Multipath Execution: Opportunities and Limits

Pritpal S. Ahuja, Kevin Skadron, Margaret Martonosi†, Douglas W. Clark

Depts. of Computer Science and †Electrical Engineering
Princeton University
Princeton, New Jersey 08544
{psa,skadron,doug}@cs.princeton.edu, mrm@ee.princeton.edu

Abstract

Even sophisticated branch-prediction techniques necessarily suffer some mispredictions, and even relatively small mispredict rates hurt performance substantially in current-generation processors. In this paper, we investigate schemes for improving performance in the face of imperfect branch predictors by having the processor simultaneously execute code from both the taken and not-taken outcomes of a branch.

This paper presents data regarding the limits of multipath execution, considers fetch-bandwidth needs for multipath execution, and discusses various dynamic confidence-prediction schemes that gauge the likelihood of branch mispredictions. Our evaluations consider executing along several (2–8) paths at once. Using 4 paths and a relatively simple confidence predictor, multipath execution garners speedups of up to 30% compared to the single-path case, with an average speedup of 14.4% for the SPECint suite. While associated increases in instruction-fetch-bandwidth requirements are not too surprising, a less expected result is the significance of having a separate return-address stack for each forked path. Overall, our results indicate that multipath execution offers significant improvements over single-path performance, and could be especially useful when combined with multithreading so that hardware costs can be amortized over both approaches.

1 Introduction

Modern processors employ a variety of sophisticated branch-prediction schemes to avoid delay penalties imposed by conditional-branch resolution. Correct predictions of branch outcomes, sometimes accompanied by predictions of the target's address, can almost eliminate these penalties. But mispredictions that remain still cause serious disruptions in instruction flow, as the processor wastes time following the wrong path until the misprediction is detected. As issue widths increase and processor pipelines deepen, the misprediction penalty increases.

Many programs suffer a substantial number of mispredictions. Since the delays caused by conditional-branch mispredictions remain a serious problem and more sophisticated prediction techniques still encounter information-theoretic limits [2], we investigate a different kind of remedy: the simultaneous execution of *both* the taken and not-taken instruction sequences following a conditional branch, with cancelation of the one that turns out to be incorrect when the branch is finally resolved. Because additional branches are likely to be encountered before the first branch is resolved, we consider the possibilities and potential benefit of executing more than two paths simultaneously. Each time *multipath execution* forks successfully, it eliminates a misprediction, although possibly with the expense of increased hardware contention.

To appear in the 12th International Conference on Supercomputing, July 1998

Ideally, forking would happen only on mispredicted branches. A *confidence predictor* [6] attempts to evaluate the likelihood that a branch has been correctly predicted. Multipath execution uses confidence prediction to reduce hardware contention: instead of forking, the processor speculates conventionally past higher-confidence branches, following only the predicted path.

Multipath execution requires additional hardware throughout the processor, of course. The most affected section is the instruction fetch unit, which needs extra resources for each path. Since more instructions will be in flight at once, additional functional units, registers, cache ports, and so on are also needed, but the structure and control of this part of the machine are very much like contemporary multi-issue dynamic-execution processors.

In this paper we provide information for architects and designers of future processors who would like to include multipath capabilities. In particular, we investigate the following:

- If hardware resources are abundant, how close can a multipath design come to the performance of perfect branch prediction?
- How should instruction-cache bandwidth be apportioned among competing paths?
- What is the range of performance seen with different confidence-prediction schemes? We consider *naive forking*, which forks at every opportunity; various more selective but still-buildable schemes; and as an upper bound we consider *omniscient forking*, which forks exactly on mispredictions.

Our experiments show that multipath execution can offer sizeable IPC improvements over more traditional single-path execution models: 4 paths yield speedups of up to 30% on future-generation processor configurations. For programs with very high accuracies which are unlikely to benefit from multipath execution, our experiments show that performance does not decrease. Although 4-path speculation would have a significant hardware cost, there are extensive overlaps with the hardware required by multithreading approaches such as simultaneous multithreading (SMT) [15], and combinations of multipath and multithreading make efficient use of such hardware [18].

We also demonstrate that total instruction fetch bandwidth is a key lever on performance, and that heuristics to devote extra fetch resources to probably-correct execution paths can help further improve performance.

Two recent papers are closely related to our work. Wallace, Calder, and Tullsen show significant speedups with a processor organization that combines SMT with multipath execution: thread

contexts not used for program-level threads are used for multipath execution [18]. Klauser, Paithankar, and Grunwald describe Selective Eager Execution, a multipath organization similar to our own, and report similar speedups [8].

In earlier work, Heil and Smith used trace-driven simulation to study the performance of dual-path execution [4]. Tyson, Lick, and Farrens described how to use state from a two-level branch predictor's history table as a dynamic confidence predictor [16]. Mechanisms for dynamically computing branch-prediction confidence were also investigated by Jacobsen, Rotenberg, and Smith [6]. Still earlier, Uht and Sindagi [17] investigated *disjoint eager execution*, in which static probabilities were attached to each branch outcome, and forking decisions made in favor of the path having the highest likelihood. Wrong-path prefetching, which prefetches the first block on the unpredicted path of a branch, was explored by Pierce and Mudge [11] and can be considered a much simpler form of multipath execution. Machines from the late 1970s (the IBM 3033 and 3168) actually incorporated a form of wrong-path prefetching.

The remainder of this paper is structured as follows. Section 2 discusses our proposal for multipath execution in more detail. Section 3 then describes the simulator and benchmarks. Section 4 explores the limits of the multipath idea, Sections 5 and 6 examine fetch bandwidth and confidence prediction requirements, and Section 7 combines the previous ideas by looking at a realistic (though forward-looking) organization. Section 8 concludes the paper.

2 Multipath Execution

At those conditional branches where it is difficult to predict the correct execution path, simultaneously executing both paths prevents losing cycles to misprediction and recovery. Each resulting path, or thread, may reach further branches; given sufficient hardware, control may fork once again. Branches which do not or cannot fork are predicted and speculated conventionally. Once a branch resolves, the wrong path and any of its child paths are squashed. The benefits from eliminating mispredictions in this way are tempered by the increased contention for resources as paths divide. This paper proposes and evaluates mechanisms for implementing multipath execution, and for controlling the potential explosion of paths.

2.1 Differences from Conventional CPUs

The baseline hardware we assume has three main differences from current high-performance CPUs:

Confidence Prediction and Fork Control Unit. At each conditional branch, deciding whether to fork an additional path depends on *(i)* whether there are any path resources currently free and *(ii)* whether the branch is likely to be mispredicted. Item *(i)* is easy to track, but item *(ii)* is more difficult. A simple policy would be to spawn a new path at every conditional branch as long as resources are available; we refer to this as *naive* forking, and introduce its performance results in Section 3.

More elaborate policies try to reduce hardware requirements by being more precise in determining (*i.e.,* guessing) when the branch will be mispredicted. For these policies, we assume a dynamic confidence-prediction unit operates in parallel with the (more familiar) branch-prediction unit. While the branch predictor returns its prediction of whether or not the branch will be taken, the confidence-predictor returns its prediction of whether or not the branch predictor will be correct. One could philosophically consider the branch and confidence predictors to together constitute an even more elaborate branch predictor, but in this study we assume traditional branch-prediction hardware, and that the confidence predictor is a separate unit. Section 6 discusses confidence prediction's successes and limits in greater detail. The upper bound on the performance of all such schemes is demonstrated by a strategy that forks precisely at the right time, on every branch misprediction (when path resources are available). We refer to this as *omniscient* forking. Note that this can still perform worse than omniscient branch prediction: forking can still create resource contention, and if mispredictions are tightly clustered, forking contexts may be exhausted.

Multiple Path Contexts. Multipath execution assumes multiple thread contexts, to allow several control flows to execute concurrently. In particular, the fetch unit must fetch from multiple paths each cycle. (Section 5 evaluates various multipath/fetch-bandwidth combinations.)

The essential hardware for each path is:

- a program counter
- a copy of, or its own port into, elements of the instruction-fetch unit, including the instruction cache, branch predictor, and confidence predictor
- a separate return-address stack—*not* a port into a unified one (Section 5.3)
- a register map, and shadow register maps sufficient for any unforked (conventionally speculated) branches which execution might have to unroll

The fetch unit tags instructions with a thread-ID. Since each thread has its own register map, renaming remains essentially unchanged from a conventional CPU; there are no renaming dependencies among threads. Predecode bits can be maintained in the instruction cache, indicating dependencies among instructions fetched for a particular thread. This further speeds up renaming and ensures that all the instructions fetched in a cycle can be renamed quickly.

The instruction window contains a mixture of instructions from active threads. Instructions are tagged with a thread identifier. One could instead provide separate instruction windows per path, but that approach would give poor performance when only a small number of threads are active, *i.e.,* when the program has few mispredicted branches.

Issue remains unchanged. Renaming ensures a coherent view of the register space, so instructions may arbitrate for execution as soon as their operands become ready. Context tags in the load-store queue ensure that threads see data flow through memory from only the appropriate path.

Selective Misprediction Recovery. In a single-path processor, the CPU handles a mis-speculated path by squashing all the instructions in the RUU after the mispredicted branch. In a multipath processor, however, instructions from many paths—including the correct path—may exist simultaneously and interleaved in the RUU. The CPU must have the ability to selectively squash only those instructions that belong to a particular mis-speculated path and its children. To accomplish this, every path has a unique *path ID* which encodes the forking history of recent branches using binary-prefix notation. New IDs are created when a branch forks, and IDs can be recycled. If a forking branch's ID is x, then the taken path's ID is $x1$, and the not-taken path's ID is $x0$. When that branch resolves, it broadcasts the correct path's ID, say $x1$. The CPU then squashes all instructions that follow the branch and are on the wrong path or a descendent, *e.g.* on paths $x0$, $x00$, $x01$; instructions from $x1$, $x10$, $x11$, etc. are spared. Non-forking, conventionally-speculated branches broadcast similarly, but for a mispredicted branch no instructions from the correct path exist. Squashed instructions turn themselves into NOPs. The resulting

"holes" propagate like normal instructions but are ignored until commit, when they are reclaimed. A similar idea was independently proposed by Klauser, Paithankar, and Grunwald [8].

The path IDs are implemented as circular bitmaps, with a global head pointer and a per-instruction tail pointer. The global head pointer indicates the oldest active forked branch. An instruction's tail pointer indicates how many subsequent branches have forked. Together, the pointers indicate what portion of any bitmap contains useful information: newer instructions, for example, have a tail pointer farther from the global head pointer, because more intervening branches have been seen. Since branches may resolve out of order, and the global head pointer cannot advance until the oldest forked branch retires, bitmaps could grow until the tail pointer would overtake the head. Forking halts at this point. But the bitmap length is equal to the depth of the forking tree: unless the tree is long and narrow and some branch takes an unusually long time to retire, bitmap length is not a problem. A bitmap needs to be at least as long as the log_2 of the maximum number of outstanding paths.

A per-branch pointer must be stored as well, indicating which bitmap position corresponds to each branch. When a branch retires and its thread's local head pointer matches the global head pointer, the global head pointer can be advanced.

2.2 Similarities to Conventional CPUs.

It is important to note that the back end of a multipath CPU is essentially a conventional CPU augmented with extra issue and execution capacity to handle the extra running paths. For the bulk of this paper, our experiments use a "wide-open" back-end configuration chosen such that neither issue capacity nor functional units are a bottleneck. Then in Section 7, we consider cases with more limited back-end capacity.

The hardware, as described thus far, is similar to that used for simultaneous multithreading [15]. Although SMT is a complementary strategy that could be used in conjunction with multipath execution, we do not consider it here. This paper focuses solely on the performance benefits of multipath execution at branches, and corresponding fetch-bandwidth and confidence-prediction requirements, rather than other aspects of program or workload parallelism. Wallace, Calder, and Tullsen examine consider a combined SMT/multipath organization in [18].

3 Methods

3.1 Simulator

We evaluate our ideas using *HydraScalar*, a detailed, multipath, cycle-level simulator we have derived from Wisconsin's SimpleScalar toolset [1]. HydraScalar interprets executables compiled by *gcc* version 2.6.3 for a virtual instruction set closely resembling MIPS IV. HydraScalar instantiates a virtual machine and emulates the object program's execution in order to accurately simulate behavior on mis-speculated paths.

Like a real processor, HydraScalar checkpoints appropriate state as it encounters branches, and then proceeds down the predicted path or down both paths, executing wrong-path instructions as appropriate. Upon detecting a mispredicted branch or a forked branch, wrong-path instructions are squashed, and recovery from the checkpointed state is straightforward. Because instructions from multiple threads are interleaved in the instruction window, squashed instructions leave holes that are eventually freed at commit.

3.2 Baseline Processor

For a baseline CPU, our simulator models an advanced processor resembling in many respects the Alpha 21264 [3]. Integer ALU instructions execute according to an 8-stage pipeline. Instruction fetch takes 2 cycles; decode, register mapping, and queueing take 1 cycle each; the out-of-order back end requires 1 cycle each for register fetch, execute, and writeback; and commit is in-order and takes one more cycle. Because branch mispredictions are detected and handled in writeback, the minimum misprediction latency is 8 cycles. For direct branches, if only the branch target has been mispredicted, a dedicated adder in the decode stage calculates the correct target, and these misfetches suffer only a 2-cycle penalty.

Since the success of multipath execution relies partly on the *failure* of the branch predictor, a key aspect of the processor model is the branch predictor we use. Except where mentioned otherwise, we use a hybrid, McFarling-style branch-prediction scheme [10] similar to that in the 21264 [3]. The scheme uses a 4K table of 2-bit saturating counters to choose between two component branch predictors. Both components are two-level predictors [14], and neither combines PC bits with history bits in indexing the tables of up-down counters. The first component is a per-address-history scheme with a 1K table of 10-bit branch-history registers, and a 1K table of 3-bit saturating counters. The second scheme is a global-history scheme with a 12-bit global branch-history register and a 4K table of 2-bit saturating counters. The branch predictor is updated in commit; this introduces a delay between predicting a branch and updating the predictor, which is why our measured prediction accuracies (Table 1, below) are lower than those reported in most branch-prediction studies.

In the baseline configuration, 16 instructions can be fetched on each cycle, divided among the active threads. A particular thread can predict multiple branches per cycle and fetch past not-taken branches, but cannot fetch past taken branches.

We simulate a unified instruction window, issue queue, and rename-register file—a *register update unit*, or RUU. The architectural registers (32 each for integer and floating-point) are separate and updated on commit. Renaming looks in a register mapping table to determine whether operands reside in the RUU or have been committed to architectural state. This mapping table is copied when a new path is forked, and a shadow copy of the table must be saved each time the processor speculates past a branch. If all a thread's shadow maps are full, fetch stalls for that thread. Our baseline configuration allows 20 in-flight branches per thread.

The confidence predictor responds in a single cycle. This is plausible because this matches the response time of the branch predictor, and our confidence predictors (Section 6) use simpler hardware. Instruction fetch down the forked thread's speculative path can commence on the cycle after the fork decision.

Four final details: *(i)* issue selects the oldest ready instructions, regardless of path, *(ii)* data cache hits take 2 cycles, *(iii)* speculative memory dependencies are managed using a global address-resolution buffer tagged with path IDs, and *(iv)* each thread has a private copy of the return-address stack, copied from the parent at fork time.

In order to focus on multipath behavior, we simulate an infinite processor from the decode stage on. This prevents instruction-window size, decode/rename/issue bandwidth, and the like from obscuring the potential impact of fetch bandwidth or confidence prediction. Section 7 shows that an implementable configuration should also provide substantial speedups.

3.3 Benchmarks

Multipath execution can only benefit applications that suffer from poor branch prediction. Because this paper focuses on fetch-bandwidth and confidence-prediction requirements for multipath, we evaluate our scheme using the five SPECint95 benchmarks summarized in Table 1. Section 7 adds results from the remaining SPECint programs and shows that the technique we suggest ben-

	Warmup Insts	Branches per Instruction		Branch Accuracies	
		All	Cond	All	Cond
go	926 M	0.144	0.111	0.795	0.750
gcc (cc1)	221 M	0.194	0.144	0.855	0.857
compress	2576 M	0.202	0.133	0.921	0.880
li (xlisp)	271 M	0.236	0.137	0.936	0.916
perl	601 M	0.193	0.129	0.918	0.933

Table 1: Benchmark summary. Statistics are from the post-warmup, 50M-instruction simulations. Accuracy for "All" gives target-address accuracy for all branches, including indirect jumps; accuracy for "Cond" gives direction-prediction accuracy for conditional branches.

efits them as well. Given the long execution times of SPEC95 benchmarks and the substantial slowdown with cycle-level simulation, full runs of our benchmarks are infeasible. Instead we collect data for 50 million instructions from runs with the SPEC reference inputs.[1] All benchmarks are compiled using `gcc -O3 -funroll-loops` (`-O3` includes inlining).

We carefully select this window to capture behavior that is representative in terms of branch prediction, cache performance, and overall IPC. For each benchmark, we used fast instruction-level simulation to accumulate *misprediction-rate traces* for entire runs of our benchmarks. Our traces report, for each million-instruction *interval*, the average miss or misprediction rate during that interval. An example of these interval misprediction-rate traces for *compress* appears in Figures 1 and 2, along with the simulation window we use. The x-axis of the graph is time, in millions of instruction executions. Each data point on the graph indicates the misprediction rate seen during that interval.

Like *compress*, most benchmarks show an initial phase of execution where branch prediction is markedly different. Some benchmarks also go through subsequent program phases with better or worse prediction accuracy even after warmup.

We position the simulator window after any initial phase and straddle subsequent phases, as we have shown for *compress*. We check cache behavior in the same fashion. If we follow these requirements, our results are not sensitive to the window's position: even though our simulations are short relative to the whole program, our results change little if we extend our simulations. Simulations from initial phases, on the other hand, can give substantially different results. Interval misprediction-rate traces for the other SPECint benchmarks can be found in [13], along with a more detailed discussion of placing a 50M-instruction simulation window. In particular, these 50M windows should suffice for studying branch prediction.

The simulations run in a fast instruction-level warmup mode, simulating only caches and branch predictor/confidence structures, until shortly before the simulation window is reached. Then they switch to detailed, cycle-level simulation. Table 1 includes the length of the fast-mode ("warmup") phase for each benchmark, including 1 million instructions in which simulation runs in full detail to prime other structures.

4 Quantifying Opportunities for Multi-Path Execution

We begin by considering the performance benefits that multipath execution, at least in an idealized form, can offer. Figure 3 illustrates behavior under four different execution scenarios. In each graph, the bottom and top horizontal lines give baseline and idealized single-path performance. The idealized performance is the IPC that would result from having the directions (but not necessarily the target addresses) of conditional branches predicted with 100% accuracy.

The "naive" line represents forking every time a branch is encountered and a context is available. Even when each path is allowed to fetch as many instructions as the single-path baseline processor—*i.e.*, forking does not divide the fetch bandwidth—IPC rises slowly, falling well short of the ideal even with a large number of paths. If paths must instead share bandwidth, performance peaks at 4 paths (8 paths for *go*) and then declines.

The "omni" line represents omniscient forking. That is, the processor knows exactly when the branch predictor will mispredict, and forks precisely then so long as a context is available. Each path is again allowed to fetch as many instructions as the single-path baseline processor. Performance with this configuration reaches the 100% line with only a small number of paths. *Go* is an exception: it has so many mispredictions that it must fork more often in order to approach the ideal.

These plots show that forking selectively should improve substantially over naive forking. They also show that even with omniscient forking, a non-trivial number of paths are needed to eliminate branch mispredictions. Finally, the fact that performance does reach the 100% line using a moderate number of paths was a useful confirmation of HydraScalar's accuracy.

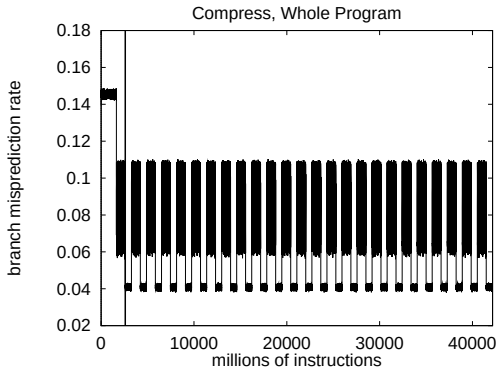

Figure 1: Interval miss rates for a full run of compress with the reference input. The vertical line near the left edge of plot shows the location of our simulation window.

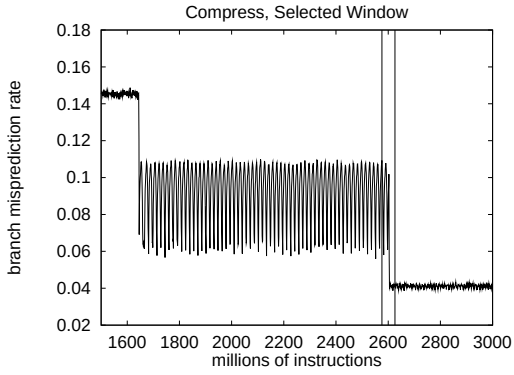

Figure 2: A segment of the full trace for compress. Our simulation window is shown by the two vertical lines.

[1] Some benchmarks come with multiple reference inputs. *Go* results use the `9stone21` input, a 21x21 board, and a play level of 50; *gcc* uses the `cccp.i` input, and *perl* the scrabble input.

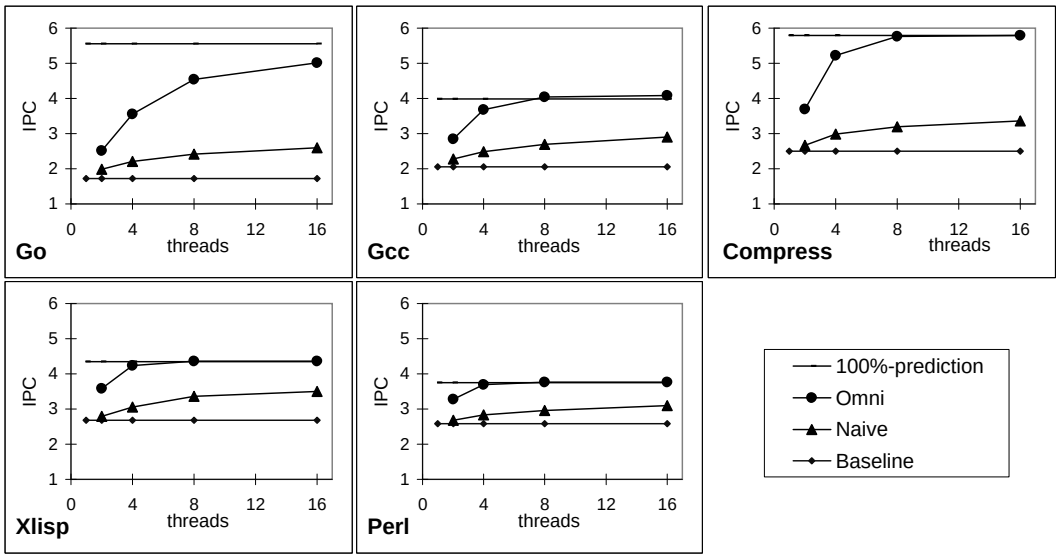

Figure 3: IPCs with omniscient and naive forking. Each path receives as much fetch bandwidth as the single-path baseline processor (16 instructions/cycle). Single-path IPCs for baseline and 100%-direction-prediction configurations are shown as horizontal lines.

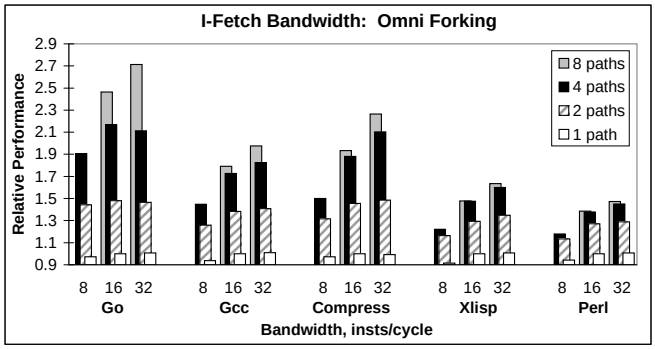

Figure 4: Application IPC as a function of instruction-fetch bandwidths using the omniscient forking policy. (Note non-zero y-axis origin, and that all results are normalized to 1-path/16-insts.)

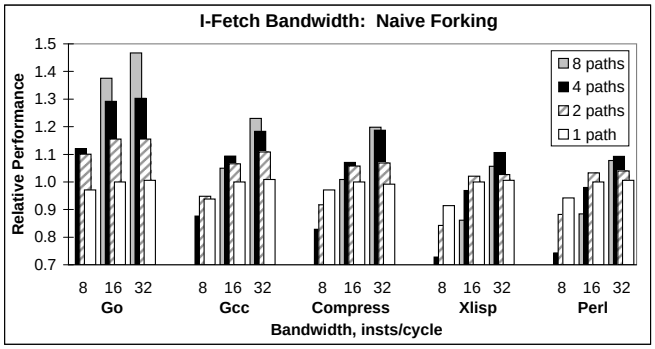

Figure 5: Application IPC as a function of instruction-fetch bandwidths using the naive forking policy.

5 Hardware Effects

Before turning our attention to when forking should occur, we examine some front-end hardware issues that have strong leverage on multipath performance.

5.1 Fetch bandwidth

Figures 4 and 5 show normalized IPCs for our benchmarks with either omniscient or naive forking being used. For a particular benchmark, each layered group of bars represents a certain instruction-fetch bandwidth. We examine bandwidths of 8, 16 and 32 instructions per cycle. No particular path receives priority; available bandwidth is split evenly among the paths. Within a layered group of bars, different shades represent different degrees of multipath. Notice that we do not simulate 8 paths when bandwidth is only 8 instructions per cycle; individual paths receive so little fetch bandwidth that this policy is uniformly terrible. We normalize all results to the 16-instructions/1-path configuration, which is why the other single-path speedups vary slightly from 1.0.

Omniscient-forking speedups depend heavily on fetch bandwidth for 4- and 8-path configurations. These configurations see significant IPC speedups up to 32 instructions fetched per cycle. In contrast, the performance of the 1- and 2-path configurations levels off; fetching more than 16 instructions per cycle offers little performance benefit. While it is intuitive that exploring additional paths requires fetching more instructions, the graphs illustrate that the effect is not multiplicative with the number of paths.

Figure 5 now presents the same data for the naive forking policy. (Note the difference in y-axis scale.) This graph differs in two main ways. First, the IPCs using naive forking are dramatically lower across the board. Second, IPC does not monotonically increase with the number of paths explored if instruction fetch bandwidth is held constant. This is because naive forking allows path resources and fetch bandwidth to be allocated, in part, to paths that turn out to be useless. In the omniscient case, such wasted resources would not occur; forking only occurs when it helps performance. Like omniscient forking, benefits with naive forking depend on instruction-fetch bandwidth.

5.2 Fetch-Priority policy

To avoid wasting fetch resources along incorrect paths, we explore heuristics for priority-based allocation of fetch bandwidth among the executing paths. While we cannot know the correct path *a pri-*

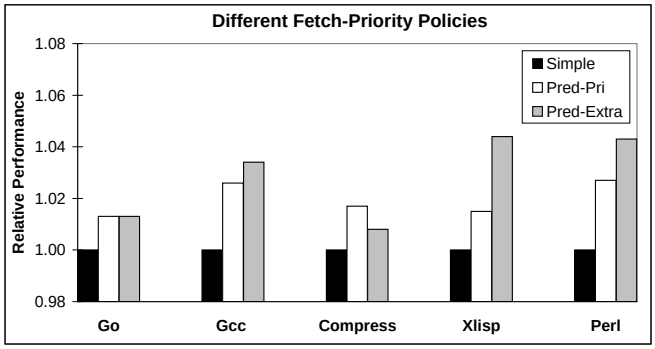

Figure 6: Speedups resulting from priority-based fetch policies: 4 paths/16 insts.

ori, we can note that the predicted path—the path indicated by the branch predictor—is correct most of the time. Since the paths other than the predicted one are often wrong, performance often improves when the predicted path gets somewhat more I-fetch bandwidth than other paths. Applied too extremely, paths starve and behavior reverts to the single-path case.

We compared three schemes: a simple round-robin policy ("simple"), a round-robin policy that ensures the predicted path can fetch every cycle ("pred-pri"), and a predicted-path-priority policy that allows non-predicted paths to fetch only one cache line per cycle, while the predicted path gets any left-over bandwidth ("pred-extra"). Fetch schemes favoring the predicted path require that the predicted path can be known and looked up at any time. This can be implemented using a set of per-path bitmasks similar to path IDs.

Figure 6 shows that giving more priority to the predicted path helps performance slightly— a few percent in most cases. Pred-extra outperforms the other two policies for all but one application. For these five benchmarks, *go* sees the least improvement from priority-based fetch schemes. This is because *go* has the least accurate branch prediction, so it is less likely to properly allocate fetch resources to the appropriate path.

5.3 Return-Address Stack

Return-address stack accuracy can be an especially strong lever on multipath performance. A single, unified stack does not function properly in a multi-path processor. With concurrent paths simultaneously modifying the stack, entries are popped and pushed by both correct and incorrect paths, making corruption almost certain. For example, after a fork, both paths might encounter calls to `printf()`. Both push a return address, even though only one return address belongs on the stack (only one call to `printf()` eventually commits). Neither mechanisms for repairing a return-address stack after mis-speculation [7, 12] nor per-path copies of the top-of-stack pointer can prevent this sort of corruption.

Per-path copies of the return-address stack are the best solution [12]. Multipath execution already requires path contexts; the return-address stack is merely an additional element in the path context, and something that multithreading also requires [5]. Copying the stack should be no more expensive than saving and restoring the register map. All the other studies in this paper have assumed per-path return-address stacks.

6 Confidence Predictors

Forking accuracy is crucial to multipath performance. Forking unnecessarily not only reduces the available fetch bandwidth for the correct path, but can also prevent the hardware from forking in the near future. Conversely, forking too infrequently fails to take advantage of the multipath-execution hardware. Prior sections have discussed potential benefits of multipath execution assuming very simple or unrealistic forking policies at each conditional branch (*i.e.*, naive and omniscient). Here we look at more selective and realistic policies in detail.

6.1 Confidence Prediction Methods

Confidence-based forking mechanisms can be constructed from two orthogonal components: a policy for counting mispredictions of recent branches, and a policy for applying this counting history.

Hardware State. First we consider ways to count mispredictions; these basic policies were first discussed in [6].

No Hardware Table. First, and most simply, if no hardware state is kept, then we either revert to the naive policy discussed earlier or use a static profile-based scheme discussed below.

Ones Counter. The simple hardware-based confidence predictors considered here use a table indexed by the branch PC. With a *ones-counting* scheme, each entry in the table is an n-bit shift register. These bits represent the accuracy of the branch predictor for the last n branches that mapped to this entry. A 1 represents a correct prediction and a 0 represents a misprediction. If the number of 1's exceeds a certain threshold, the confidence predictor has high confidence that the branch predictor will correctly guess the direction of the next branch indexing to this entry, and the CPU does not fork. Conversely, if the confidence predictor has low confidence, the CPU forks (if there are contexts available). The CPU updates the table entries when a branch is committed. Our experiments use 16K-bit tables—2K entries by 8 bits per entry—and a threshold of 6 (*i.e.*, fork if 2 or more mispredictions have been seen). At the start of the simulation, each table entry contains all 0's.

Saturating Counter. In a saturating-counter scheme, each table entry is interpreted as a signed integer. Each time a branch is correctly predicted or mispredicted, the entry's contents are incremented or decremented, respectively. If the integer is greater than a given threshold—which is not necessarily 0—high confidence is predicted. Our experiments use 4K-entry tables with 4-bit saturating counters and a threshold of 4. Table entries are initialized to 0.

Resetting Counter. In this scheme, as with saturating counters, correctly-predicted branches cause the contents of the appropriate table entry to be incremented. However, when a branch is mispredicted, the entry's contents are reset to 0, not merely decremented. The table entries' contents are thus interpreted as unsigned numbers. Our experiments use 4K-entry tables with 4-bit resetting counters and a threshold of 11. Table entries are initialized to 0.

Means for Applying Confidence State. We can extend the basic techniques above by incorporating policies for *applying* knowledge from the dynamic confidence history.

No Policy. This reverts to one of the basic techniques above.

Profile-Based. Any of the above schemes can be applied in a profile-based manner. In profile-based approaches, each conditional branch is classified based on observed misprediction statistics from a pre-run profiling step. Very reliably-predicted branches are placed in a "do not fork" category—multipath hardware is never allocated to these branches. Branches that mispredict frequently are placed in a "fork aggressively" category—the CPU should always attempt to fork when these branches are encountered. There may also be intermediate, "fork conservatively" categories for somewhat reliably-predicted branches; such branches initiate forking if certain conditions are true (see below). A simple and fast tool can perform the profiling, and, if desired, the profile information can be augmented by the compiler. A change in the instruction set architecture may be required: an additional $log_2 N$ bits, where N is

the number of categories, allocated within conditional branch instructions. Profile-based confidence can be performed without a hardware table [6]—branches are statically placed in either the "do not fork" or "fork aggressively" categories. The processor attempts to fork only when a branch from the latter category is encountered. Assigning all branches with a greater than 35% misprediction rate to the "fork aggressively" category is successful. Other cutoffs (*e.g.*, 40%, 50%, 70%, etc.) were also tried, with lesser success, although that may change for non-SPEC95 programs.

When combining profiling with dynamic confidence history, each category of profiled branches has a different forking threshold. A branch's table entry count is compared to its class threshold, and if the former exceeds the latter, the CPU does not fork. For example, in our *ones-profile* predictor, branches in the "fork aggressively" category have a threshold of 6. These branches initiate forking if the number of 1's in the table entry is less than or equal to 6 (*i.e.*, 7 or 8 of the previous 8 branches must have been correctly predicted to prevent forking). Branches in the "fork-conservatively" category have a threshold of 4. In this case, only 5 (or more) of the previous 8 branches need be correctly predicted to prevent forking. We tried three different ways to assign categories to branches, based on either misprediction rates or misprediction counts. No single categorization was optimal for all benchmarks. The data presented in Figure 7 uses the best categorization scheme for each benchmark. We used these same categorizations for the combined profile/resource-based predictor experiments (see below).

Resource-Based. With a finite number of path contexts, the CPU cannot always fork when it wants to. If the CPU has reached its forking capacity, subsequently-fetched branches cannot fork until some contexts are freed. Therefore the CPU should sometimes refrain from forking on a branch that is marginally low-confidence, in case an even lower-confidence branch soon follows. Resource-based schemes address this. These schemes use the same counting tables discussed above, but with multiple thresholds to determine whether to fork. When fewer contexts remain free for forking, these schemes use stricter (*i.e.*, lower) forking thresholds. This makes it easier for a branch to be rated high confidence, reducing the likelihood that the CPU will be unable to fork on a truly low-confidence branch later on.

Combining Profile-Based and Resource-Based Approaches. Incorporating hardware availability (*i.e.*, number of forking contexts available) with a profile-based predictor is also possible. Profiling now categorizes branches into, say, categories 0, 1, 2, and 3. The lower the category, the lower the confidence (and the more aggressive the attempt at forking). When deciding whether to fork, a branch's category is now compared with the number of available forking contexts. If the branch's category is less than the number of free contexts, then fork. Thus, category-0 branches are always forked if any free contexts are available. Category-*N* branches are never forked given a machine with *N* contexts.

6.2 Results

Figure 7 shows the relative performance of these confidence-prediction schemes on a 4-path processor, normalized against the single-path performance. The ones counters, resetting counters, simple profiling, and resource-based ones-counters (ones-R) performed quite well over all the benchmarks. The profiling/resource-based hybrid scheme (profile-R) performed well for *compress*, *xlisp*, and *perl*, but quite poorly for *go*. The ones-profile scheme was poor for some benchmarks but adequate for others. The saturating-counters scheme performed miserably over all benchmarks.

We tried varying both the widths and thresholds for the ones, saturating, and resetting counters. Of the configurations explored, none outperformed the configurations presented here. Since the

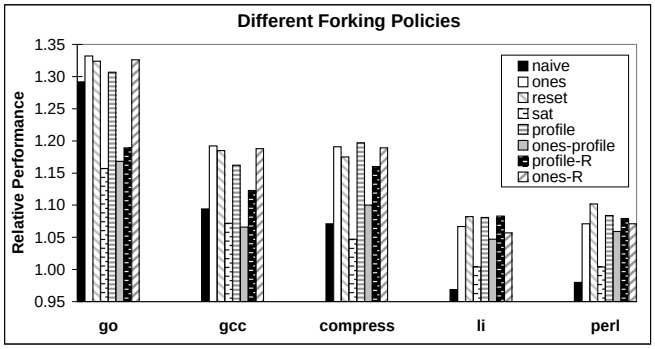

Figure 7: Speedups of a four-context processor, using different forking policies, normalized against a non-forking processor. (Note non-zero y-axis origin.)

ones-counter's performance was similar to the resetting counters and superior to the saturating counters, we did not evaluate combinations of resetting or saturating counters with profiling-based or resource-based policies.

The relatively good performance of the simple profiling scheme warrants further research, especially since no confidence-prediction hardware is required.

We performed these experiments for 2- and 8-context processors also and obtained similar relative performance.

6.3 Performance on Well-Predicted Programs

For the most part, this paper has focused on applications for which branch predictors have mediocre or poor accuracy, and which could benefit from multipath execution. A multipath scheme, however, should have a forking policy that helps these applications without unduly harming applications that do get good branch predictor performance. To check on this, we have gathered results for *vortex*, a SPECint95 benchmark which gets 98% accuracy with our branch predictor. Speedup with 4 paths and a naive confidence scheme is 0.74. But speedup with 4 paths and a ones-counting scheme is 1.02. While naive forking performs disastrously for *vortex*, the basic ones-counting scheme is successful at maintaining the performance of the single-path approach.

7 Finite Processor Model

Up to this point, the processor modeled has had a "wide-open" back end (unlimited functional units, RUU capacity, etc.) in order to isolate the fetch-bandwidth and confidence-prediction requirements of multipath execution. In this section, we limit the processor to a hardware model that may be realizable in a technology generation or two.

This finite processor can fetch, decode, issue, and commit 16 instructions each cycle. The capacities of the RUU and load-store queue are 256 and 128 instructions, respectively. The level-one instruction and data caches are 256KB and 2-way associative, with 32-byte lines and 2-cycle hit times. The unified, 16MB, level-two cache has 4-way associativity, 32-byte lines, and a 15-cycle hit time. The per-path return address stacks have 32 entries each.

Figure 8 summarizes our results for a 4-path run using this platform. It plots performance relative to a single-path baseline run using the limited configuration (this makes the "wide-open" speedups seem larger than those reported elsewhere in this paper).

For all the benchmarks, limiting the back end has substantial effect, reducing the performance by 12.4%. Nevertheless, across the five benchmarks that have been the focus of this paper, speedups

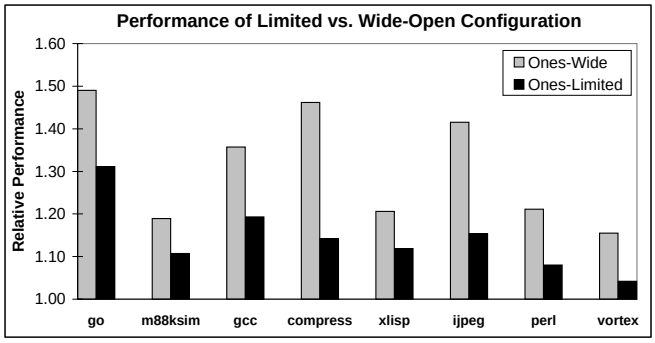

Figure 8: Relative performance of multipath execution with a wide-open and with a limited back-end configuration.

from a realistic multipath configuration range from 8% to 31%, with an average of 16.9%.

Although *m88ksim*, *ijpeg*, and *vortex* have not been among our focus applications, we include results for them to show that multipath execution benefits these programs as well. *M88ksim* achieves a branch-prediction accuracy of 94.2%, *ijpeg* achieves 87.4%[2], and *vortex* achieves 97.8%. Speedups for these programs range from 4% for *vortex* to 12% for *ijpeg*, making the average speedup over the entire SPECint suite 14.4%. This suggests that a ones counter can control forking and not harm performance for programs with good branch-prediction accuracy.

8 Conclusions and Future Work

As CPU issue widths become larger and pipelines become deeper, the performance costs of conditional branches become increasingly significant. Despite use of a sophisticated branch predictor, IPC would improve in our focus applications by factors of 1.5 to 3.25 if the direction of conditional branches could be predicted with 100% accuracy. For these reasons, our work has proposed multipath-execution schemes and evaluated their performance on a range of programs.

While the hardware requirements of a 4-path approach are not insignificant, this technique substantially complements other likely directions for future microprocessors. These include clustered approaches, multithreaded processors [9], and particularly simultaneous multithreading (SMT) [15]. Multipath speculation at conditional branches is largely an orthogonal way to make use of this hardware at times when other multithreading mechanisms do not. For example, Wallace, Calder, and Tullsen describe a system that combines SMT with multipath execution in [18].

From this work, we draw the following conclusions: First, by exploring performance trends for up to eight paths, we have shown that multipath execution can in the limit approach the performance of an idealized scheme in which conditional branches are predicted with 100% direction accuracy. But even with omniscient forking, this requires a non-trivial number of paths.

Second, at practical path counts (4 or less), multipath execution offers speedups over single-path execution even with naive forking schemes. Using a simple but more selective ones-counting confidence predictor, rather than the naive scheme, affords speedups of up to 4–31% over the single-path case, with an average of 14.4% for the SPECint suite.

Third, instruction fetch bandwidth and return-address stack configuration are two of the most important levers on performance. In addition, systems implementing priority-based fetch schemes can further improve performance by a few percent.

Finally, among several possible confidence-prediction schemes, the most effective for our benchmarks and branch predictor is a fairly simple ones-counting approach.

The significant performance benefits of multipath execution warrant further study, particularly in conjunction with multithreading and other forms of parallelism.

[2] But *ijpeg* has large basic blocks, resulting in a branch frequency about $1/3$ that of the other programs (0.063 branches/inst).

Acknowledgments

This work was supported in part by NSF grant CCR-94-23123, NSF Career Award CCR-95-02516 (Martonosi), and an NDSEG Graduate Fellowship (Skadron). We thank the referees for their helpful suggestions.

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Tech. Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

[2] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proc. ASPLOS-VII*, pages 128–37, Oct. 1996.

[3] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[4] T. H. Heil and J. E. Smith. Selective dual path execution. Tech. report, Univ. of Wisconsin-Madison Dept. of Elec. & Comp. Eng., Nov. 1996.

[5] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *Proc. PACT '96*, pages 169–73, Oct. 1996.

[6] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proc. Micro-29*, pages 142–52, Dec. 1996.

[7] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *Int'l J. Parallel Programming*, 25(5):363–83, Oct. 1997.

[8] A. Klauser, V. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath Architecture. In *Proc. ISCA 25*, July 1998.

[9] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proc. ASPLOS-IV*, pages 308–318, Oct. 1994.

[10] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[11] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proc. Micro-29*, pages 165–75, Dec. 1996.

[12] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. Tech. Report TR-577-98, Princeton Dept. of Comp. Sci., Mar. 1998.

[13] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Tradeoffs among branch prediction, instruction-window size, and cache size. Tech. Report TR-578-98, Princeton Dept. of Comp. Sci., April 1998.

[14] T.-Y. Teh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. ISCA-20*, pages 257–66, May 1993.

[15] D. Tullsen, S. Eggers, J. Emer, et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multitheading processor. In *Proc. ISCA-23*, May 1996.

[16] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Tech. Report CSE-TR-346-97, University of Michigan, EECS Dept., 1991.

[17] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proc. Micro-28*, pages 313–25, Dec. 1995.

[18] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proc. ISCA 25*, July 1998.