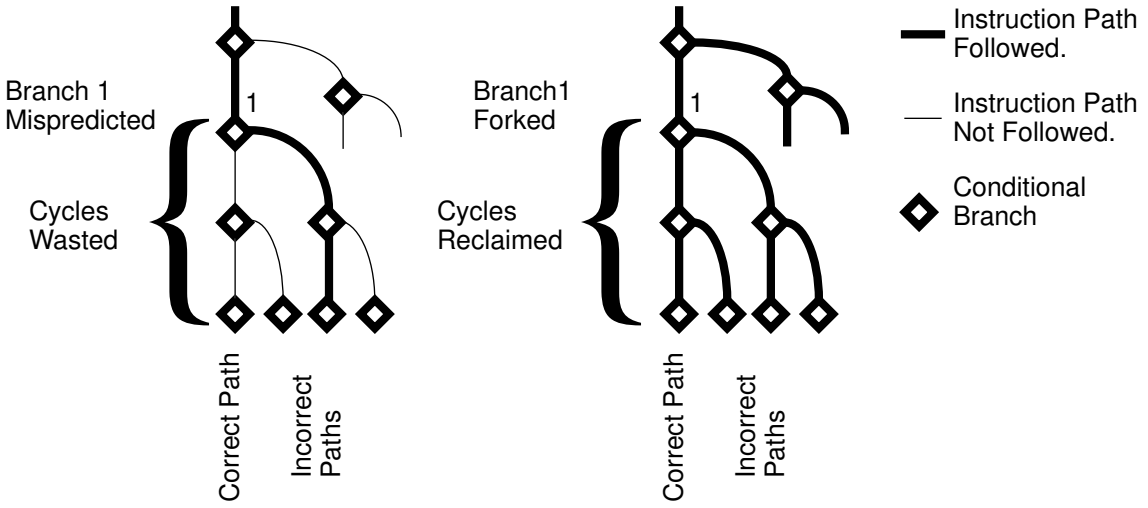


FIGURE 1. Multiple path execution eliminates branch misprediction penalties. Branch 1 is mispredicted, typically resulting wasted cycles. However, multiple path execution reclaims these cycles by following both paths after each branch simultaneously.

The perceived problem with this approach is that it implies exponential hardware growth because each branch path may also contain a conditional branch which leads to another doubling of paths. In modern processors it is not unusual to have four or five pending conditional branches at any given time -- this would suggest tens of simultaneous execution paths, and this is too many to be practical.

We consider a more limited approach where hardware resources are provided for at most two simultaneous execution paths. A second execution path is selectively forked for only certain conditional branches, not all. Because there are sufficient resources for only one additional execution path at any given time, the additional path should be chosen very carefully. In particular, a second path should be forked for branch predictions that have the greatest likelihood of being incorrect; i.e. where the payoff is likely to be highest. This process is referred to as Selective Dual Path Execution (SDPE).

An important underlying technology is a mechanism for determining the accuracy of conditional branch predictions. For SDPE it is sufficient to divide conditional branch predictions into two sets: a high confidence set and a low confidence set. A hardware-generated high/low confidence signal can be used to trigger the forking of a second execution path when the confidence in the branch prediction is deemed to be low. Recently developed branch confidence hardware will be used for this purpose and is described in the Section 2.

1.1 Prior Work

Following both paths after every branch is known as Eager Execution. Eager Execution requires following many tens of paths and is studied using an ideal model in [16]. Disjoint Eager Execution (DEE) [13] executes the path that is the most likely to be correct out of

all unexecuted possible paths. This involves calculating the probability that all preceding branches have been correctly predicted. DEE first follows the predicted paths, speculating past branches. Each branch prediction has some probability of being incorrect, and eventually the accumulated probability that the current path is correct becomes small. DEE then returns to the first unresolved branch and begins following not-predicted paths. DEE can execute multiple paths simultaneously, as long as hardware resources are available. The implementation studied uses a single fixed prediction probability for all branch predictions, resulting in paths being executed in a fixed pattern.

As part of an instruction level parallelism study, Wall considers *branch fanout* [15], which also executes both paths following conditional branches. To avoid an exponential increase in the number of paths, a fanout limit is placed on the number of paths that are simultaneously executed. The branches chosen for fanout are selected using a static prediction probability assigned through program profiling. Branch fanout is studied with and without branch prediction up to a fanout limit of four.

Going back in history, the IBM 3168 and 3033 mainframes could fetch instructions from both paths following a conditional branch, though instructions from only one path could be decoded and executed [2]. The IBM 3168 could fetch down two paths simultaneously, and the IBM 3033 could fetch instructions from three paths.

Current processors also fetch instructions from multiple paths in limited ways. For example, the MIPS R10000 requires a one cycle delay to decode and predict branches. This cycle is used to fetch instructions sequentially following the branch. If the branch is predicted taken, the extra fetched instructions are stored in a Resume Cache [5] in a partially decoded state. If the branch is discovered to be not taken, then the sequential instructions are quickly recovered from the Resume Cache, eliminating one cycle from the misprediction penalty. The IBM POWER1 processor statically predicts all conditional branches not taken. However, the processor fetches some instructions from the taken path as a hedge, which results in a single cycle penalty when the branch is taken [17].

1.2 Paper Summary

The rest of the paper is divided into the following sections. Section 2 contains a description of the branch confidence mechanism and describes the forking policies studied. Section 3 describes the trace driven model and gives simulation results. Section 4 describes implementation issues and alternatives. Section 5 concludes the paper.

2.0 Underlying Hardware Mechanisms

To support SDPE, two underlying mechanisms must be considered. The first is a hardware device for separating high and low confidence branch predictions, and the second is a policy to be followed when a second low confidence prediction is encountered and there are already two paths being executed.

2.1 Branch Confidence Mechanisms

A branch confidence mechanism sorts conditional branch predictions into low and high confidence sets based on previous predictability. Branch confidence mechanisms are studied in depth in [6]. Here we provide a description of a mechanism that works effectively and has a practical implementation. This confidence mechanism is used by the study in Section 3.

The confidence mechanism (Fig. 2) consists of a table of resetting m-bit counters, and is indexed by the XOR of a truncated program counter and a global branch history register (GBHR). The program counter points to a conditional branch instruction that is being predicted and the GBHR indirectly indicates the control path followed before arriving at the branch in question. For a table of 2^n entries, n-low order program counter bits are XORed with an n-bit GBHR; this is essentially the same mechanism used by the gshare branch predictor [8].

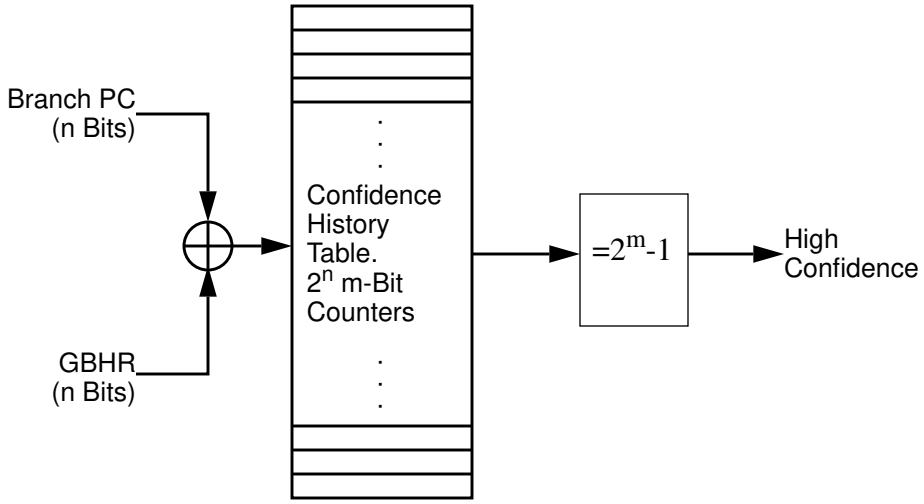


FIGURE 2. The Resetting Counter Confidence Mechanism consists of an array of saturating counters indexed by the branch PC XORed with a Global Branch History Register.

For a given conditional branch, the counter at the confidence table entry is incremented if the branch is predicted correctly, up to a maximum counter value (seven in the implementation used in this paper). The counter is reset to zero when the branch is mispredicted. If the counter is less than its maximum value, the prediction is considered to be a low confidence one; if it is equal to its maximum value, the prediction is assigned high confidence.

In [10] it is shown that this algorithm works nearly as well as more complex algorithms that keep exact histories of branch outcomes. The resetting counters work well because a mispredicted branch is more likely to be mispredicted in the near future (see section 3.3). With a 3-bit resetting counter, a branch is considered low confidence for the next six occurrences following a misprediction.

Ideally, the set of low confidence branch predictions would exactly equal the set of all mispredictions. Of course, this can not be implemented in practice (or we could solve the

branch prediction problem completely by reversing all predictions in the low confidence set!). Rather, we try to concentrate a large number of the mispredictions into the low confidence set -- realizing that some will be missed and some correct predictions will be included.

A current superscalar processor can have approximately five conditional branches pending at any given time, and if only one of the five can fork a second path, intuition suggests a low confidence set that contains about 20% of all branch predictions. For our selected benchmark set and branch predictor (described in Section 3), a branch confidence table with 3-bit resetting counters identifies 20% of the branch predictions as low confidence, and these contain 75% of all mispredictions.

2.2 Branch Forking Policies

The second underlying mechanism is a policy to be followed when a low confidence prediction is encountered while two branch paths are already being executed. Following subsections describe some alternatives.

2.2.1 Canceled Path Policy

Canceled Path (CP) is the simplest policy. Low confidence branches encountered while a second path is already being executed are simply ignored. Fig. 3 illustrates the execution of a hypothetical instruction stream using the CP policy.

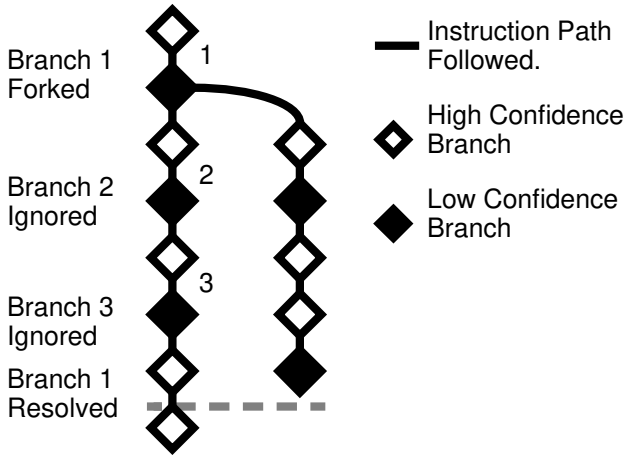


FIGURE 3. The Canceled Path forking policy does not fork branches when two paths are already being executed. Predictions for Branches 1, 2, and 3 are all in the low confidence set, but only Branch 1 can be forked.

The processor predicts conditional branches until Branch 1 is encountered. This branch is deemed low confidence by the confidence mechanism, so the processor forks a second path. Before Branch 1 is resolved, however, Branches 2 and 3 are also predicted with low confidence. Because a second path is already in progress, neither of these two branches forks a second path. If Branches 2 or 3 are mispredicted, the full penalty for their misprediction will be paid.

2.2.2 First Delayed Policy

With the First Delayed (FD) policy, when a second low confidence prediction is encountered, the processor state at that point is saved. Then, after the preceding low confidence branch is resolved, the processor can fork a second thread beginning at the saved state. Fig. 4 illustrates the FD policy. Branch 1 is predicted with low confidence and a second path is forked. Branch 2 is also predicted with low confidence. Because a second path was

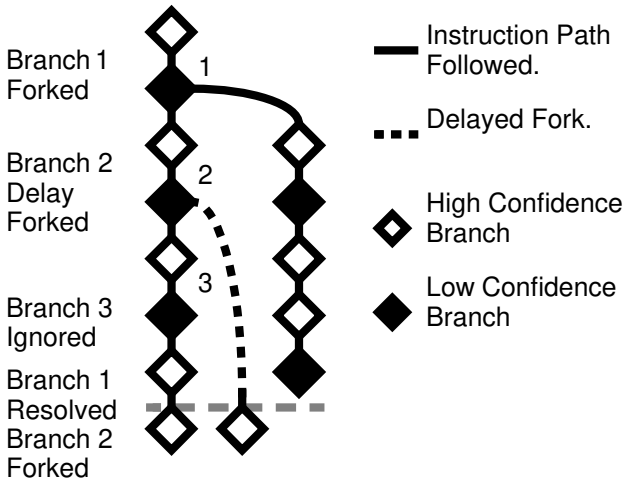


FIGURE 4. The First Delayed forking policy records the processor state for the first branch which should be forked while two paths are being executed. Branch 2 is forked when Branch 1 is resolved.

forked by Branch 1, a fork following Branch 2 is delayed until Branch 1 is resolved. If Branch 2 is mispredicted, the cycles between encountering Branch 2 and resolving Branch 1 are wasted. In the figure, Branch 3 is also predicted with low confidence, but can not be forked, because Branch 2 has already been selected to be delay forked. If Branch 3 is mispredicted, it will suffer a full misprediction penalty.

In general, a delayed forked path can occur along either path of the preceding forked branch. Such delayed forks on the second path are not shown in Fig. 4 for simplicity. Once the current branch is resolved, it is known which delayed branch is valid. This means state for two delayed-forks must be kept.

2.2.3 Last Delayed Policy

The Last Delayed (LD) policy is related to the First Delayed policy. However, instead of saving the state when the first low confidence branch cannot be forked, the processor saves the state when the latest low confidence branch is encountered and cannot be forked due to a second path already being executed. In other words, while running in dual path mode, the saved state is overwritten each time a later low confidence branch is encountered.

Fig. 5 illustrates the LD policy. Branch 1 is predicted with low confidence and causes the fork of a second execution path. When Branch 2 is encountered, the processor prepares to

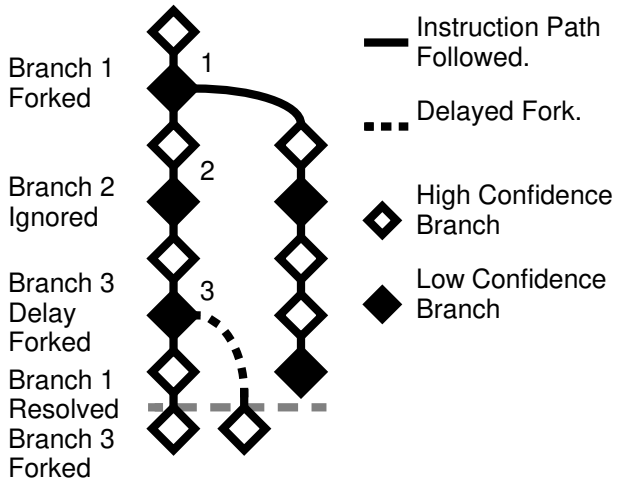


FIGURE 5. The First Delayed forking policy records the processor state for the last branch which should be forked while two paths are being executed. Branch 3 is forked when Branch 1 is resolved.

delay fork this branch and saves state. However, when Branch 3 is encountered later, the processor abandons the delayed fork of Branch 2, and saves state to delay forking after Branch 3. If Branch 2 is correctly predicted and Branch 3 is mispredicted, the cycles between encountering Branch 3 and resolving Branch 1 are wasted. This will be fewer wasted cycles than with the FD policy because the fork following Branch 3 is delayed fewer cycles. However, there is a also a chance that the last delayed branch is on a mispredicted path. For instance, if Branch 2 is mispredicted, the full misprediction penalty will occur.

3.0 Trace Driven Performance Study

A trace-driven simulation was used to study the effectiveness of SDPE. In this section we describe benchmarks used, the simulation model, and performance results.

3.1 Methodology

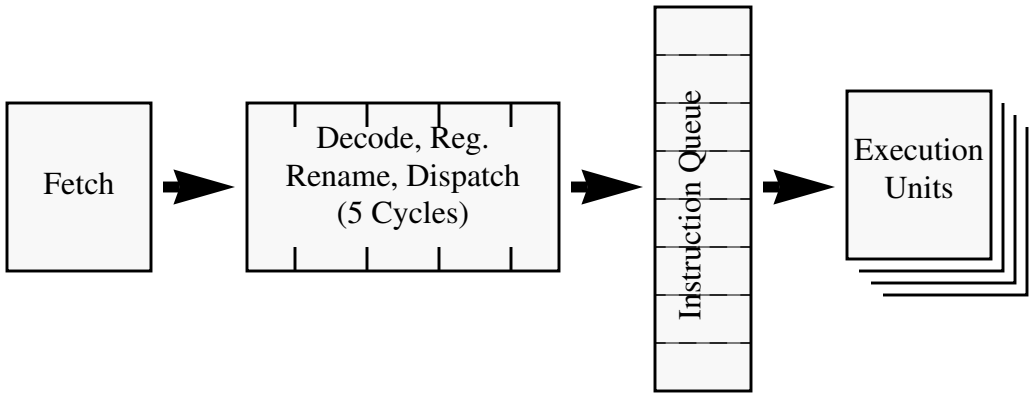
The selected benchmarks are from the Instruction Benchmark Suite[14]. These traces were obtained from a MIPS R2000 processor by physically probing the processor during program execution. IBS traces contain all code executed by the program, including kernel

code from system calls and TLB miss-handlers. The benchmark characteristics are listed in Table 1

TABLE 1. Instruction Benchmark Suite

Benchmark	Instructions (Millions)	Conditional Branches (Millions)	Branch Misprediction Rate (%)
GCC	126	16.4	17.2
SDET	43	4.1	15.1
MPEG	111	9.5	13.6
GROFF	116	12.5	11.2
GS	120	15.3	9.7
VERILOG	52	6.3	9.5
NROFF	135	22.9	5.0
JPEG	104	15.8	3.4
CUMULATIVE	807	102.8	10.6

Fig. 6 shows the processor model, and lists the model characteristics. These parameters are used for all simulations, unless otherwise stated. Instructions are fetched from a trace



- **Superscalar Processor**
  - 8-Way Fetch
  - 4-Way Symmetric Out-Of-Order Issue
  - 32 Entry Instruction Queue
- **L1 I-Cache**
  - 32K
  - 2-Way Set Associative
  - LRU Replacement
  - 16 Word Lines
  - 2-Way Interleaved
  - Prefetches 1 Line
  - 10 Cycle Miss Penalty
- **Branch Prediction**
  - 8K 2-Bit
  - PC XOR Global Branch History (gshare)
  - Speculation Past 7 Branches
  - 7 Cycle Minimum Misprediction Penalty
- **Confidence Mechanism**
  - 8K Entry Resetting Counter
  - PC XOR Global Branch History

FIGURE 6. The trace driven processor model is a superscalar out-of-order processor.

and dispatched into the instruction window. Pipeline stages between fetch and dispatch are simulated by delaying the instruction's entry into the window by the appropriate number of cycles. Dependence analysis is used to schedule the issuing of instructions from the

window into the execution units. Instruction latency determines how many cycles after issue the result of the instruction is available. Realistic instruction latencies are simulated in the model; instruction latencies are shown in Table 2

TABLE 2. Instruction Latencies

Integer Instructions	Latency (Cycles)	Floating Point Instructions	Latency (Cycles)
ALU	1	ADD/SUB	3
LOAD/STORE	2	DIV (Single)	11
BRANCH	1	DIV (Double)	18
MULT	3	MULT	3
DIVIDE	11	LOAD/STORE	2

The only cache simulated is the Level 1 instruction cache (L1 I-cache). All other caches are ideal -- they always hit. The L1 I-cache is simulated because SDPE is involved directly with the instruction fetch mechanism, and SDPE puts additional stress on the instruction cache. The L1 I-cache is interleaved to provide access to two consecutive cache lines each cycle [3]. This allows contiguous blocks of instructions to be fetched across cache line boundaries. Each cycle 32 instructions from two lines are fetched. Of these, up to eight instructions following the current PC are selected. Only one branch prediction and target prediction can be performed each cycle, however, and any control transfer instruction terminates the block of fetched instructions. That is, at most one basic block, up to a maximum of eight instructions, is fetched per cycle.

The L1 I-cache always prefetches the next line after the line containing the current PC. This is done even if the current line is in the cache. On a cache miss, both the current line and the next line are brought into the cache.

Conditional branches are predicted using a 16Kbit gshare [8] predictor. This predictor contains 8K two-bit counters. The array is indexed by XORing the PC of the branch with the Global Branch History Register (GBHR). The GBHR contains the outcomes of the last 13 conditional branches in order. The Confidence mechanism also has 8K entries, each of which is a 3-bit resetting counter as described in Section 2.1 These entries are accessed in the same way as the gshare predictor.

For disambiguation, memory addresses of loads and stores are calculated early, and compared against all other loads and stores. However, values are not forwarded between stores and loads.

Because this is a trace driven model, speculative instruction paths are not available. When a branch is mispredicted, the fetch unit merely stops fetching new instructions from the trace. Once the branch is resolved, instruction fetching continues. If the mispredicted branch is a low confidence one that causes a forked path, the penalty for the misprediction is annulled. New instructions are fetched on the next cycle as normal.

Because only the correct path is available this simulation model cannot simulate interference that will occur between the correct path and the incorrect path. In particular, instruc-

tions from the incorrect path do not use any fetch bandwidth, instruction window space, or issue bandwidth. Cache and predictor pollution effects are also not modeled. These and other deficiencies of our trace-driven simulation are addressed in Section 3.7.

3.2 Initial Results

Fig. 7 shows performance with different branch forking policies. The base case uses only

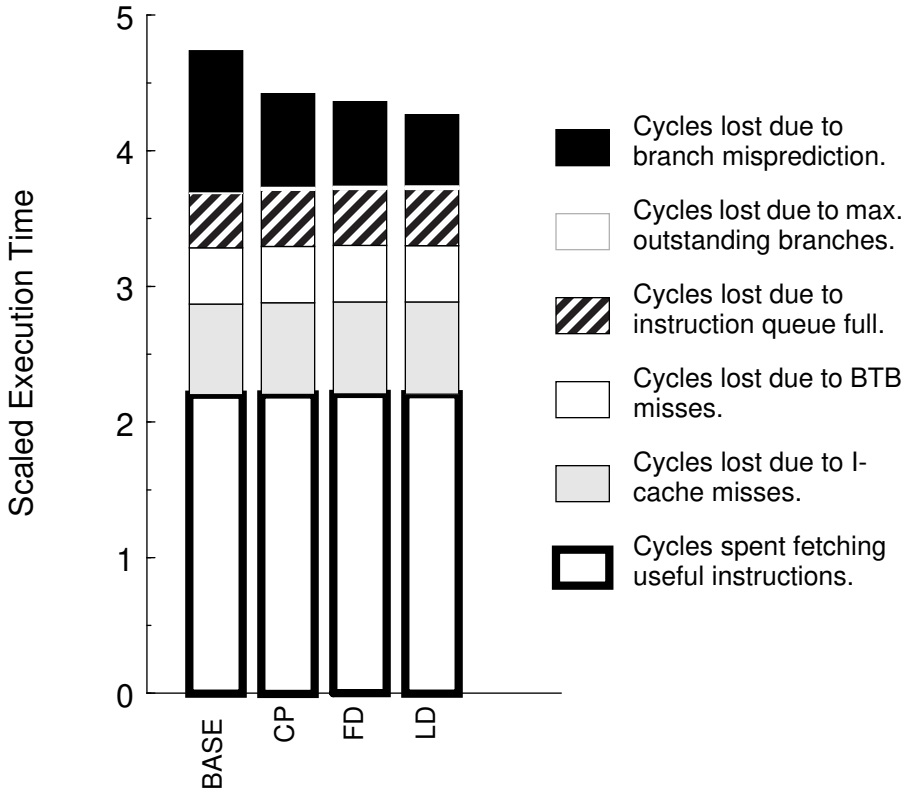


FIGURE 7. Execution times for SDPE using each forking policy. SDPE reduces the cycles lost due to branch misprediction.

branch prediction and speculative execution -- no dual path execution. Three SDPE models are used, the first uses the cancelled path (CP) policy, the second uses first delayed (FD), and the third uses last delayed (LD). All eight benchmarks are simulated to completion. Performance is measured in clock cycles. Cumulative data is calculated by first weighting each benchmark to contain the same number of conditional branches. The scaled results are then summed.

In Fig. 7, execution time is broken down according to the way individual cycles are spent at the fetch unit. Measuring the processor performance at the fetch point is preferred for this study because SDPE focuses on improving branch performance -- a function performed by the fetch unit. Also, it is conceptually easy to discern what the processor is doing at the fetch engine on a cycle-by-cycle basis.

The bottom segment of the bars in Fig. 7 shows the number of cycles spent fetching valid instructions. Every cycle in which at least one useful instruction is fetched from the trace is placed in this category. The next stacked segment shows the number of cycles when no useful instruction is fetched because of I-cache misses. The next segment shows cycles lost due to BTB misses. The cross hatched segment shows the number of cycles where the instruction window is full; this stalls the pipeline, so no new instructions can be fetched. For a small sliver of cycles just above the cross-hatched bar, the maximum number of outstanding conditional branches is reached, and the next instruction is a conditional branch -- this causes instruction fetching to stall. The top bar indicates the cycles lost due to branch misprediction.

As is typical in such measurements, there is some overlap between various categories. That is, some fetch cycles are lost for multiple reasons. For example, because two I-cache lines are fetched each cycle, a miss can occur accessing the second line. When this happens the instructions that are available in the first line are fetched, and this is recorded as a useful cycle. Even though a miss also occurs during the cycle, thus limiting the number of instructions fetched, it is not recorded as a miss cycle.

Returning to Fig. 7, the overall performance improvement is about 10% with the LD policy. However, SDPE is targeted at saving lost cycles due to branch mispredictions (the top segment of the bars in Fig. 7). And branch misprediction is the single largest cause of lost cycles. In the base case, without SDPE, cycles lost due to branch misprediction account for 22% of execution time on average. Because SDPE reduces only branch misprediction cycles, the effectiveness of SDPE is limited to 22%. From this perspective, SDPE does its job well. In particular, cycles lost due to branch misprediction are reduced significantly. The NC policy reduces misprediction cycles by 34%, the FD policy by 41%, and the LD policy by 50%.

Although the LD policy appears to be the best performing policy, this may not be the case in practice. This study simulates the LD policy in a way that tends to be more optimistic than the other policies. A real processor may have to speculate past several branches to reach the branch selected by LD. However, in a trace driven study incorrect paths can not be followed, so during dual path execution a mispredicted branch will be the last instruction fetched until a preceding forking branch is resolved. According to the simulated LD policy, if this branch is predicted with low confidence, a delayed fork will occur at this point. Thus the LD policy optimistically chooses the first mispredicted branch for delayed forking. Because of this optimism, we use the more conservative FD policy in simulation studies to follow.

Since SDPE targets cycles lost due to branch mispredictions, SDPE can be expected to perform better on benchmarks with higher misprediction rates. GCC has the highest misprediction rate of all the benchmarks, and JPEG has the lowest misprediction rate. Fig.

8 shows the results for these two benchmarks. SDPE does very well on GCC, reducing

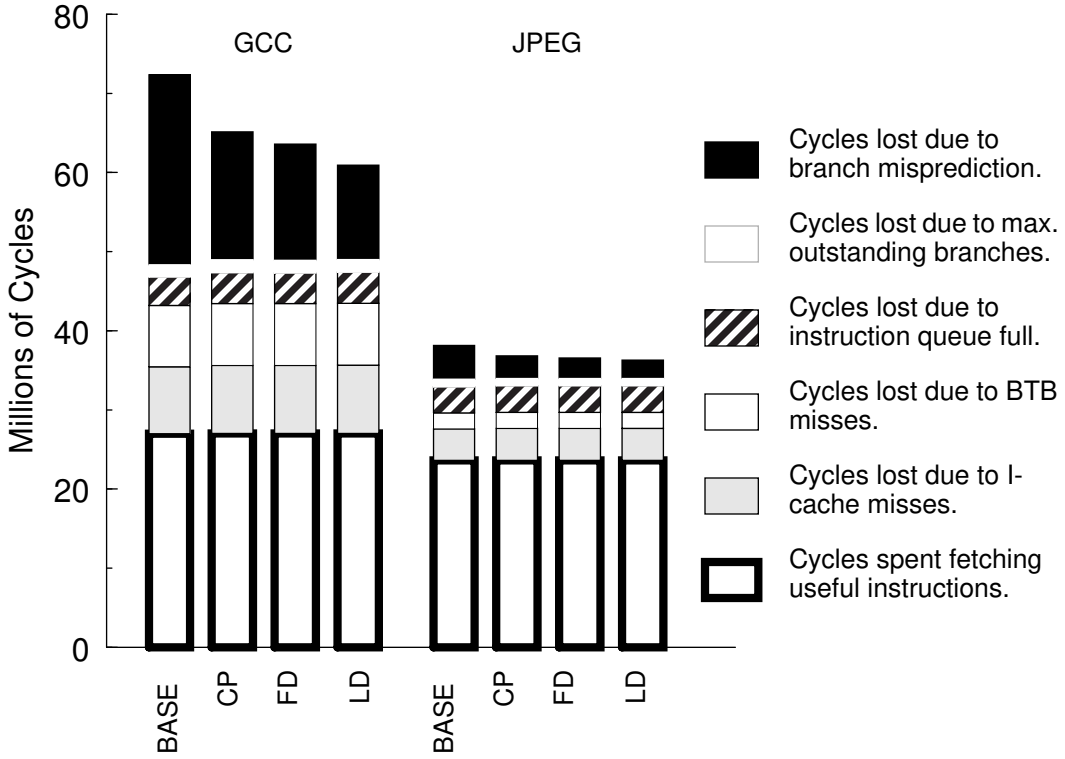


FIGURE 8. Execution times for GCC and JPEG using SDPE. Benchmarks with high branch misprediction rates gain more from SDPE.

execution time by 10.0%, 12.1%, and 15.8% for the CP, FD, and LD policies respectively. At the other extreme, SDPE reduces the execution time of JPEG by 4.9% using LD.

3.3 Limitations on Performance Improvements

During simulations, we measured the accuracy of the confidence mechanism. It is able to isolate 74.8% of the mispredicted branches in the low confidence set. So, ideally SDPE should be able to eliminate 74.8% of the branch misprediction penalty. However, SDPE eliminates at most 50% of the branch misprediction penalty for the LD policy. Two closely related phenomena account for this apparent discrepancy.

First, mispredicted branches tend to come in clusters. This can best be seen by comparing the distribution of distances between mispredicted branches in the benchmarks with a completely random distribution (a geometric distribution). The distance between mispredicted branches is the number of branches that separate them. If a mispredicted branch is immediately followed by a second, the distance is one.

Fig. 9 contains histograms of the distances between mispredicted branches as they occur in the benchmarks and the distances if the mispredictions were geometrically distributed.

This graph shows a high degree of clustering of mispredicted branches in the benchmarks.

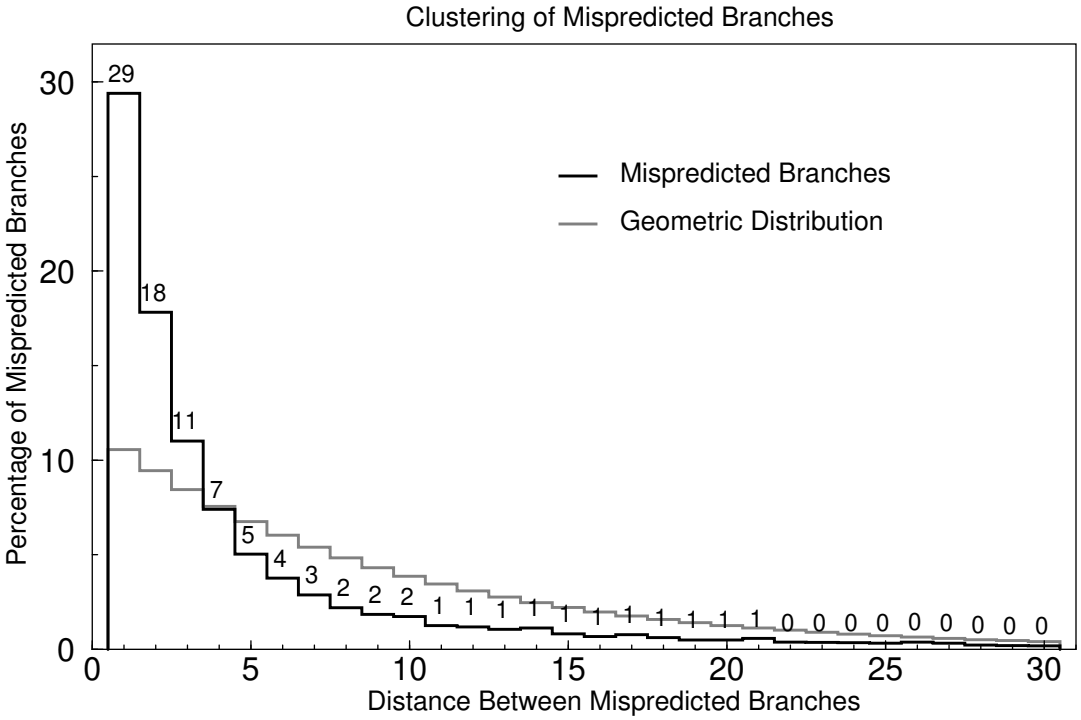


FIGURE 9. Distribution of branch mispredictions. Mispredictions tend to arrive in clusters.

For the benchmarks 10.9% of branches are mispredicted, so with a geometric distribution the percentage of distance one mispredictions is 10.9%. On the other hand, in the benchmark traces, 29% of the mispredicted branches are distance one. And 58% of mispredicted branches are within 3 branches of the previous mispredicted branch.

.Second, branch predictions assigned low confidence also occur in clusters. Fig. 10 shows histograms for low confidence branch predictions, similar to Fig. 9.

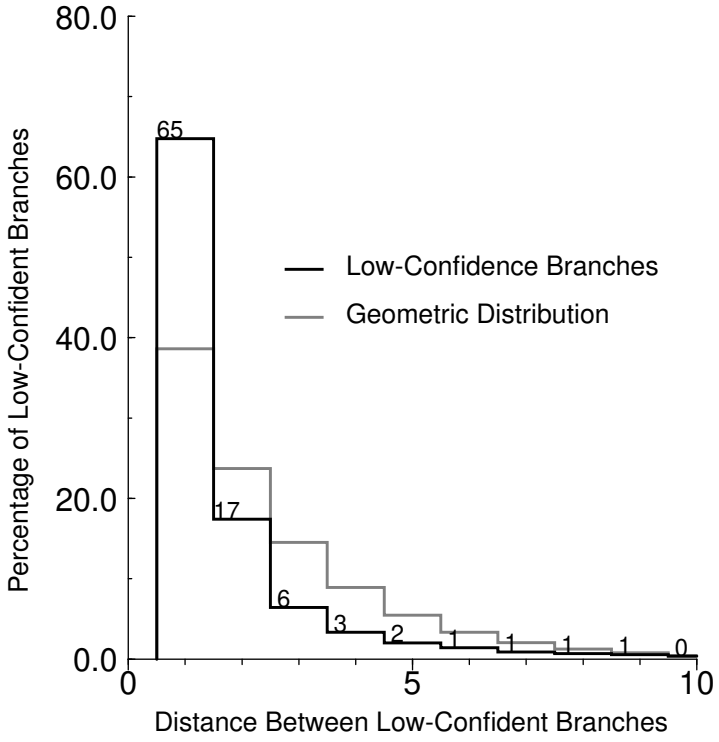


FIGURE 10. Distribution of low confidence branch predictions. Low confidence branch predictions tend to arrive in clusters.

This clustering increases the likelihood of the following two scenarios, both of which diminish the gains from SDPE. If a branch is mispredicted, but not forked, any closely following low confidence branch gains no benefit from forking, it is already on an incorrectly speculated path and will be discarded. And, if a low confidence branch is forked, any closely following low confidence branch cannot be forked.

3.4 Varying Branch Misprediction Penalty

There is a trend toward deeper processor pipelining; for example, the latest generation DEC processor, the 21264 [7], and the latest Intel processor, the Pentium Pro [4], are both more deeply pipelined than their predecessors. As levels of processor pipelining increases, it is likely that the cycles lost for each mispredicted branch will also increase. Wider instruction fetching, register renaming and larger register files will also contribute to higher branch misprediction penalties when the instruction fetch pipeline must be refilled.

Intuitively, as branch misprediction becomes worse, SDPE should become more effective. This is illustrated in Fig. 11 where the simulated branch misprediction penalty is increased from seven to ten cycles by adding extra pipeline stages between instruction fetch and dis-

patch. The overall speedup obtained with SDPE using the FD policy increases from 9.0%

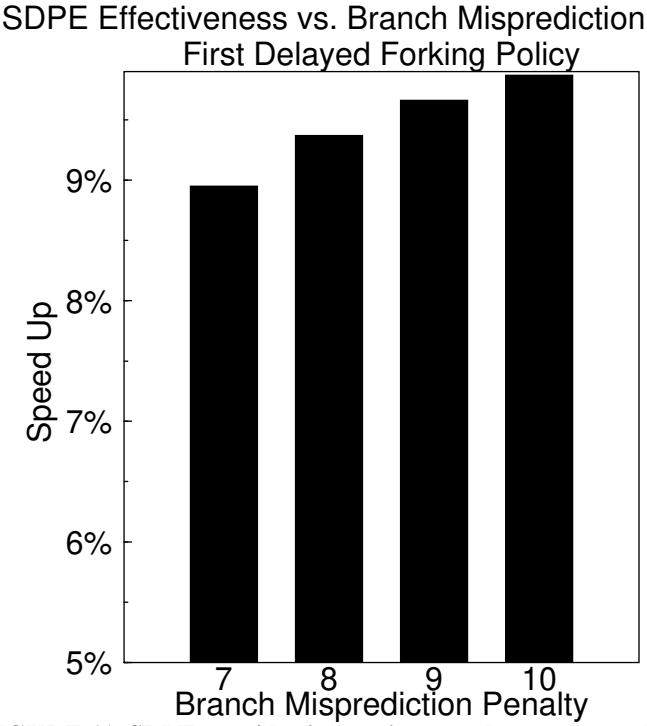


FIGURE 11. SDPE provides increasing speedups, as branch misprediction penalties increase.

to 9.9%. The percentage of execution cycles lost due to branch misprediction increases from 22.2% with a seven cycle branch misprediction penalty, to 25.1% with a ten cycle branch misprediction penalty. On the other hand as the pipeline depth is increased more branches overlap in execution. This increases the effect of clustering, which counteracts the increase in SDPE effectiveness.

3.5 Varying Confidence Mechanism Size

The performance of SDPE depends on the ability of the confidence mechanism to concentrate branch mispredictions in the low confidence set. By reducing aliasing a larger confidence table may improve performance. On the other hand, a smaller table would lead to reduced cost -- the confidence table is 50% larger than a branch prediction table with the same number of entries, owing to the use of 3-bit resetting counters versus 2-bit saturating counters. Consequently, we study performance for a range of confidence table sizes.

For the IBS benchmarks, SDPE appears to be insensitive to the size of the confidence history table. Fig. 12 shows the behavior of SDPE with the FD policy for confidence tables varying from 512 entries to 32K entries. The difference is negligible.

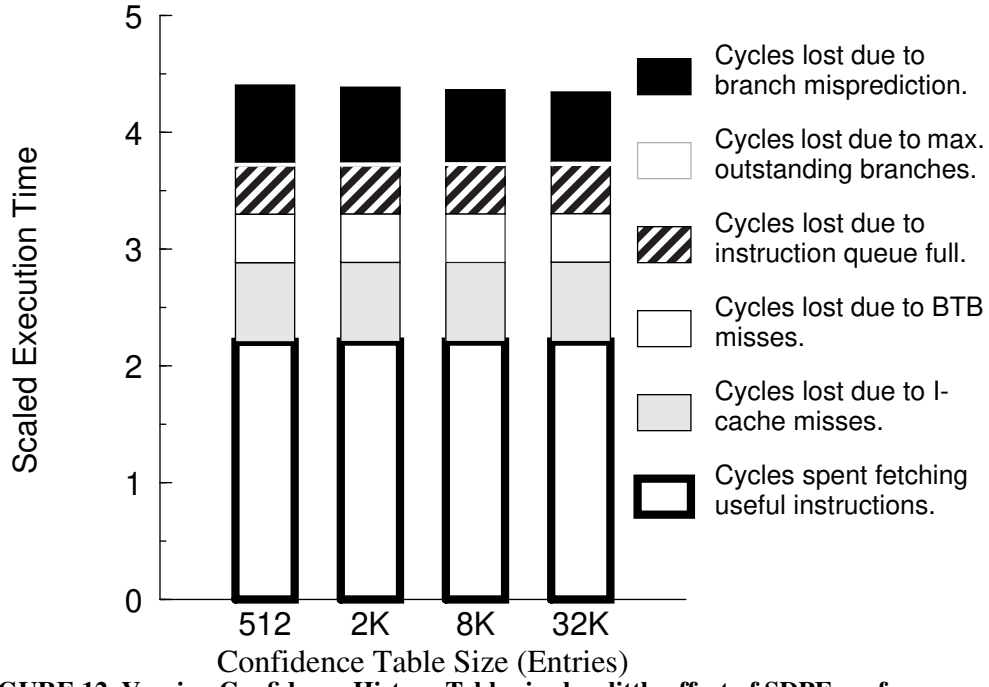


FIGURE 12. Varying Confidence History Table size has little effect of SDPE performance.

The size of the confidence table does have some effect on the accuracy of the confidence mechanism, however. Fig. 13 plots the fraction of forked branches that are predicted and

mispredicted. Larger confidence tables isolate more mispredicted branches in a smaller

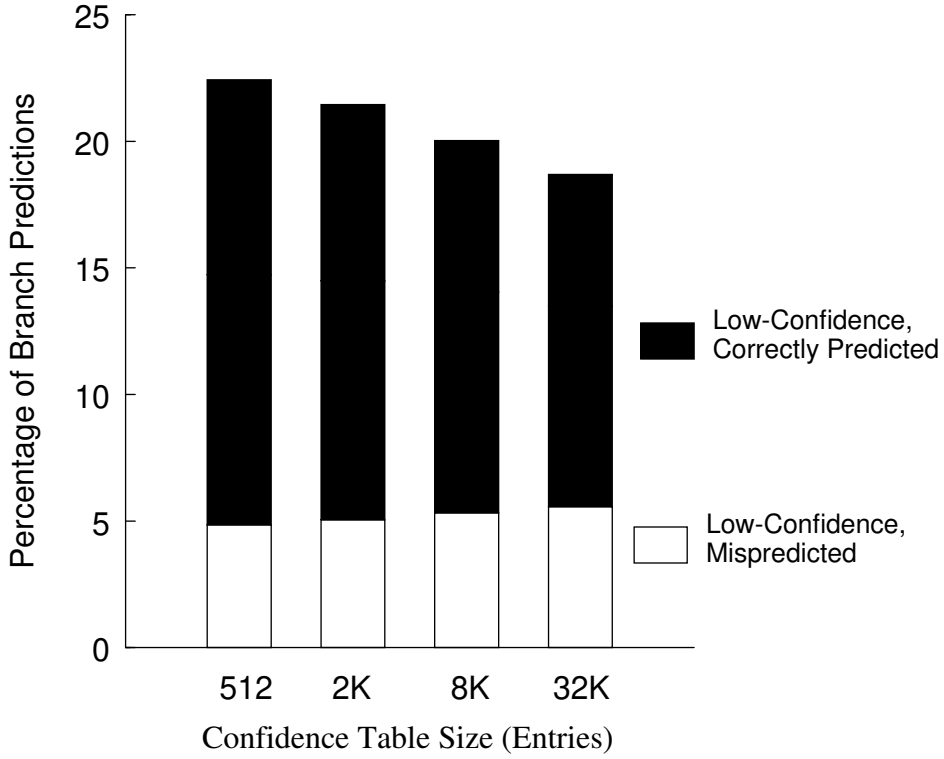


FIGURE 13. Larger Confidence History Tables capture more branch mispredictions in a smaller set of predictions.

low confidence set. However, most misprediction cycles are not lost because the confidence mechanism fails to recognize branches that will be mispredicted, but because of clustering of mispredicted branches and low confidence branches. The small decrease in the accuracy of the confidence mechanism does not have much effect on the percentage of mispredicted branches that fork a second path, or on the total cycles lost due to misprediction.

3.6 Varying Branch Predictor Size

Another way of decreasing the branch misprediction penalty is to increase prediction accuracy by using larger branch predictors. Consequently, we explored the performance benefit of using larger branch predictors. Fig. 14 shows the results.

Fig. 14 compares SDPE using an 8K entry branch predictor with a processor using 32K and 64K entry branch predictors and no SDPE. The figure shows that SDPE reduces

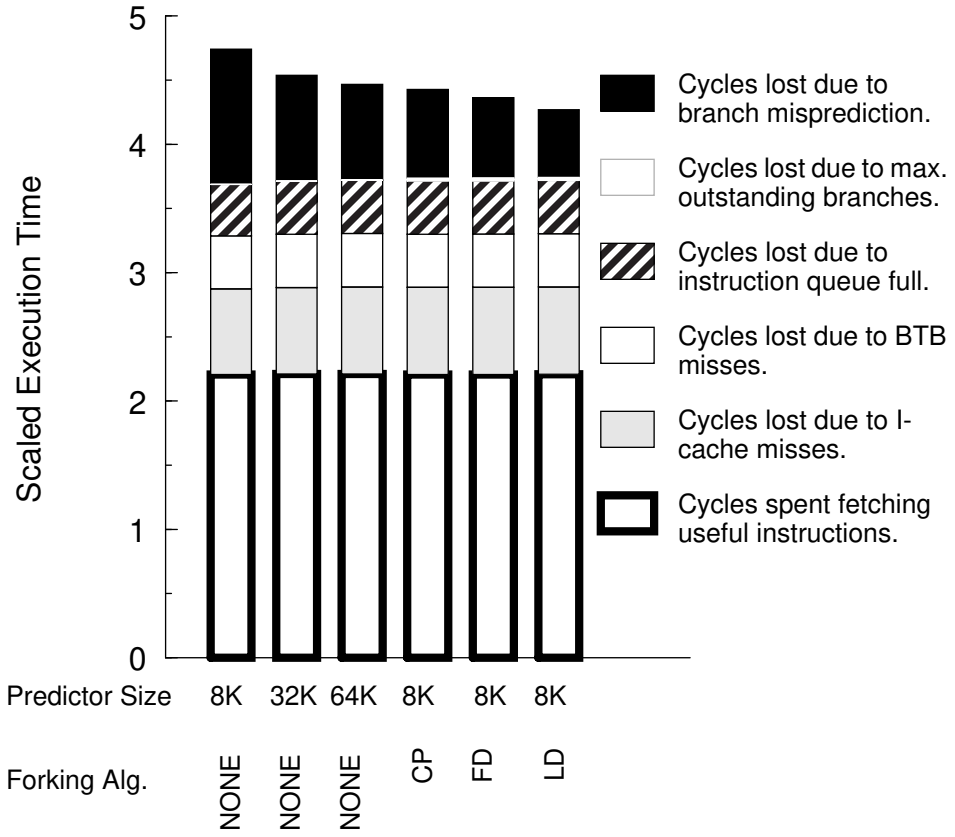


FIGURE 14. SDPE reduces cycles lost to branch mispredictions more than larger branch predictors alone.

branch misprediction penalty more effectively than a larger branch prediction table. Performance gains from larger branch predictors tend to level off as aliasing effects are reduced. Furthermore, SDPE is a fundamentally different solution to the branching problem. Adding SDPE to branch prediction results in improved performance.

3.7 Deficiencies of Simulation Model

Our simulation study represents an initial effort at determining the performance potential of selective dual path execution. Although it allowed a quick exploration of the design space, the simulation model used a number of simplifications -- the most significant being its trace-driven structure. In this section, we identify some of the deficiencies of the simulation model and qualitatively discuss their performance impact.

3.7.1 Data Cache

The simulated execution engine uses an ideal D-cache that always hits. Adding a realistic D-cache will cause instructions that depend on load data to be blocked from issuing while

a cache miss is handled. From the fetch-centric viewpoint taken in this paper, this will cause the instruction window to fill more often, leading to more fetch engine stalls. Because overall program execution time will tend to rise, the percentage of total cycles wasted due to branch misprediction will decrease. This will reduce the relative effectiveness of SDPE.

On the other hand, conditional branches are often dependent on data loaded from memory [12]. If a branch instruction depends on a load that misses in the cache, the branch will stall in the instruction window. And, if the branch is mispredicted, this could significantly lengthen the potential branch misprediction penalty. In this case, SDPE may be more effective with a real D-cache.

3.7.2 Cache Pollution

Cache pollution occurs when cache line replacements occur because of mispeculated instructions. Speculated load instructions may bring in lines that replace lines that would otherwise be used by non-speculative loads. Speculative stores are buffered and do not modify the D-cache until the appropriate branch outcome is known.

Similarly, instruction fetches down a mispredicted path can pollute the instruction cache. However, this is not necessarily detrimental. In fact, fetching down the wrong branch path has been put forward as an effective instruction prefetching strategy [9]. While there could also be some prefetching benefit to D-cache pollution, it is more likely that D-cache pollution will degrade performance.

3.7.3 Stale Branch Predictor State

In a real implementation, the branch predictor is updated when a conditional branch is resolved, and branch predictors tend to rely heavily on recent branch history. However, at any given time there are often several pending branches whose outcome is not known, and which are unable to update the predictor. Consequently, the predictor state being used to predict new branches can be stale and prediction accuracy may be reduced.

The presence of stale predictor state was not simulated in our model. This effect tends to make branch prediction worse in reality than in the simulation. However, this will lead to *increased* benefits from SDPE because SDPE will tend to eliminate more misprediction penalty when there are more mispredictions.

3.7.4 Stale Confidence State

This problem is similar to the stale branch predictor state problem. The confidence mechanism can not be updated until a conditional branch is resolved. Our earlier results on clustering of mispredictions showed that if a conditional branch is mispredicted, the branches immediately following it are more likely to be mispredicted. However, because it takes some time to resolve a branch, a mispredicted branch may not have been recorded when a following branch mapping to the same confidence table entry is predicted. Consequently, the confidence mechanism may indicate that the second branch prediction has high confi-

dence, when in fact it should have low confidence. Hence, stale state in the confidence mechanism may degrade SDPE performance.

4.0 Implementation

Thus far we have discussed the performance potential of SDPE using a simple processor model that does not directly address many implementation issues. In this section an overall implementation for SDPE is proposed, and certain key aspects are being discussed in more detail. The proposed implementation tends to conform to the simulation model, but areas where the implementations departs from the simulated model, or where a departure seems justifiable, are mentioned.

4.1 Overview

The basic pipeline of the proposed implementation, shown in Fig. 15, is be similar to that used in current superscalar microprocessors, for example the MIPS R10000 or DEC 21264 [5, 7]. Instructions from both paths are fetched in the first pipeline stage. Next

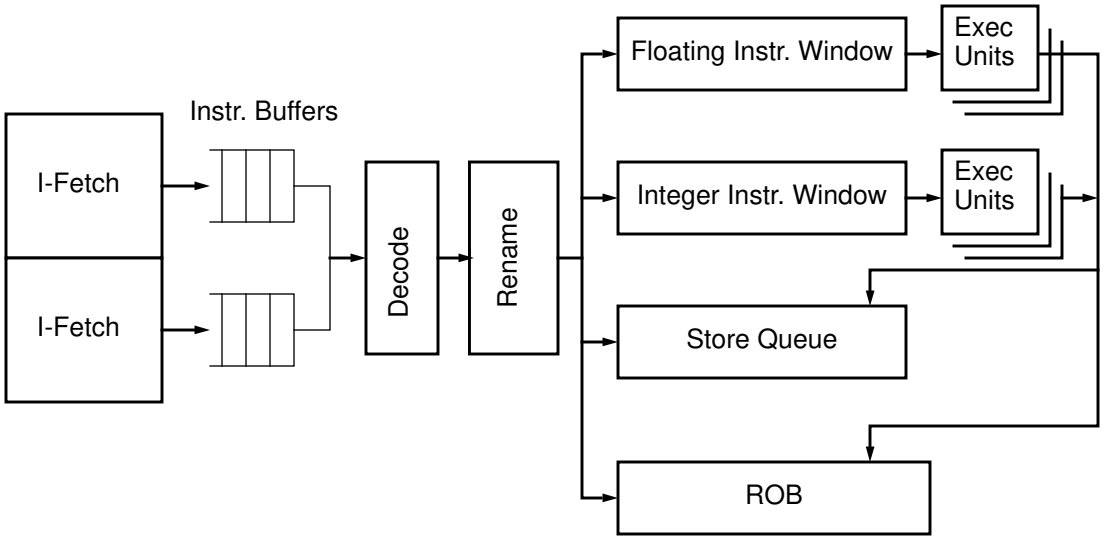


FIGURE 15. The proposed SDPE implementation is patterned after current superscalar processors supporting out-of-order execution.

instructions are decoded. Buffers are added before the decode stage to allow temporary separation of the two instruction paths. Each cycle, instructions from only one path are released from the decode stage to the register rename stage. Instructions from the other path remain buffered. Instructions are renamed and then dispatched into the instruction window. At the same time, reorder buffer (ROB) entries are reserved. Instructions wait in the instruction window until their operands are ready, and then issue to the appropriate execution units. Operand registers are read, and execution begins on the following cycle. When instruction execution completes, results are written back to the physical register file. The ROB is notified of instruction completion, and completed instructions in the ROB are committed in order.

4.2 Instruction Fetch Mechanisms

Initially, we considered the simple approach of time interleaving instruction fetching from the two paths, fetching from only one path during any given clock cycle. Using the trace driven simulator this alternating fetching method was studied. However, this study indicated that the alternating fetch method is not very effective. Following is a brief summary of that study.

Because the predicted path is more likely to be correct than the forked second path, there could be some benefit to giving more of the available fetch bandwidth to the predicted path. Consequently, we controlled the division of fetch bandwidth in two ways. First, the ratio of cycles devoted to the two paths can be varied. The *forked fetch duty cycle* determines the fraction of cycles during which the predicted path is fetched. A duty cycle of 1/2 fetches from each path every other cycle. A duty cycle of 2/3 fetches from the predicted path two out of three cycles. Second, the amount of fetching that occurs on the second forked path can be limited by constraining the number of cycles that can be spent fetching instructions from the forked path. As usual, the trace driven model simulates fetching from the wrong path by fetching no new instructions from the trace. When fetching alternates between the two paths, the simulator fetches no instructions during cycles when the incorrect path would be being fetched.

Fig. 16 shows the results of using alternating fetching with SDPE and the FD policy. A

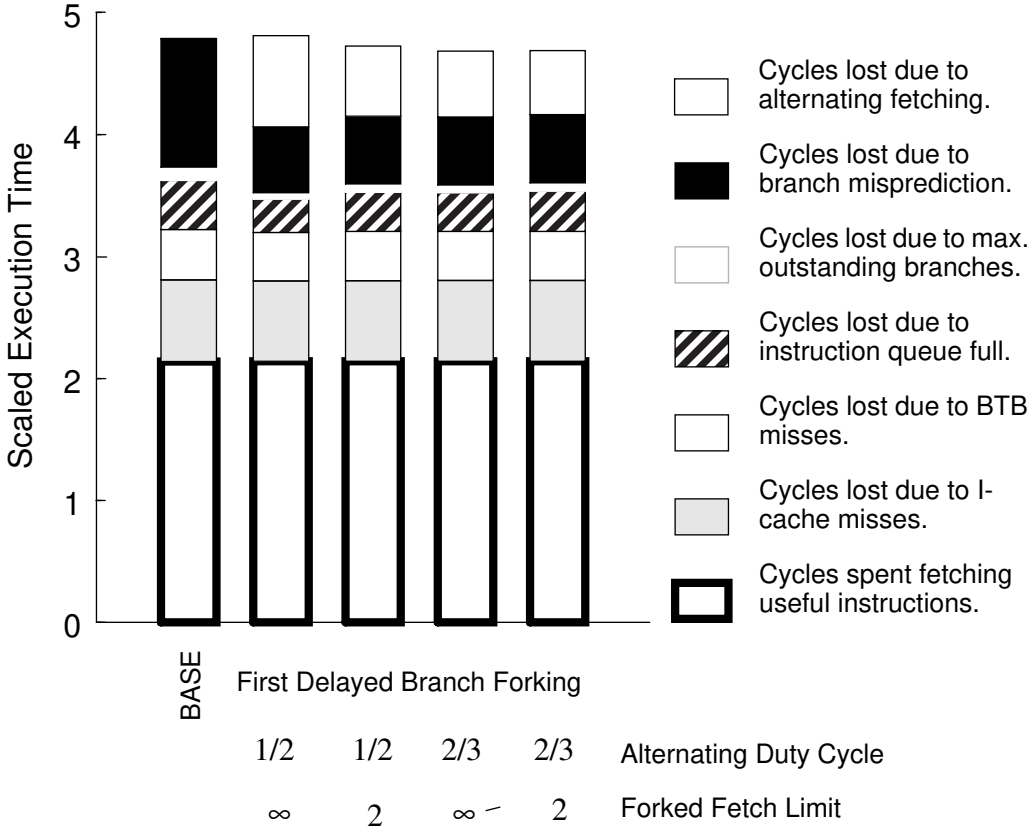


FIGURE 16. Alternating fetching of paths in SDPE reduces most of performance improvement obtained through SDPE.

new segment has been added at the top of the bars to represent cycles lost due to alternating instruction fetching. In some cases SDPE with alternating instruction fetching results in a performance loss versus no SDPE, and only a slight performance gain of 2.5% is seen in the best case. The reason for this loss is fairly simple. When alternate fetching with the single instruction fetch unit we have been assuming, at most one basic block per cycle can be fetched. This is not enough to support two execution paths at an adequate level. And when more bandwidth is given to the predicted path, the forked second path does not get enough bandwidth in those cases where it is needed.

The shortcomings of alternate fetching with a single fetch unit led to a more complex method where two fetch units are used, one for each path. To support this, the I-cache, BTB, branch predictor, and confidence mechanisms are dual ported in some fashion. This can be done through duplication, interleaving to allow multiple accesses, or truly dual porting the arrays. The confidence mechanism only needs to be dual ported if a policy is used where delayed forking can occur on the forked second path. With this method, two basic blocks per cycle can be fetched during dual path execution, one from each path.

An alternative, which we did not study, is to use a single fetch unit that can fetch multiple basic blocks in a single cycle. Such units are proposed in [1, 10, 17].

One section of the fetch mechanism is detailed in Fig. 17. Each cycle the current PC for

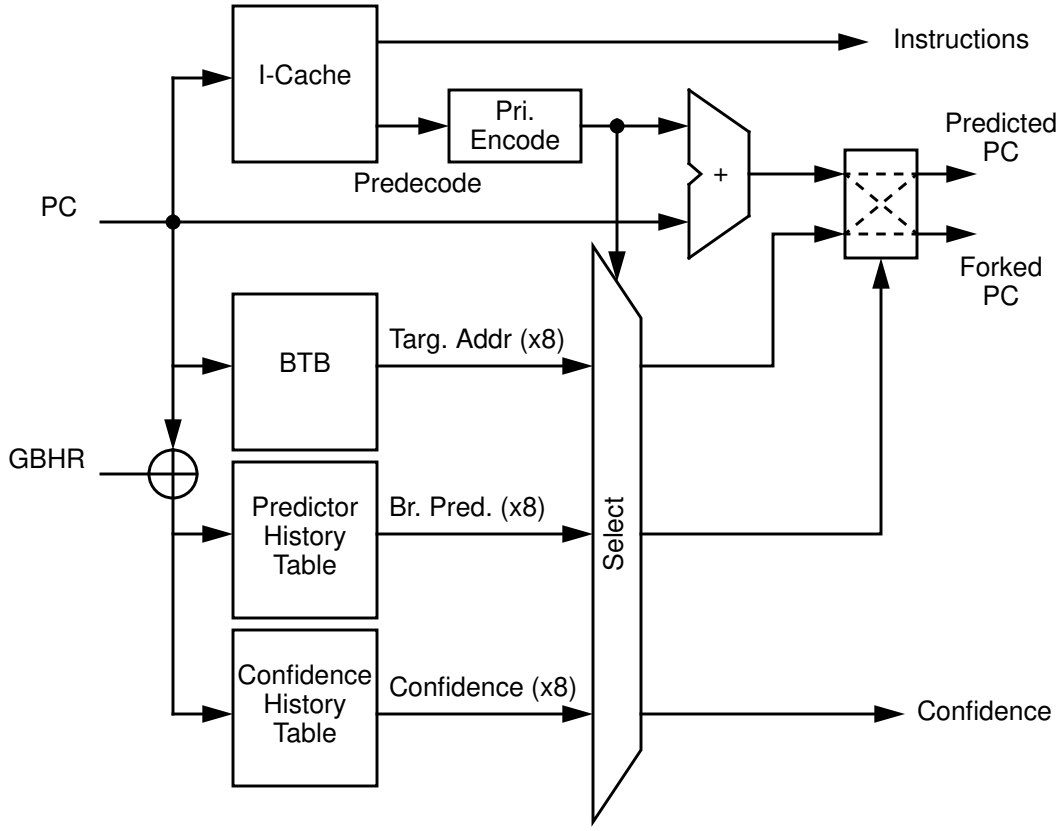


FIGURE 17. One half of the fetch mechanism is capable of fetching instructions from one path each cycle. The fetch mechanism must be duplicated to fetch from both paths each cycle.

the path is used to index into the I-cache. The I-cache provides eight instructions following the given PC. In parallel, the BTB, Branch Prediction History Table, and Confidence History Table are accessed. Along with instructions from the I-cache, predecode bits are read. Predecode bits indicate if an instruction is a control transfer instruction (CTI), and if it is a conditional branch. The predecode bits are used to find the first CTI in the fetched instructions. Once the first CTI is known, instructions following it can be discarded. The predecode bits are used to select the correct jump target address from the BTB, the correct branch prediction from the predictor, and the correct confidence value from the confidence table. This information is used to generate the predicted PC for the next cycle.

4.3 Buffer and Decode Logic

Fig. 18 illustrates the Buffer and Decode pipeline stages. In any given cycle, instructions from either the predicted path or the forked path are allowed to proceed through the decoders. Meanwhile, instructions from the other path are buffered.

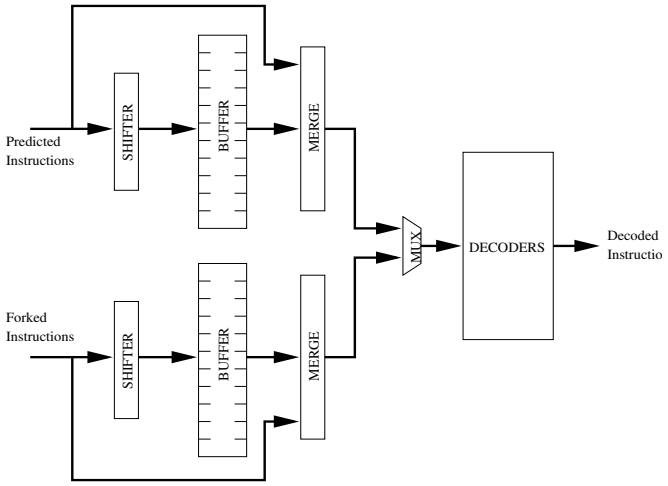


FIGURE 18. Buffering in the Decode stage allows later stages to handle instructions from only one path each cycle, without sacrificing instructions throughput.

Alternating paths for the register rename and the dispatch stages simplifies logic because instructions from only one path must be dealt with each cycle. However, in order to avoid the same kind of bandwidth problem observed in the alternating instruction fetch unit, instructions from multiple basic blocks must be handled in the rename and dispatch stages. With dual fetch units, and alternating between buffers, a buffer will typically accumulate two cycles' worth (i.e. two basic blocks) of fetched instructions before passing them up the pipeline.

Another option is to duplicate the register rename logic and the dispatch logic so instructions from both paths can be renamed and dispatched each cycle. This would cause a large increase in complexity because both paths are renamed from the same physical register file.

4.4 Register Renaming

Register renaming allows instructions from both paths to coexist in the instruction window and execution units. Values produced by instructions proceeding the forked branch can be used by instructions on either path. Values produced by instructions after the forked branch can only be used by instructions on the same path. Simultaneous Multi-Threading

(SMT) uses a similar technique to separate register values produced by multiple threads [11]. A register renaming mechanism is illustrated in Fig. 19.

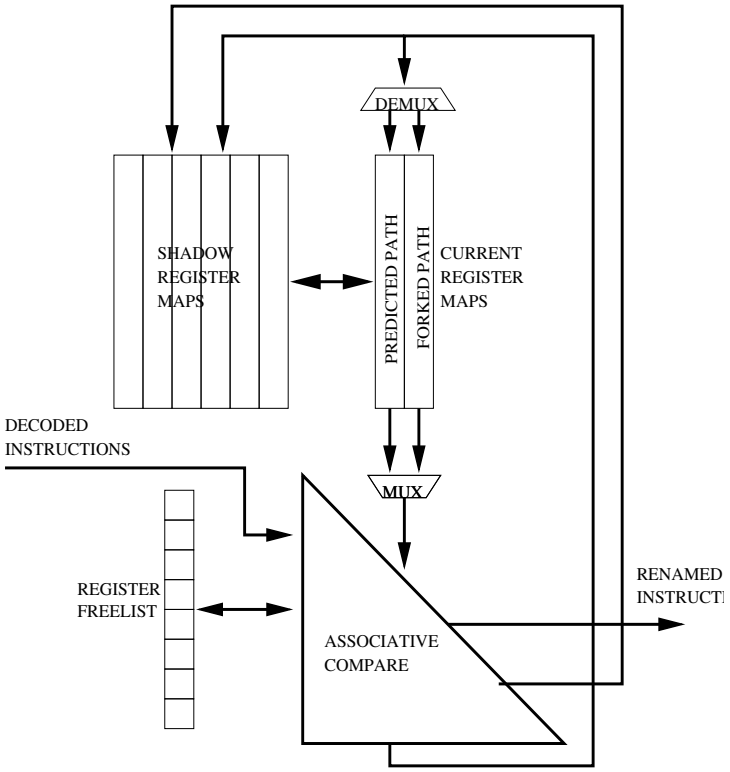


FIGURE 19. Register renaming renames several instructions in parallel, and saves register maps for predicted branches in shadow register maps.

Conceptually, each instruction selects a physical register from the free pool to hold its result. It then looks up the physical register used for each logical operand register in the current register map. Then the register map is updated to include the new mapping from the logical result register to the physical register. The following instruction can then be processed. Of course a real implementation must rename a number of instructions in parallel.

Dual path execution requires two current register maps be maintained, one for each path. When a second path is forked, the current register map is copied into the forked register map. Thus, the maps used for each path are the same at the point of the branch fork. This allows values produced above the fork to propagate to receiving instructions on both paths. As instructions are renamed on each path, different physical registers are mapped to the instructions on each path, and the separate maps are used. This keeps values produced on each path separate.

For recovering from mispredictions and selecting the correct path after branch resolution, an adaptation of the method used in the MIPS R10000 [5] can be used. This method stores a *shadow copy* of the register map as it exists at the time a conditional branch is predicted. Rolling back to the branch in the case of a misprediction involves replacing the current register map with the appropriate shadow map. After a forked branch is resolved, the reg-

ister map for the incorrect path can be discarded. The register map for the correct path must be placed into the current register map for the predicted path, which is used when only one path is being executed.

4.5 Physical Registers

Supporting two paths simultaneously will require more physical registers than for a conventional processor. SDPE essentially increases the number of instructions in flight, and extra physical registers are needed to support these in-flight registers.

If the number of physical registers is a problem, it is possible to reduce the required physical registers by limiting the number of instructions that can be fetched from the forked path, as was done in Section 4.2. The forked path can not use more registers then the number of instructions fetched. Thus, if the number of instructions on the forked path is limited to 16 instructions, and the base processor has 64 physical registers, the SDPE processor will need 80. Sixteen instructions will support 3 to 4 fetch cycles.

4.6 Branch Path Squashing

When a forked conditional branch is resolved, in addition to recovery of the rename map discussed as in the preceding subsection, several other actions must take place.

The forked PC and forked GBHR must be copied from the forked path fetch unit to the predicted path fetch unit. The physical registers used by the incorrect path, but not used by the correct path must be placed back in the free list. Instructions in the instruction window, the store queue, and the ROB must be marked invalid. This can lead to complications because the ROB and store queue intermix blocks of instructions from both paths, so the ROB and store queue may contain holes. These can be ignored, which results in temporarily unused ROB and store queue slots. The holes may be reclaimed if some kind of compaction mechanism is implemented. Another possible solution is implement separate store queues and ROBs for each path.

5.0 Conclusions

Trying to solve the conditional branch problem by building more-and-more accurate predictors will no doubt reach a point of diminished returns. Selective Dual Path Execution provides a way of further reducing branch misprediction penalties by accepting that there will be mispredictions, and then dealing with the situation by executing instructions on both branch paths when there is a relatively high likelihood that the prediction will be wrong.

The initial study presented here indicates that SDPE can potentially reduce branch misprediction penalties by 34% to 50%, depending on the forking policy used. This amounts to an approximately 10% overall performance improvement. An improvement of this size is certainly worthwhile. As issue widths widen and pipeline lengths increase, however,

branch misprediction penalties, as a fraction of total execution time, will rise substantially. Consequently, the payoff for SDPE should rise as well.

A possible problem that we have identified is that improvements are limited somewhat by the statistical properties of conditional branches. Specifically, clustering of mispredicted branches and low confidence branches reduce the fraction of branch mispredictions that SDPE can capture. A partial solution may be to consider additional forked paths beyond two.

We have also observed that an implementation of SDPE must have careful attention paid to supporting sufficient instruction fetch bandwidth. One straightforward approach is to duplicate the instruction fetch unit, although other approaches could be used. In an implementation, register renaming naturally extends to support multiple instruction paths. Some minor complications occur when squashing instructions after a forked branch has been resolved, however.

Finally, we observe that in many respects the implementation of SDPE is similar to simultaneous multithreading (SMT) in that it must support multiple concurrent instruction streams. In fact, SDPE may complement SMT very well. If an SMT processor were given SDPE capabilities, then more processor resources could be directed toward high performance for a single architected thread. An SMT processor might provide a good way to support more than two forked paths, and would provide a good context for investigating trade-offs involving multiple forked paths.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

We would like to thank Eric Rotenberg for the original source for the trace driven simulator, and for numerous helpful suggestions and discussions. We also want to thank Erik Jacobsen for useful information regarding the branch confidence mechanism.

References

[1] T. Conte, K. Menozoa, P. Mills, and R. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Intl. Symp. on Computer Architecture*, pages 333-344, June 1995.

[2] W. D. Connors, J. Florkowski and S. K. Patton. The IBM 3033: An Inside Look. In Datamation, pages 198-218, May 1979.

[3] G.F. Grohoski, J.A. Kahle, L.E. Thatcher, and C.R. Moore. Branch and Fixed-Point Instruction Execution Units. In *IBM RISC System/6000 Technology.* Publication Number SA23-2619. IBM Corp, 1990.

[4] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. Microprocessor Report, pages 9-15, February 1995.

[5] L. Gwennap. MIPS R10000 Uses Decoupled Architecture. Microprocessor Report, pages 18-22, October 1994. 1993.

[6] E. Jacobsen, E. Rotenberg, J. E. Smith. Assigning Confidence to Conditional Branch Predictions. To appear in *Proc. 29th Annual Symp. and Workshop on Microprogramming and Microarchitecture* (*MICRO-29*), 1996.

[7] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Presented at *Microprocessor Forum,* October 1996.

[8] S. McFarling. Combining Branch Predictors. In *Digital Western Research Lab Technical Note TN-36*, June 1993

[9] J. Pierce, Intel and T. Mudge. Wrong-Path Instruction Prefetching. To appear in *Proc. 29th Annual Symp. and Workshop on Microprogramming and Microarchitecture* (*MICRO-29*), 1996.

[10] E. Rotenberg, S. Bennett and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. To appear in *Proc. 29th Annual Symp. and Workshop on Microprogramming and Microarchitecture* (*MICRO-29*), December 1996.

[11] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable simultaneous Multithreading Processor. In *23rd Annual Intl. Symp. on Computer Architecture*, pages 191-202, May 1996.

[12] D. Tullsen, S. Eggers, H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ACM*, pages 392 - 403, 1995.

[13] A. Uht, V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. 22nd Annual Symp. on Computer Architecture*, pages 313-325, November 1995.

[14] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest and J. Emer. Instruction Fetching: Coping with Code Bloat. In *Proc. 22nd Annual Symp. on Computer Architecture,* pages 345-356, June 1995.

[15] D. Wall. Limits of Instruction-Level Parallelism. *Digital Western Research Laboratory Research Report 93/6*, November 1993.

[16] S. S. H. Wang and A. K. Uht. Ideograph/Ideogram:Framework/Architecture for Eager Execution. In *Proc. 23rd Annual Symp. and Workshop on Microprogramming and Microarchitecture* (*MICRO-23*), pages 125-134, November 1990.

[17] S. Weiss, J. Smith. POWER and PowerPC. Morgan Kaufmann Publishers, Inc., pages 100 - 104, 188 - 192, 1994.

[18] T.-Y. Yeh and Y.N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *7th Intl. Symp. on Supercomputing. pages 67-76*, July 1993.