Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms

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

Departments of Computer Science and Electrical Engineering
Princeton University
E-mail: {skadron,psa,martonos,doug}@cs.princeton.edu

Abstract

This paper evaluates several mechanisms for repairing the return-address stack after branch mispredictions. The return-address stack is a small but important structure for achieving better control-flow prediction accuracy and therefore better performance. But wrong-path execution after mispredictions frequently corrupts the return-address stack, making repair mechanisms necessary. If the processor implements multipath execution—simultaneously executing both sides of a branch—the contention among different paths makes the problem more severe.

For conventional, single-path processors, this paper proposes saving both the top-of-stack pointer and the top-of-stack contents for later restoration in case of a misprediction. This simple technique achieves nearly 100% hit rates and improves performance by up to 8.7% compared to a stack with no repair mechanism. For multipath processors, providing each path with its own return-address stack completely eliminates contention, improving performance by over 25%.

1 Introduction

Control-flow mispredictions have become a serious bottleneck to better processor performance. Each misprediction results in five, ten, or more cycles of wasted instruction fetch. Because mispredictions so strongly constrain fetch bandwidth, improving downstream components—instruction-window size, issue width, etc.—has limited payoff. This paper examines procedure-call returns, one important source of mispredictions.

Procedure returns present the same problem as other indirect branches: because a procedure might be called from many different locations (consider `printf()`), the target of a particular return instruction varies. But extra information is available: returns should pair up with corresponding subroutine calls. Since the return address is known at the time of the call (it is usually the next instruction after the call), a *return-address stack*[1] [20, 37] can match returns with corresponding calls. Like other prediction techniques, the stack's prediction is only a hint: the return instruction must still read the actual return-address register and commit. If the predicted return address is wrong, the corresponding mis-speculated computation must be squashed.

A simple stack, however, fails in the presence of speculative execution. The fetch engine predicts branch outcomes and speculatively fetches subsequent instructions: this may result in mis-speculated calls or returns that are later squashed. Calls and returns, however, update the stack in the fetch stage, while mispredictions are detected much later, in the writeback stage. Calls and returns on a wrong path therefore corrupt the stack. This becomes a substantial problem in current, high-performance processors that speculate aggressively. Without some repair mechanism to undo the effects of squashed instructions, the stack contents often do not correspond to the program's current call-return sequence, and return-address mispredictions result.

This paper examines the behavior and performance impact of return-address stacks, concentrating on mechanisms for repairing or preventing corruption. Cycle-level simulation of the SPECint95 suite is used to evaluate several mechanisms, and to gauge sensitivity to stack size. The paper also considers how a new execution model, *multipath execution* [1, 21, 35, 36], affects the return address stack. Multipath execution simultaneously executes down both sides of one or more branches. If all the concurrent paths modify one unified stack, corruption again becomes so frequent—even with repair mechanisms that work well for single-path execution—that performance is generally better without a stack. We suggest two different mechanisms for avoiding this corruption.

Related work. The literature discusses procedure returns infrequently. The most relevant work, by Jourdan *et al.*

[1] This paper uses "stack" and "return-address stack" interchangeably, and never speaks of the actual program stack in memory.

[19], evaluates repair mechanisms for the return-address stack as well as other branch-prediction structures. As we do, they find stack repair to be necessary for effective return-address prediction. They evaluate a self-checkpointing mechanism that saves popped entries to avoid overwriting them with future mis-speculated pushes. Each stack entry, in addition to saving a return address, also contains a pointer to the next valid stack entry. Earlier work includes Kaeli and Emma's 1991 paper describing a two-stack return-address predictor [20], and Webb's 1988 paper [37].

Jacobsen, Rotenberg, and Smith discuss return-address prediction in the context of a processor organized around traces [18]. The trace mechanisms replace the return-address stack. Returns that do not appear at the end of a trace are followed in the same trace by instructions from the return site. These return-site instructions may or may not be correct, but replace the prediction that a return-address stack would provide. Traces which end in a return can be predicted using path-based next-trace prediction. This is particularly easy if the call appears in the same trace as the corresponding return. They also describe a *return history stack* which is not used to predict returns, but instead to cope with the loss of path history caused by subroutines. They observe that branch behavior after a subroutine call often correlates with behavior before the call. Unfortunately, subroutines of any substantial length fill up path history needed for next-trace prediction, replacing information about path history that precedes the subroutine call. The return history stack saves the pre-call history and restores it after subroutine returns.

Hily and Seznec evaluate return-address-stack performance in a simultaneously-multithreaded processor; because calls and returns from different threads can be interleaved, they find per-thread stacks are a necessity [16].

Most work on control-flow prediction has focused on predicting directions of conditional branches [2, 4, 11, 24, 25, 31, 38, 39]. Attention is however turning toward other types of branches, as obtaining further improvements in conditional-branch accuracy becomes difficult. Researchers have recently begun focusing on general indirect jumps. Examining C++ programs, Calder and Grunwald observed that many indirect branches have a favored target. They proposed associating a two-bit counter with branch target buffer (BTB) entries: two consecutive instances of a particular target address must occur in order to change an indirect branch's target [5]. More recently, Chang, Hao, and Patt have proposed augmenting a BTB with branch history to select among various possible targets for a particular indirect branch. Instead of history bits indicating branch directions, their proposed history contains previous branch targets [7]. Driesen and Hölzle further examined history mechanisms for indirect-branch prediction [9]. Because returns are a special case of indirect branch, history mechanisms like these can potentially capture caller history well enough to distinguish among possible return targets. These general mechanisms, however, do not achieve the near-100% accuracies possible with a return-address stack. Because return-address stacks are so inexpensive and can be so effective, processors are likely to retain the return-address stack for the foreseeable future.

Most current high-performance processors include a return-address stack, but precise details about their management are rarely available. The DEC Alpha 21164 implements a 12-entry stack that can overflow and underflow [8]. The 21264 increases the size to 32 entries [15], and includes a (proprietary) stack-repair mechanism [10]. The Pentium MMX and Pentium II also implement a repair mechanism which uses valid bits to detect corrupted entries [13]. Cyrix recently patented a repair mechanism that preserves the top-of-stack pointer [27], similar to one of the mechanisms described here.

Contributions. This paper makes the following contributions:

- We closely examine the performance of return-address stacks in a simulated model of a contemporary processor, and find that corruption caused by branch mispredictions is an important source of wasted cycles.
- We show that a simple stack-repair mechanism can remove essentially all speculatively-caused stack corruption, improving performance on the SPECint95 programs by up to 8.7%.
- We also show that performance of multipath execution benefits substantially from per-thread return-address stacks: performance improves by up to 26% compared to a single, unified stack.

2 Design Issues

2.1 Basic stack operation

Figure 1 shows a simplified view of a modern fetch stage. Calls—either *jump-and-link* or *jump-and-link-register* instructions—push a return address onto the stack. Returns, instead of reading the BTB, receive a prediction by popping the stack. The most recent return jumps back to the most recent call site, and so forth; so long as calls and returns match up, returns are correctly predicted. The stack can be managed in a variety of ways, but we model it as a circular LIFO buffer, pushed and popped during fetch. The stack can both overflow (overwriting the oldest entry) and underflow (returning invalid data from an already-popped entry). We briefly return to the subjects of overflow and underflow in Section 4.3.

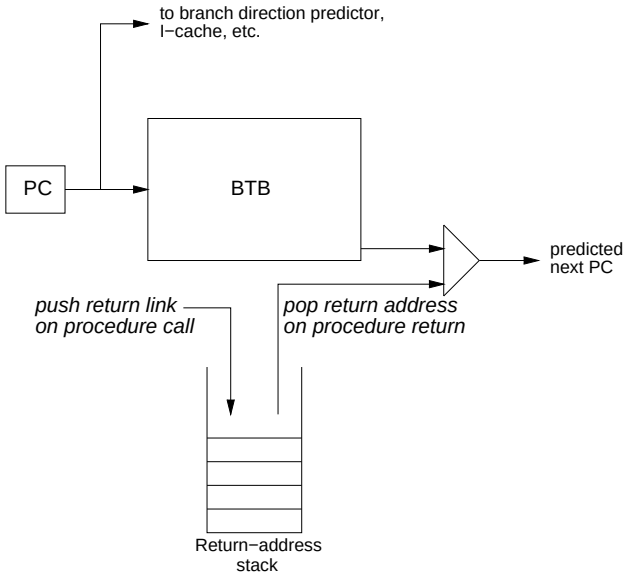


Figure 1. Basic operation of the return-address stack.

Some architectures do not include specific return instructions, instead relying on an indirect-jump instruction. Returns can still be easily distinguished if they use a specific general register that other indirect jumps avoid, an easy task for the compiler. Some legacy code, if built with an older compiler that did not make such a distinction, might suffer from poor return-address-stack behavior unless corrected with a binary-rewriting tool.

In a processor with no speculation, only overflow and unmatched call/return sequences corrupt the stack.[2] If the stack's prediction is treated as a hint, the stack needs no tags or valid bits. But a problem arises when speculative execution is permitted. After a branch misprediction, some number of cycles elapse before the branch condition resolves and the misprediction is detected. In the meantime, the processor speculatively fetches and executes further instructions which will be squashed. If any of the fetched instructions are calls or returns, they push or pop the return-address stack, because pushes and pops take place during fetch. These mis-speculated calls/returns will eventually be squashed, yet without additional state, the stack cannot be repaired—the stack just contains return addresses, with no way to determine which entries have been corrupted. A sequence of more mis-speculated pops than pushes or vice-versa throws the entire stack out of alignment, as Figure 2 illustrates. The `while` is incorrectly predicted to terminate. Subsequent returns and calls are mis-speculated and will eventually be squashed, but their effects remain in the stack even after the misprediction has been cleaned up. Correct returns then continue to mispredict, because each pop matches up with the wrong pushed value. If instead a matched number of mis-speculated pops and pushes occurs, the stack remains properly aligned, but with the wrong value in some entries. Note that because it takes some time for the branch predictor to be updated, the `while` might be mispredicted several times in a row, in which case this sequence of harmful effects happens each time.

The previous explanation assumed that pushes and pops take place during fetch. Updating the stack in the commit stage avoids corruption by mis-speculated instructions, but creates a different set of problems. Returns can be mispredicted even on a correct path, because an entire procedure can be in flight before the call at the beginning of the procedure gets a chance to commit and push its return address onto the stack. When the call has not yet pushed by the time the corresponding return is fetched, that return instruction reads a non-existent value from the stack. A further problem occurs when two or more returns occur in close succession. If the first one cannot pop the stack before the second one looks in the stack, the second one reads from the wrong location. The remaining discussion only considers repair mechanisms for stacks that are updated in the fetch stage.

2.2 Mis-speculation repair mechanisms

This paper considers two simple repair mechanisms. Simply saving the current top-of-stack pointer at the time of each branch prediction is sufficient to avoid losing stack alignment. It completely prevents corruption from a mis-speculated sequence of pops only or pushes only.

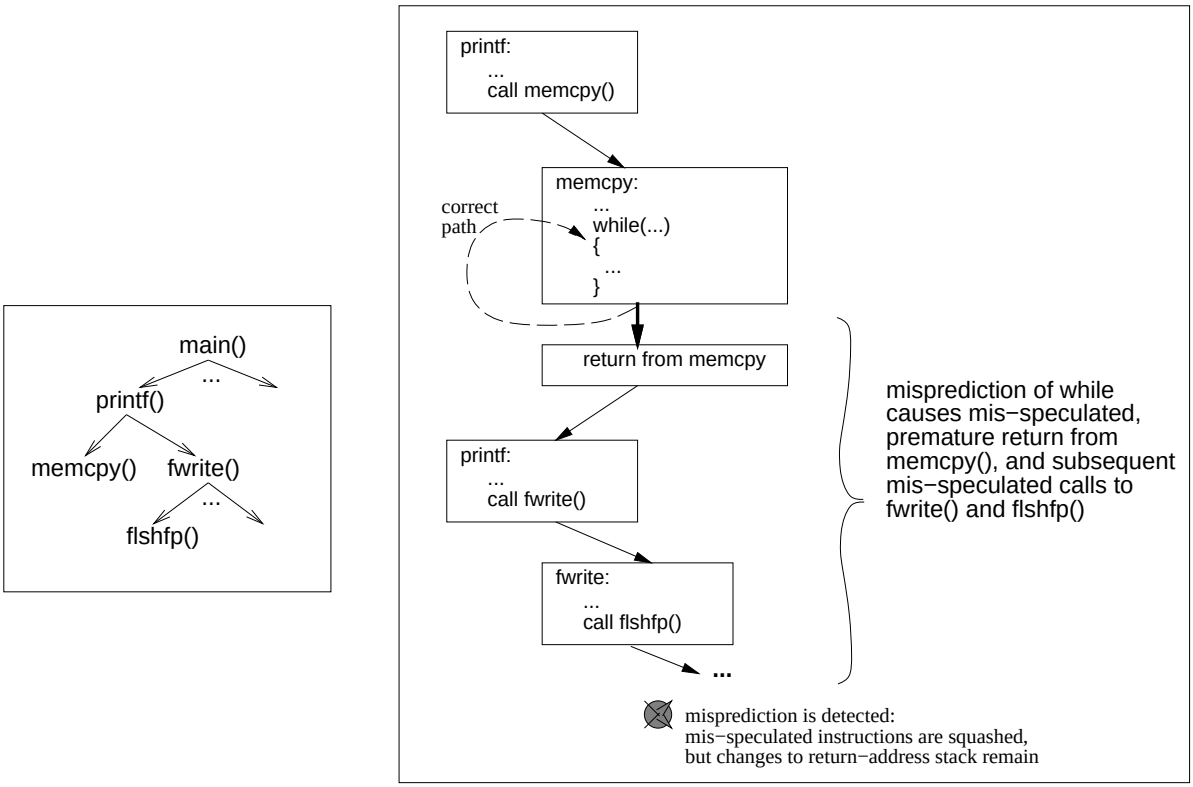
Saving the TOS pointer is reasonably cheap. Each time the processor speculates past a branch, it already saves some shadow state, such as the register-rename map, in case that branch is mispredicted. Saving the TOS pointer merely adds several bits per branch to this shadow state. The size of this shadow state is limited: in current processors, the shadow storage can accommodate at most a few in-flight branches—4 in the case of the MIPS R10000 [28], and 20 in the case of the 21264 [10].

Note that the shadow state may in fact be physically distributed: the shadow register maps near the register-mapping table, the shadow TOS pointers near the return-address stack, and so forth.

After a misprediction, the TOS pointer can be restored at the same time as the register map. No verification of individual stack entries is required. The stack itself remains simple: it still contains only the raw return addresses, without tags or valid bits.

A more aggressive scheme might also save the contents of the top stack entry along with the TOS pointer, requir-

[2] Such unmatched sequences might arise from events like `longjmp()/setjmp()` or exception handling; context switches also create the appearance of an unmatched sequence.



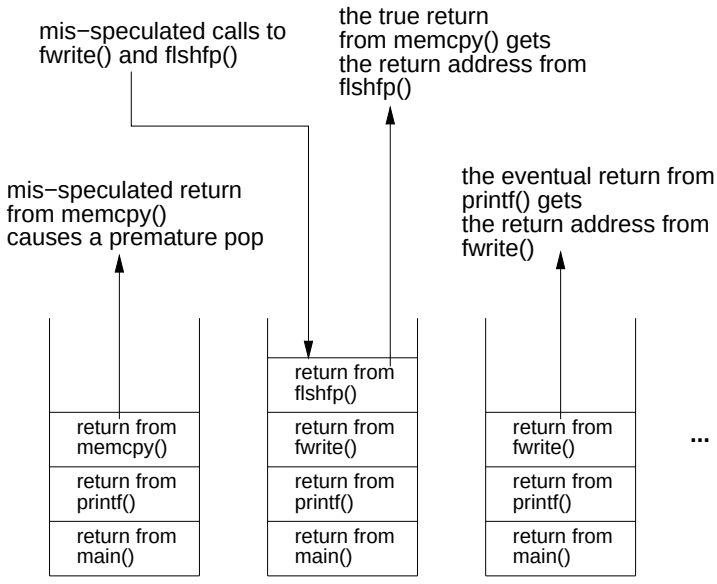


Figure 2. A case in which mis-speculation corrupts the stack.

A misprediction of the while in `memcpy()` causes a premature return and pop, and subsequent mis-speculated calls to `fwrite()` and `flshfp()`. Because the return-address stack is updated in fetch, these changes to the stack remain after the misprediction is caught. The correct return from `memcpy()` then pops the wrong return address, as do later function returns.

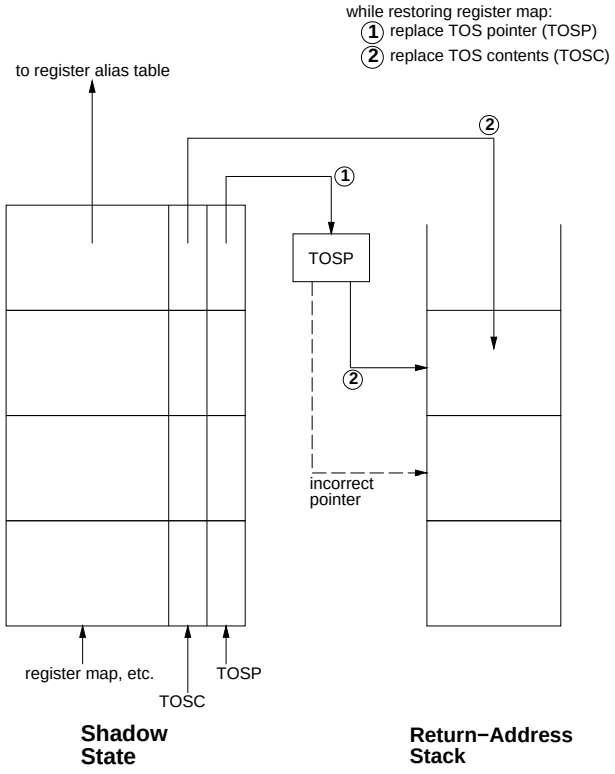


Figure 3. A high-level schematic of a return-address stack that restores both the TOS pointer and the TOS contents after detecting a misprediction.

ing one additional (and much larger: 18–30 bits instead of just 3–5 bits) field in each shadow-state entry. After a misprediction, the TOS pointer and then the TOS contents are restored. Figure 3 shows a high-level schematic for this technique. Now a mis-speculated sequence of at least two pops followed by at least one push is necessary to corrupt the stack.

One can, of course, save an arbitrary number of return-address-stack entries this way; the extreme would be to checkpoint the entire return-address stack each time a branch is predicted. Our results provide data for full-stack checkpointing as an upper limit. The sophisticated scheme proposed by Jourdan, *et al.* [19], can have the effect of checkpointing the full stack, but requires a larger number of stack entries than the methods proposed here because it preserves popped entries. Each of these stack entries also requires more space, because in addition to storing the return address, a stack entry in this scheme also contains a pointer to the next entry that should be popped. On the other hand, this approach stores only two small pointers in the shadow state, the TOS pointer and the "NEXT" pointer (which points to the next free stack entry).

The cost of the TOS-pointer and TOS-contents ("pointer & data") repair scheme can easily be compared to the Jourdan *et al.* self-checkpointing scheme. Suppose the stack contains n entries of 30 bits each (the lowest-order 2 bits can be discarded), and the shadow state supports up to m in-flight branches. Note that the stack need not actually hold full 30-bit return-addresses, since text sizes are rarely so large.

If the stack does hold full 30-bit return addresses, the pointer & data scheme requires $30\,n$ bits for the stack, and $(30 + \log n)\,m$ for the return-address-stack shadow state. The self-checkpointing scheme requires $(30 + \log n)\,n$ bits for the stack and $(2 \log n)\,m$ bits for the shadow state. With a 32-entry stack and 20-entry shadow state, like the 21264, the pointer & data scheme requires $960 + 700 = 1660$ bits, compared to $1120 + 200 = 1320$ bits for the self-checkpointing scheme—both quite small. The pointer & data approach also requires a wider bus between the stack and its shadow state (but these may be located close together): $30 + \log n$ bits vs. $2 \log n$ bits for the self-checkpointing approach's two pointers. Although this comparison has assumed the two approaches use the same size stack, the self-checkpointing scheme works poorly with very small stacks, because it does not have enough entries to store new return addresses while checkpointing older ones.

These proposed TOS-fixup mechanisms are simpler than tagging or using valid bits. Tagging must associate call and return addresses (and procedures might have many return instructions) and match tags on each access to determine whether to use the stack's prediction or the BTB's. Valid bits require identifiers for each in-flight branch; after a misprediction, these tags permit the processor to identify which stack entries have been corrupted.

3 Simulation Technique

3.1 Simulator

We use *HydraScalar*—an enhanced and multipath-capable version of SimpleScalar's [3] *sim-outorder*—for our experiments. SimpleScalar provides a toolbox of simulation components—like a branch-predictor module, a cache module, and a statistics-gathering module—as well as several simulators built from these components. Each simulator interprets executables compiled by *gcc* version 2.6.3, targeting a virtual instruction set that most closely resembles MIPS IV [29]. The simulators instantiate a virtual machine and can emulate the object program's execution in varying levels of detail.

HydraScalar simulates in detail an out-of-order execution, five-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, write-

Parameter	Value	Comments
Processor core		
Instruction-window size	64	RUU
Instruction register size	8 instructions	Buffer b/t fetch and decode
Decode/rename latency	4 cycles	Min time b/t fetch and issue
Fetch width	up to 4 instructions per cycle	Must be in same cache block
Decode width	up to 4 instructions per cycle	In-order
Issue width	up to 4 integer ops per cycle plus 2 FP ops per cycle	Out-of-order
Commit width	up to 4 instructions per cycle	In-order
Functional units	4 ALU/logical (1), 2 branch/shift(1), 1 integer multiply/divide (12/20), 1 FP add (4), 1 FP multiply (4), 1 FP divide/sqrt (16/33)	Latency appears in parentheses
Memory ports	any combination of 2 memory ops	
Branch prediction		
Branch predictor	hybrid: 4 K 2-bit selector, 12-bit history 1 K 3-bit local pred, 10-bit history 4 K 2-bit global pred, 12-bit history	
BTB (branch target buffer)	2048-entry, 2-way	updated only if taken
Mispredict penalty	2 cycles for misfetch, 7 cycles otherwise	
Memory hierarchy		
L1 data-cache	64 K, 2-way (LRU), 32 B blocks, 8 MSHRs, 1-cycle latency	
L1 instruction-cache	64 K, 2-way (LRU), 32 B blocks, 1 cycle latency	
L2	unified, 8 M, 4-way (LRU), 32 B blocks, 4 MSHRs, 12-cycle latency	
Memory	100 cycles	
L1→L2 bus	1 transaction every 2 cycles	
L2→mem bus	1 transaction every 8 cycles	

Table 1. Baseline configuration simulated by HydraScalar.

back, and commit. An arbitrary number of stages can be added between decode and issue to simulate time spent renaming and enqueuing instructions. Issue selects the oldest ready instructions for execution.

Cycle-by-cycle simulators like HydraScalar that do their own instruction fetching and functional simulation (as opposed to relying on direct execution to provide instructions for simulation) can accurately model mis-speculated paths. Like a real processor, HydraScalar checkpoints appropriate state as it encounters branches, and then proceeds down the predicted path, executing wrong-path instructions if appropriate. Upon detecting a mispredicted branch, wrong-path instructions are squashed, and recovery from the checkpointed state is straightforward. This modeling captures mis-speculation consequences like prefetching, cache pollution, and return-address-stack pollution, and is critical for accurately simulating multipath execution.

Table 1 summarizes our baseline model, loosely modeled after the reported configuration of an Alpha 21264 [15]. The conditional-branch direction-predictor is a McFarling-style, two-component hybrid branch predictor [26] that combines a 4K GAg (global-history) predictor with a 1K $\times$ 10 PAg (local-history) predictor [38]. For each prediction, a selector chooses the component most likely to be correct by consulting its own 4K table of saturating 2-bit counters, indexed by global history [6]. Since many entries in the direction predictor correspond to not-taken branches (or are simply idle), the BTB is decoupled [4], only allocating entries for taken branches. This permits the BTB to have fewer entries. The fetch engine fetches through not-taken branches and stops at taken branches.

SimpleScalar updates the branch-prediction state during the instruction-commit stage. This means there is a window of time, while a branch traverses the pipeline, during which its outcome is not available and the branch predictor uses slightly "stale" information. The prediction accuracies reported here are therefore not as high as those reported in trace-driven branch-prediction studies.

	Warmup Insts	Branches per Instruction				Branch Accuracies			
		All	Return	Indir	Cond	All	Return	Indir	Cond
go	926 M	0.144	0.011	0.002	0.111	0.758	0.432	0.629	0.754
m88ksim	26 M	0.212	0.018	0.003	0.162	0.931	0.704	0.251	0.954
gcc (cc1)	221 M	0.194	0.015	0.030	0.144	0.827	0.546	0.350	0.861
compress	2576 M	0.202	0.028	0.000	0.133	0.925	0.993	0.063	0.888
li (xlisp)	271 M	0.236	0.027	0.082	0.137	0.906	0.721	0.814	0.918
ijpeg	824 M	0.059	0.001	0.003	0.051	0.894	0.728	0.984	0.879
perl	601 M	0.193	0.019	0.077	0.129	0.893	0.664	0.332	0.937
vortex	2451 M	0.166	0.021	0.021	0.121	0.968	0.901	0.768	0.980

Table 2. Benchmark summary.

Statistics are taken only from the post-warmup, 50 M-committed-instruction simulation window, and use the baseline configuration in Table 1. "All" refers to all branches, whether conditional, direct-jump, indirect-jump, or return. "Indirect branches" here does not include returns. "Branch accuracy" refers to target-address prediction, except for the conditional-branch column, which presents direction-prediction accuracies.

Conditional-branch direction-mispredictions suffer at least a seven-cycle latency, because the branch condition does not resolve until the writeback stage. Conditional jumps for which the predicted direction is correct—and direct jumps—can still miss in the BTB (a *misfetch*), but a dedicated adder in the decode stage computes branch targets so that BTB misses can be detected early. A BTB miss still redirects the fetch engine, but detecting the misfetch in decode means the resulting bubble is only 2 cycles long. This is then the only penalty that misfetched branches experience. Indirect jumps—even though known to be taken—need to read the register file and these simulations assume that cannot be done from decode. Indirect-jump targets therefore cannot be computed by the dedicated adder in the decode stage, so if the BTB mispredicts the target, the error is only detected in the writeback stage.

HydraScalar simulates a 64-entry unified active list, issue queue, and rename register file—a *register update unit*, or RUU [32]. The architectural registers (32 each for integer and floating-point) are separate and updated on commit; renaming determines whether operands reside in the RUU or in architectural state. A 32-entry load-store queue (LSQ) disambiguates memory references: stores may only pass preceding memory references whose addresses are known not to conflict. Otherwise, issue selects the oldest ready instructions.

Finally, the cache hierarchy is a conventional two-level, non-blocking organization with separate first-level instruction and data caches. For these simulations, HydraScalar models a pipelined bus with a fixed fetch spacing.

3.2 Benchmarks

The SPEC integer benchmarks [33] are summarized in Table 2. Simulations use the provided reference inputs. All benchmarks are compiled using `gcc -O3 -funroll-loops` (`-O3` includes inlining). Simulations include all non-kernel behavior, such as library code. We omit the floating-point suite because those programs have excellent branch-prediction accuracies—stack corruption is not a problem—and call-return frequencies at least an order of magnitude lower than those found in SPECint programs.

Some benchmarks come with multiple reference inputs, in which case one has generally been chosen. For *go*, we chose a playing level of 50 and a 21x21 board with the `9stone21` input. For *m88ksim*, we used the `dhrystone` input; for *gcc*, `cccp.i`; for *ijpeg*, `vigo.ppm`; and for *perl*, we used the scrabble game. For *xlisp*, we ran the program with all the supplied LISP files as arguments.

We perform full-detail simulation for a representative, 50-million-instruction segment of the program that avoids the program's initial phases and any warmup effects. For each benchmark, we find the segment using a simple miss-rate simulator that measures the cache miss rate and branch misprediction rate for each 1-million-instruction interval, independent of the previous interval. Testing this segment's representativeness by running a subset of our experiments for 250 million instructions verifies that the chosen segments provide representative behavior in terms of return-address-stack behavior as well as branch prediction accuracy, cache performance, and overall IPC. A detailed examination of these time-series miss-rate results appears in [30], which discusses the choice of a simulation sample in more detail.

Cycle-level simulations are run in a fast mode to reach the chosen simulation window. In this fast mode no microarchitectural simulation takes place; only the caches and branch predictor are updated. Table 2 includes the length of the fast-mode ("warmup") phase for each benchmark, including 1 million instructions in which simulation runs in

	Prediction Accuracy				Speedup		
	no fixup	pointer only	pointer & data	complete fixup	pointer only	pointer & data	complete fixup
go	0.432	0.737	0.994	1.000	1.034	1.065	1.065
m88ksim	0.704	0.898	1.000	1.000	1.046	1.076	1.076
gcc	0.546	0.876	0.984	1.000	1.057	1.076	1.079
compress	0.993	0.997	1.000	1.000	1.001	1.002	1.002
xlisp	0.721	0.877	0.982	0.997	1.058	1.087	1.096
ijpeg	0.728	0.989	0.995	1.000	1.003	1.004	1.003
perl	0.664	0.928	0.963	0.990	1.078	1.087	1.100
vortex	0.901	0.993	0.999	1.000	1.043	1.045	1.046

Table 3. Return-address-stack fixup: prediction accuracy and speedup with a 32-entry stack.
Speedups are normalized to the no-fixup case.

full detail to prime other structures.

Our results do not include the effects of context switches, which leave a corrupted stack when the process resumes. One way to avoid the resulting mispredictions is to save the stack as part of the context, but this should be unnecessary. Of chief importance in obtaining good return-address-stack performance is avoiding frequent corruption near the top of the stack, where most activity takes place. Since return instructions are so frequent compared to context switches, context switches should have negligible impact.

4 Evaluation in a Conventional CPU

4.1 Repair mechanisms

We first evaluate the repair mechanisms described earlier: *(i)* no fixup, *(ii)* TOS-pointer-only fixup, *(iii)* pointer-and-data fixup (restoring both the TOS pointer and the TOS contents), and *(iv)* full stack checkpointing. All four schemes update the return-address stack in the same way during fetch; they differ only in their repair mechanism. Table 3 compares the return-address-stack hit rates and speedups for these mechanisms with a 32-entry stack.

Without repair, return-address-stack hit rates are mostly in the 60–75% range, with an average of 71%. Aggressive fixup—restoring both the TOS pointer and the TOS contents—helps dramatically. It nearly eliminates return mispredictions, with corresponding speedups of 4.5–8.7% (average = 5.5%). Two benchmarks, *compress* and *ijpeg*, are exceptions and do not benefit from fixup. *Compress* executes procedure calls about as often as other benchmarks (see Table 2), and in fact has a fairly high conditional-branch misprediction rate; its call/return sequences are nevertheless well-behaved. *Ijpeg* has comparatively few calls to begin with, so the hit rate of only 72.8% barely hurts performance. For all these benchmarks, there is little need for saving more than the top entry of the stack contents: sequences of two or more pops followed by one or more pushes are rare.

A significant fraction of the benefit from pointer & data fixup can be obtained by restoring only the TOS pointer. Pointer-only fixup brings most of the hit rates up to about the 90% range (*go* is the exception), reducing mispredictions by 50–93%. Except for *compress* and *ijpeg*, which do not need any fixup, this yields speedups of 4.3–7.8%. The success of simple, pointer-only fixup indicates that mispredictions are often caught before a pop can be followed by a push on a particular mis-speculated path. The gap between pointer-only and pointer & data fixup then indicates the frequency of pop-followed-by-push sequences, and the small gap between pointer & data fixup and full stack checkpointing indicates the rarity of more complex sequences like two pops followed by a push.

These data indicate little need for more sophisticated mechanisms, like tagging, valid bits, or full-stack checkpointing.

4.2 Size

The previous data assumed a fixed-size stack of 32 entries; this section evaluates the contribution of return-address-stack size. Because fixup can make such a substantial difference, Tables 4 and 5 show stack hit rate and speedup for a stack with pointer-and-data fixup, while Tables 6 and 7 show similar data, but for a stack using no fixup. The column for size 0 in Tables 4 and 6 show the prediction success of the BTB: return addresses are found there just over half the time, on average.

With pointer-and-data repair, performance increases with stack size until about 4–8 entries. *Gcc* and *xlisp* benefit marginally from a further increase to 16 entries. Some programs, notably *go* and *compress*, get a large boost in perfor-

	0	1	2	4	8	16	32	128
go	0.381	0.841	0.952	0.991	0.994	0.994	0.994	0.994
m88ksim	0.813	0.687	0.881	0.970	1.000	1.000	1.000	1.000
cc1	0.356	0.675	0.788	0.927	0.979	0.983	0.984	0.984
compress	0.510	0.976	1.000	1.000	1.000	1.000	1.000	1.000
xlisp	0.683	0.741	0.834	0.923	0.956	0.976	0.982	0.984
ijpeg	0.793	0.911	0.971	0.996	0.998	0.998	0.998	0.998
perl	0.579	0.509	0.709	0.903	0.959	0.963	0.963	0.963
vortex	0.475	0.431	0.700	0.896	0.994	0.999	0.999	0.999

Table 4. Return-address-stack prediction accuracy as a function of stack size.
With pointer-and-data fixup. For size 0, the BTB is the source of return-address predictions.

	0	1	2	4	8	16	32	128
go	1.000	1.047	1.061	1.066	1.066	1.066	1.066	1.066
m88ksim	1.000	0.987	1.035	1.057	1.059	1.059	1.059	1.059
cc1	1.000	1.051	1.071	1.097	1.108	1.109	1.109	1.109
compress	1.000	1.145	1.154	1.154	1.154	1.154	1.154	1.154
xlisp	1.000	1.011	1.042	1.082	1.093	1.101	1.104	1.105
ijpeg	1.000	1.002	1.002	1.002	1.002	1.003	1.003	1.003
perl	1.000	0.979	1.038	1.097	1.117	1.120	1.120	1.120
vortex	1.000	0.986	1.073	1.155	1.212	1.214	1.214	1.214

Table 5. Speedup as a function of return-address-stack size.
With pointer-and-data fixup. Speedups are compared to having no stack, getting return addresses from the BTB.

	0	1	2	4	8	16	32	128
go	0.381	0.597	0.494	0.444	0.434	0.433	0.432	0.433
m88ksim	0.813	0.607	0.695	0.703	0.702	0.703	0.704	0.703
cc1	0.356	0.556	0.570	0.551	0.546	0.545	0.546	0.545
compress	0.510	0.974	0.993	0.993	0.993	0.993	0.993	0.993
xlisp	0.683	0.635	0.688	0.726	0.719	0.693	0.721	0.711
ijpeg	0.793	0.874	0.767	0.751	0.743	0.742	0.728	0.700
perl	0.579	0.471	0.588	0.663	0.662	0.663	0.664	0.663
vortex	0.475	0.426	0.682	0.849	0.901	0.901	0.901	0.901

Table 6. Return-address-stack prediction accuracy as a function of stack size.
With no fixup. With size 0, the BTB is the source of return-address predictions.

	0	1	2	4	8	16	32	128
go	1.000	1.018	1.008	1.003	1.002	1.002	1.002	1.002
m88ksim	1.000	0.966	0.983	0.985	0.985	0.985	0.985	0.985
cc1	1.000	1.031	1.034	1.031	1.031	1.031	1.031	1.031
compress	1.000	1.144	1.151	1.151	1.151	1.151	1.151	1.151
xlisp	1.000	0.984	1.002	1.017	1.015	1.003	1.015	1.011
ijpeg	1.000	1.001	1.000	0.999	1.000	1.000	0.999	0.999
perl	1.000	0.968	1.006	1.027	1.026	1.027	1.026	1.027
vortex	1.000	0.984	1.066	1.134	1.162	1.162	1.162	1.162

Table 7. Speedup as a function of return-address-stack size.
With no fixup. Speedups are compared to having no stack, getting return addresses from the BTB.

mance from just a single stack entry. Others perform *worse* with a 1-entry stack than with no stack, which results from underflow: with a 1-entry stack, the stack only contains the most recent return address. A sequence of 2 returns pops an empty stack and almost certainly mispredicts the second return.

The presence of some sort of stack is important: adding a stack produces speedups of 6–15% (except for *ijpeg*, which has too few returns to matter). But without some repair mechanism, mis-speculation corrupts the stack so frequently that most of this gain cannot be realized; Tables 6 and 7 show that without fixup, performance is insensitive to the presence of a stack. Compared to a stack that omits fixup, only *compress* and *vortex* would be significantly penalized by just using the BTB.

4.3 Stack overflow and underflow

Our simulations have permitted the return-address stack to both over- and under-flow. *Overflow* occurs when a call is executed and the stack is full, either because of stack corruption or because there are more calls in progress than stack entries. The stack "wraps around," and as a result, the push overwrites the oldest stack entry. If a sequence of too many calls has caused overflow, a later return causes *underflow* by popping an empty stack, and the return receives an invalid result. This may be harmless if most call-return activity occurs near the top of the stack (for example, the overwritten entry might just be the call to `main()` from `crt0.o`). Underflow can also occur if mis-speculated pops, context switches, and the like destroy some values on the return-address stack and the stack later becomes prematurely empty.

An occupancy counter can detect instances of both overflow and underflow, but preventing overflow events is in fact undesirable. It saves the return address of the oldest calls in progress at the expense of newer information. Since the damage caused by overflow does not appear until a later underflow, the occupancy information can be used to detect when the stack is empty and in these cases the BTB can try to supply the return-address prediction. Unfortunately, in order to be useful, the occupancy-counter must be checkpointed with each branch prediction in the same manner as the TOS pointer and the TOS contents. Since over- and underflow are mainly a problem with small stacks, it may be better to just make the stack deeper.

5 Evaluation in a Multipath Processor

Multipath execution [1, 21, 22, 35, 36], rather than predicting which direction a branch takes, forks and executes speculatively down both paths. Once the branch resolves, the incorrect path is squashed. Forked paths can in turn fork, but path contexts and execution resources are limited, so forking should be done selectively, on branches more likely to mispredict. This choice of when to fork can be made statically or dynamically [1, 17, 14, 35].

Although multipath execution requires substantial extra hardware—per-path contexts, more fetch, rename, and issue bandwidth, and a larger instruction window—this extra hardware overlaps substantially with other likely directions for future microprocessors. These include clustered approaches [12], multithreaded processors [23], and particularly simultaneous multithreading (SMT) [34].

This paper mentions multipath execution because the design of the return-address stack proves critical to multipath performance. A single, unified stack does not function properly in a multipath processor. With concurrent paths simultaneously modifying the stack, corruption is almost certain, even with full-stack checkpointing. Figure 4 shows how this corruption might happen. For example, after a fork, both paths might encounter the same return instruction. One of these is legitimate, while the other occurs on a mis-speculated path, but both pop the stack. The second pop results in a return to the wrong subroutine level, and the stack has now been corrupted. In fact, the results presented in this section suggest that with multipath execution, corruption is severe enough that simply omitting the return-address stack and using the BTB leaves performance almost unchanged for most programs.

The best solution we have found simply gives each path a private copy of the return-address stack. Multipath execution already requires path contexts to record various state about each path: PC, shadow register maps, etc. The return-address stack merely adds an additional element in the path context, and something that multithreading also requires [16]. Copying the stack need not take place in a single cycle. If the new stack only receives a correct TOS pointer in the first cycle, it can already receive pushes from the new path. Pops, although returning an invalid address, will move the pointer correctly so that as stack entries are gradually copied, later pops can return correct values. Copying should of course start with the top of the stack.

An alternative approach uses a unified stack, but only allows one path to access the stack. Any other active paths must use the BTB. A variety of heuristics are possible for choosing the privileged path. This work uses the predicted path—the path that would have been followed if no forking occurred—as the one allowed to access the stack. This is still the path most likely to be correct, but the drawback to this approach is that during a misprediction, the wrong path has the privilege; after the misprediction resolves, the newly-recognized predicted path may find a corrupted stack.

Tables 8 and 9 compare the performance of four configurations: *(i)* no stack, *(ii)* a unified stack, *(iii)* a unified stack

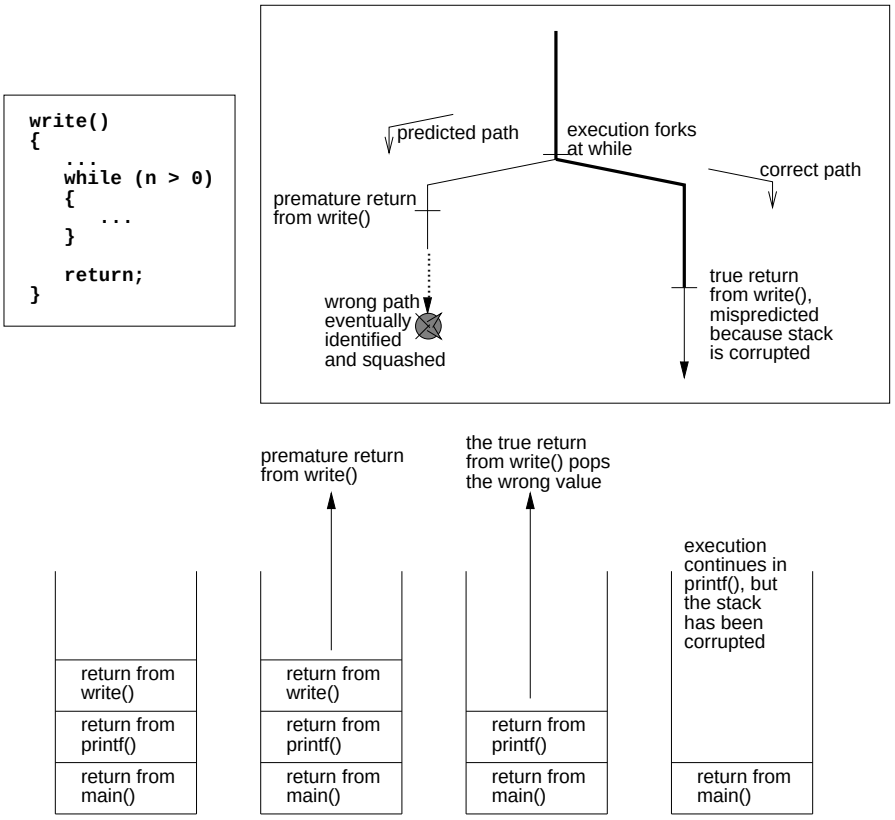


Figure 4. An example of how multipath execution can corrupt a unified return-address stack.

that only the predicted path can access, and *(iv)* per-path stacks. In these results, the per-path stacks are copied in a single cycle. Table 8 presents return-prediction accuracies; note that for the no-stack case, this accuracy presents the return-address hit-rate in the BTB. Performance is normalized to the conventional, unified-stack case. In all configurations, the stacks implement "pointer & data" fixup after mispredictions. We test both 2- and 4-path processors: the 2-path processor can fetch 8 instructions per cycle, and the 4-path processor can fetch 16 instructions per cycle. To accommodate this greater bandwidth, the downstream bandwidth has been increased correspondingly. The RUU has also been enlarged to 256 entries[3] the LSQ to 128 entries, and the L1 caches to 256K.

A unified stack achieves poor return hit rates, despite the use of pointer & data fixup. For half the benchmarks, no stack is better than a unified stack, substantially so for *m88ksim* and *xlisp*. *Compress* and *vortex*, on the other hand, suffer badly if returns are only predicted from the BTB. None of these choices has any impact on *ijpeg*, so we do not discuss it further.

Using per-path stacks provides substantial gains for all the benchmarks. Hit rates return to near 100% levels (always reaching 100% with full-stack checkpointing, whose numbers are not presented here), and speedups range from 7% with *go* to 26% with *xlisp*. *Go's* speedup is low compared to its large improvement in hit rate, but *go* has fewer returns than most of the other programs, and time spent on return-address misses is in any case dwarfed by its high number of conditional-branch mispredictions.

A unified stack accessed only by the predicted path is a clear improvement over a simple unified stack, but is substantially inferior to per-path stacks. With 2 paths, the improvement is on average half that seen with per-path stacks. With 4 paths, hit rates decline compared to the 2-path case, because the predicted path is less likely to be the correct one; speedups are sometimes higher for 4 paths than for 2 paths, because hit rates in the conventional stack decline too. Return mispredictions also tend to matter more in the 4-path case, because 4-way multipath is more effective at eliminating conditional-branch mispredictions. Another disadvantage to this approach is that, as further research leads to more selective forking, predicted-path schemes will become less effective.

[3]Because wrong paths intermingle in the RUU with correct paths, squashing must selectively invalidate RUU entries; see [1], and [21] for the mechanism. The RUU is essentially a FIFO buffer, so these now-empty entries must still propagate to the front and be "retired".

	No Stack (use BTB)		Unified Stack		Unified Stack Pred-Access Only		Per-Path Stacks	
	2 paths	4 paths	2 paths	4 paths	2 paths	4 paths	2 paths	4 paths
go	0.380	0.379	0.478	0.326	0.588	0.428	0.979	0.984
m88ksim	0.815	0.821	0.701	0.566	0.856	0.769	0.986	0.984
cc1	0.359	0.358	0.549	0.472	0.678	0.535	0.973	0.968
compress	0.511	0.533	0.884	0.788	0.972	0.901	1.000	1.000
xlisp	0.681	0.683	0.604	0.566	0.791	0.698	0.976	0.986
ijpeg	0.793	0.793	0.789	0.771	0.856	0.846	0.982	0.981
perl	0.579	0.579	0.525	0.495	0.806	0.712	0.958	0.950
vortex	0.475	0.475	0.827	0.807	0.895	0.898	0.998	0.998

Table 8. Return prediction accuracies for different stack organizations under multipath execution.
Accuracies are for the correct path (committed return instructions) only.

	No Stack (use BTB)		Unified Stack		Unified Stack Pred-Access Only		Per-Path Stacks	
	2 paths	4 paths	2 paths	4 paths	2 paths	4 paths	2 paths	4 paths
go	1.003	1.022	1.000	1.000	1.013	1.013	1.057	1.069
m88ksim	1.051	1.124	1.000	1.000	1.068	1.093	1.131	1.214
cc1	0.959	0.973	1.000	1.000	1.030	1.015	1.110	1.142
compress	0.858	0.887	1.000	1.000	1.027	1.029	1.038	1.080
xlisp	1.039	1.054	1.000	1.000	1.096	1.062	1.205	1.263
ijpeg	1.000	1.001	1.000	1.000	1.001	1.002	1.004	1.005
perl	1.028	1.042	1.000	1.000	1.131	1.102	1.214	1.242
vortex	0.752	0.730	1.000	1.000	1.054	1.098	1.164	1.237

Table 9. Relative performance for different stack organizations under multipath execution.
2-path results are normalized to the 2-path, unified-stack case, and 4-path results to the 4-path, unified-stack case.

6 Conclusions

This paper evaluates return-address-stack design in light of potential corruption by mis-speculated instructions, a problem raised by Jourdan *et al.* [19]. Without a return-address stack, return addresses are found in the BTB only a little over half the time (Table 4). A well-designed stack can eliminate most or all of these return mispredictions, producing speedups of up to 15% compared to using only a BTB. But because modern processors speculate so aggressively, an effective return-address stack must incorporate a repair mechanism. Without such a mechanism, a stack provides little benefit for most SPECint programs in our processor model.

Instead of trying to efficiently checkpoint the entire return-address-stack when speculating past a branch, a simple repair mechanism that restores the top-of-stack pointer and just one entry from the top of the stack turns out to eliminate almost all effects of stack corruption. Merely restoring the pointer performs fairly well, too.

Corruption becomes an even bigger problem with multipath execution: even with the repair scheme we describe, the competing paths so badly corrupt the stack that performance can be worse than if the stack is omitted and return predictions come from the BTB. The solution we suggest is to give each path its own return-address stack, copying the stack each time a new path is forked. This copy need not happen in a single cycle. A set of per-path, 32-entry return-address stacks improves performance in a multipath processor by up to 26%.

We have only evaluated SPECint95 benchmarks, which are written in C. C++ programs or functional-language programs (Scheme, ML, etc.) sometimes have many short functions, in which case stack performance becomes even more important. Call/return and indirect-jump prediction for these environments are an important area of future work.

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

Kai Li and the referees for helpful feedback in preparing this paper.

References

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multi-path execution: Opportunities and limits. In *Proc. 12th ICS*, pages 101–08, July 1998.

[2] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proc. HPCA-3*, pages 84–93, Feb. 1997.

[3] 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.

[4] B. Calder and D. Grunwald. Fast & accurate instruction fetch and branch prediction. In *Proc. ISCA-21*, pages 2–11, May 1994.

[5] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proc. POPL-21*, pages 397–408, Jan. 1994.

[6] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proc. Micro-28*, pages 252–57, Dec. 1995.

[7] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *Proc. ISCA-24*, pages 274–83, June 1997.

[8] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, Apr. 1995.

[9] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proc. ISCA-25*, pages 167–78, July 1998.

[10] J. Emer. Personal communication, Mar. 1998.

[11] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proc. ISCA-25*, pages 52–61, Jun. 1998.

[12] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proc. Micro-30*, pages 149–59, Dec. 1997.

[13] A. Glew. Personal communication, Mar. 1998.

[14] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proc. ISCA-25*, pages 122–31, June 1998.

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

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

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

[18] Q. Jacobsen, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proc. Micro-30*, pages 14–23, Dec. 1997.

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

[20] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proc. ISCA-18*, pages 34–41, May 1991.

[21] A. Klauser, V. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath Architecture. In *Proc. ISCA-25*, pages 250–59, July 1998.

[22] R. Kol and R. Ginosaur. Kin: A high performance asynchronous processor architecture. In *Proc. 12th ICS*, pages 433–440, July 1998.

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

[24] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proc. Micro-30*, pages 4–13, Dec. 1997.

[25] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Proc. Micro-29*, pages 153–164, Dec. 1996.

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

[27] S. McMahan, Cyrix Corp. Branch processing unit with a return stack including repair using pointers from different pipe stages. U.S. Patent No. 5,706,491, Jan., 1998.

[28] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, Jun. 1995. Version 1.0.

[29] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[30] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. A quantitative evaluation of branch prediction's impact on instruction-window size and cache size. Tech. Report TR-578-98, Princeton Dept. of Comp. Sci., April 1998.

[31] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA-8*, pages 135–48, May 1981.

[32] G. S. Sohi and A. S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proc. ISCA-14*, pages 27–34, June 1987.

[33] The Standard Performance Evaluation Corporation. WWW Site. http://www.specbench.org, Dec. 1996.

[34] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. ISCA-22*, pages 392–403, June 1995.

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

[36] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proc. ISCA-25*, pages 238–49, July 1998.

[37] C. F. Webb. Subroutine call/return stack. *IBM Tech. Disc. Bulletin*, 30(11), Apr. 1988.

[38] T.-Y. Yeh 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.

[39] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proc. ISCA-22*, pages 276–86, June 1995.