to appear in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) 1998

Dynamic Hammock Predication for Non-predicated Instruction Set Architectures

Artur Klauser[§] Todd Austin[†] Dirk Grunwald[§] Brad Calder[‡]

[§] University of Colorado at Boulder
Department of Computer Science

[†] Intel
Microcomputer Research Lab

[‡] University of California, San Diego
Department of Computer Science and Engineering

Abstract

Conventional speculative architectures use branch prediction to evaluate the most likely execution path during program execution. However, certain branches are difficult to predict. One solution to this problem is to evaluate both *paths following such a conditional branch. Predicated execution can be used to implement this form of multi-path execution. Predicated architectures fetch and issue instructions that have associated predicates. These predicates indicate if the instruction should commit its result. Predicating a branch reduces the number of branches executed, eliminating the chance of branch misprediction at the cost of executing additional instructions.*

In this paper, we propose a restricted form of multi-path execution called Dynamic Predication *for architectures with little or no support for predicated instructions in their instruction set. Dynamic predication dynamically predicates instruction sequences in the form of a* branch hammock*, concurrently executing both paths of the branch. A branch hammock is a short forward branch that spans a few instructions in the form of an* `if-then` *or* `if-then-else` *construct. We mark these and other constructs in the executable. When the decode stage detects such a sequence, it passes a predicated instruction sequence to a dynamically scheduled execution core. Our results show that dynamic predication can accrue speedups of up to 13%.*

1 Introduction

Current processors already issue 4 instructions per cycle, and future designs will be able to issue 8 or more instructions per cycle. However, increasing the issue width of the processor beyond 4 instructions will have decreasing benefits unless the compiler and architecture technology is improved. Studies on current processors, which can issue 4 instructions per cycle, show that on average only 1 to 2 instructions are issued per cycle [6, 21]. Therefore, 50% to 75% of the processor's potential performance is not being utilized. A main contributor to this performance degradation is the small size of basic blocks and high branch misprediction penalties. Even with new branch prediction architectures, some branches are still very hard to predict. For example, the SPECint95 program *go* has only 80% branch prediction accuracy using the latest branch prediction architectures [19].

Predicated execution is an important part of future architecture design. Predication allows the removal of conditional branches from the instruction stream through conditional execution of instructions. Predicated architectures fetch and issue instructions that have predicates indicating if the instruction should commit its result.

This paper describes a mechanism for removing branch penalties by implementing a restricted form of multi-path execution on dynamically scheduled architectures. We call this *Dynamic Predication* (DP) because we do not assume support for predication in the instruction set architecture (ISA). Instead, we identify branches that can benefit from predicated execution using information provided by compiler or link-time transformations. During execution, dynamic predication converts the instruction sequences for both paths of these branches into predicated form. Internally, the processor instructions use full predication (*i.e.*, all instructions can be predicated). The branch is then eliminated from the instruction stream and instructions from both paths are concurrently executed. The conversion to predicated form occurs while the program executes; hence the name Dynamic Predication.

In this paper, we concentrate on dynamic predication for simple *branch hammocks* [7], or branches that have a clear fork-join form with no nested hammocks. Adding the ability to predicate hammocks to a dynamically scheduled architecture helps (1) eliminate mispredicted branches, (2) reduce capacity demands on resources for branch prediction, and (3) improve instruction fetching by increasing the number of instructions between branches.

1.1 Dynamic Predication for Branch Hammocks

Figure 1 shows a code fragment corresponding to an "if-then-else" statement. The example has a single conditional branch at instruction (e) and a join point at (k). As shown in Figure 1, we break a *branch hammock* into four "contexts" that we use to discuss dynamic predication throughout this paper. The *fork-context* occurs up to the point of the original conditional branch. The *then-context* and optional *else-context* each contain half of the branch hammock. The *join-context* is the code following the branch hammock.

Since dynamic predication only converts branch hammocks into predicated form, it is more restrictive than a fully predicated ISA with an aggressive predicated compiler. In addition, if the hammock code is not laid-out as one consecutive block of instructions, dynamic predication is not applied, since the advantage of consecutive fetch is lost. Despite these limitations, we show that dynamic predication can improve the performance of a dynamically-scheduled architecture with a conventional ISA.

There are three problems to be solved in dynamic predication: identifying eligible branch hammocks, deciding whether to use predication to execute the branch hammock, and maintaining the processor state in the face of dynamic predication.

In this work, we assume the conditional branch starting a predicatable hammock (i.e. instruction (e) in Figure 1) would be marked by a compiler or binary instrumentation tool.

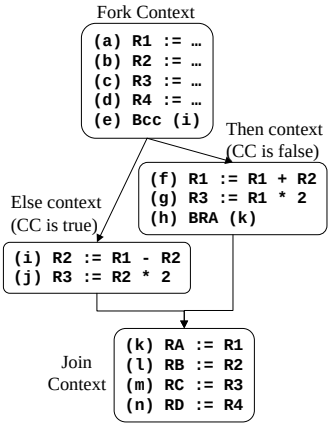

Figure 1. Branch Hammocks

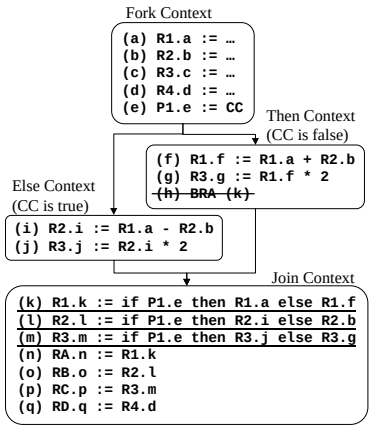

Figure 2. Translated Branch Hammocks. Rx.y: logical register x, physical register y

The decision to evaluate both paths can be done using off-line profiles or an on-line dynamic confidence estimator. If an off-line profile is used, the branch is always executed using predication. In the dynamic method, the branch is executed using predication only if a confidence estimator reports a low confidence in the branch prediction. We use a dynamic confidence estimator similar to the design of Jacobsen *et al.* [10]. Clearly, dynamic predication will have the largest influence for branches with little predictability and few instructions. If a branch can be easily predicted, then speculation is more efficient than predication. However, wide-issue processors are likely to have fetched the instructions on both branch paths, and predication can be useful even for highly predictable branches.

Dynamic predication evaluates both sides of a branch, and places additional pressure on execution resources. In Figure 1, registers R1 and R2 are each modified on only one half of the hammock, whereas R3 is modified on both halves. Register R4 is not modified by either path. Since each path redefines some portion of the register name space, and instructions from both paths are issued to the same processor, some mechanism must be used to distinguish the state for each path. We inject instructions from both predicated paths into the same pipeline. We use a labeling mechanism, described later, to identify the instructions and registers from the different paths. We remove the conditional branch and replace it with a predicate definition instruction, rename the registers on each path of the branch to maintain independent execution semantics and inject conditional move (cmove) operations to reconcile predicated definitions into a single context following the join point.

Figure 2 shows the code that is actually dispatched by our processor front-end. The conditional branch has been replaced by an assignment to a predicate register (P1). Registers defined by either path have been renamed. Cmoves, instructions (k), (l) and (m), have been dispatched prior to the join point. The cmoves merge the results from each branch path, providing a unique physical register for subsequent instructions. For example, (l) renames R2 to physical register *l*, using either the original value of R2 from (b) or the value computed at (i).

Alternatively, rather than using conditional moves, the processor front-end could cease issuing instructions when it reaches a join point and wait for the outstanding predicate computation (e) to resolve. Cmoves consume additional instruction window slots and register names. However, using cmoves does not stall the decoder, and instructions not dependent on the predicate can continue to issue - e.g., (q) depends on the value of R4, which is not redefined in the branch hammock, and can be issued once (d) completes.

2 Implementation of Dynamic Predication

To support dynamic predication, the pipeline must be extended to identify hammocks, predicate the instructions within hammocks, and support the correct execution of instructions within the created fork, then, and else contexts. In addition, this support must coexist harmoniously with general branch speculation. In the following text, we detail the enhancements we made to our baseline out-of-order issue processor pipeline. An overview of the pipeline extensions and instruction re-order buffer additions is shown in Figure 3.

2.1 Decoder Support

2.1.1 Dynamic Hammock Predication

Once the decoder determines that a hammock should be predicated, it indicates this in the re-order buffer entries for the predicated instructions. In addition, a predication context tag (described in Section 2.4) is assigned to the hammock instructions along with a predicate value. The predicate value for the predicate region, *i.e.*, true or false, is compared against the resulting predicate in the context tag when committing the instructions. If the value of the context tag is equal to the predicate value then the instruction is committed, otherwise it is squashed.

The decoder uses the branch target of a hammock branch to determine when the then, else, and join points are encountered, as previously described. When the else point is encountered, the predicate value stored with the instructions is inverted. Also, note that the unconditional branch terminating the then-path is not inserted into the instruction window, since its effect is subsumed by predication. When the join point is encountered, predication of instructions is terminated until the next hammock.

2.2 Renamer Support

The renamer must be extended to correctly implement register communication in the presence of multiple predicated execution contexts. A predicated definition of a register must only be visible in the same predicated context, or in later contexts after the join point, after the predicate has resolved to be true. The renamer must be able to accomplish this task without the knowledge of the predicate value, since stalling would be tantamount to stopping instruction fetch until the branch resolves.

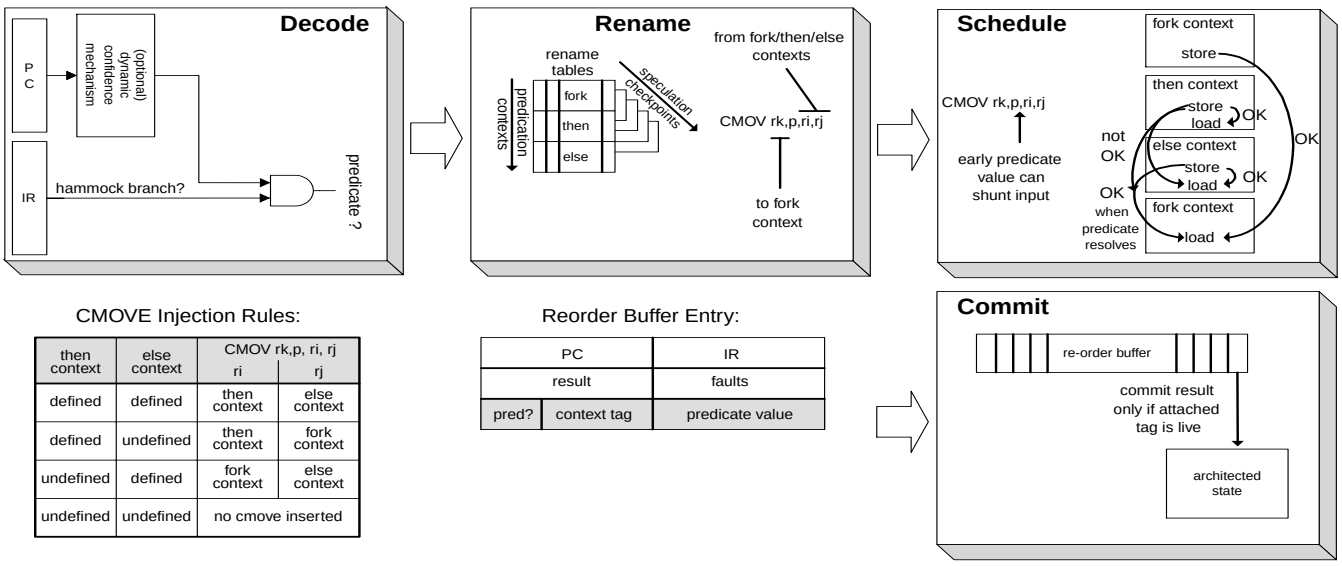

Figure 3. Pipeline Overview. Shown are the additions made to the baseline pipeline to support dynamic predication. The fields shown in gray are added to the instruction re-order buffer entries.

The rename table is extended to include three physical register definitions per logical register, as shown in the rename stage of Figure 3. The fork entry represents the definition before encountering the hammock branch. The then and else entries represent the definitions on the then- and else-paths of the hammock, and are only used while predicating a hammock. Each access to the register mapping table produces mappings for all three contexts. The context of the instruction determines which mapping is actually used. If the register is defined under the current predicate, the predicate entry is used. If not, the previous definition from the fork context is used. The predicated definitions are all cleared at the beginning of a predicate region. At the end of the predicate region, all predicated definitions are combined into a single physical storage location using physical cmove operations. Note that physical cmoves are different from logical cmoves which are found in some ISAs. A physical cmove addresses three physical registers, 2 source and 1 destination, which all correspond to the same logical register but were generated to accomodate predicated multi-path execution. The renamer injects a cmove for each logical register that was defined in a predicated context, according to the rules stated in Figure 3. Each injected cmove takes up one dispatch slot when it is put into the instruction window, which delays the dispatch of operations fetched after the join point. Dynamic expansion of ISA instructions into micro-operations is a proven technique used e.g. in the PentiumPro and K6 processor [4] implementations. Our technique is much simpler than this full translation of each instruction.

2.3 Scheduler Support

The register scheduling logic, for the most part, is unchanged. Using the renaming mechanism described in the previous section, instructions may begin execution as soon as their operands are ready. Predicated instructions do not have to wait for their predicates to be resolved, since unneeded results will be discarded by the cmove operation injected by the renamer. If the predicate arrives before both of the *use* values for a cmove instruction are calculated, the cmove can complete as soon as the value corresponding to the correct predicate context is produced.

The memory scheduler must be extended to operate correctly in the presence of multiple predicated and un-predicated contexts as shown in Figure 3. Data flow, through memory, cannot occur between disjoint predicate regions, and predicated definitions communicated through memory must be gated by their predicate values. To enforce this restriction, we added context tags to each entry of the load/store queue (context tags are detailed in Section 2.4). A reserved context identifier is assigned to loads and stores executed in un-predicated regions of code. Accordingly, a store in the un-predicated context may forward to any following load, and a store in a predicated context may forward to a load in its own context or in the un-predicated context once the store's predicate value has been verified to be true.

It is interesting to note that dynamic scheduling becomes much more challenging in the presence of generalized compiler-based predication (e.g., as in [18]). If individual instructions can be decorated with arbitrary predicates, the dynamic scheduler must assume that any two instructions that share a definition are dependent unless it can "prove" that their unresolved predicate definitions do not imply each other.

Our predication support greatly simplifies the dynamic scheduler's task by restricting predicates to sequential blocks of logically disjoint code. Thus the dynamic scheduler can trivially determine that instructions outside of the predicate region will always execute independent of the predicate values, instructions within the same predicate region will execute together independent of the predicates, and instructions in the "then" and "else" contexts never execute together and thus are independent.

2.4 Pipeline Recovery Support

Support for predicated execution requires a more selective pipeline recovery mechanism than those typically found in modern microprocessors. Most modern processors employ an N-level speculation recovery support, where a processor checkpoints the architected state of the machine at up to N precise instruction points, typically at speculated branches. Recovery is initiated by rolling back the state of the machine to a checkpoint. This discards all

instructions that were fetched after the restored checkpoint. In the presence of hammock predication, it is desirable to become more selective when killing instructions. Instead of a mechanism that squashes *all* instructions past a certain point, we use a mechanism that *deactivates* a range of instructions. This allows incorrect predicate regions that are embedded within un-predicated code to be "killed" without throwing away useful work in later instructions.

To implement this instruction deactivation support, we use an instruction tagging mechanism, where a unique tag is assigned to each predicated region of code. This tag is stored with the instructions within the predicate region in the instruction re-order buffer entry, as show in Figure 3. The tags are held in a queue, and are assigned to hammocks by the decoder as they are encountered and predicated. The instructions themselves hold the tag under which they execute, and the predicate value of their context, *i.e.*, predicate-true or -false. Tags are recovered for reuse when the last instruction in the predicate region assigned to that tag has retired. When a predicated branch condition is evaluated, instructions on the incorrect path associated with the tag are deactivated. The tag value, and its predicate result, are broadcast to all instructions in the re-order buffer. All instructions with a matching tag and opposite predicate value are marked as *inactive*. Once deactivated, an instruction will not commit its results to the architected state at retirement, nor will it declare any fault conditions it has detected, as shown in the commit stage of Figure 3. The scheduler does not issue inactive instructions; hence, once deactivated they do not use execution bandwidth. Since they have to be retired (but not committed), however, they still do consume retirement bandwidth.

In addition to the instruction tagging mechanism, the pipeline also supports the normal branch speculation recovery mechanism for branch speculation. There is a fine distinction between predication and speculation recovery support. With speculation support, the recovery mechanism checkpoints the state of the machine, and rolls back to the checkpoint when it recovers from mis-speculation. With predication, however, there is no need to roll back the state of the machine. The data flow accommodates the control conditions through the use of cmoves and it is only necessary for the recovery mechanism to omit committing results from predicated instructions on the incorrect path. Since there is no rollback, there is no need to checkpoint state for predicated branches. Since additional speculation is not allowed within predicated regions, the speculation checkpoints do not contain any predicated state; the predication state is cleared when a speculation checkpoint is restored.

Our architecture uses precise exceptions. If an instruction incurs an exception during execution, the exception condition is stored instead of the result. The exception is handled at instruction commit. Exceptions of instructions in predicated paths can be handled in the normal way. At the time when this exception is handled, the architecture can restore the same state as would be restored after speculating the hammock fork branch. No extra predicated state needs to be considered by exception handlers. Also, when the execution is restarted after exception handling, it is always restarted in non-predicated and non-speculative mode, exactly like in a normal speculative architecture.

3 Prior Work

In order to increase instruction level parallelism and reduce the number of branches in a program, *predicated execution* has been proposed as an alternative for conditional code sequences. Full predication was used in the Cydra 5 [18] architecture. Mahlke *et al.* [15] proposed *hyperblocks*, or superblock scheduling extended to support predicated architectures. Tyson [23] and Mahlke *et al* [13] studied the potential benefits of predication on branch prediction accuracy. Pnevmatikatos and Sohi [16] proposed *guarded execution*, where a single instruction specified the predication information for subsequent instructions using a bit-mask.

Tyson studied both partial predication, where an architecture supports only a few, limited predicated instructions such as conditional moves, and full predication where all instructions are labeled with a predicate, including support for speculative loads and stores. That study attempted to predicate short forward branches, which correspond to a "one-sided branch hammock" that only has a then-context. Both Tyson and Mahlke found that $\approx$30% of the dynamic branch count could be removed using full predication. Tyson also found that only 5% of the branches could be predicated using partial predication. Half of this difference was due to load and store instructions, and the other half was due to the presence of other branch instructions. Our architectural model addresses a more general control structure (two-sided branch hammocks) and support for speculative loads and stores within the dynamically predicated region. Mahlke *et al.* [14] provide more comparison of fully *vs.* partially predicated execution, and provide a detailed explanation of scheduling partial predicated code. Mahlke makes the point that simple predication schemes, such as conditional moves, increase the processor instruction count. Our model reduces the impact on the I-cache by dynamically inserting the conditional moves.

There is very little work on what we would term "dynamic predication". Sprangle and Patt [20] show that with predication, the renamer can reuse storage if it knows that the storage is predicated under complimentary conditions. Chang *et al.* [3] studied a fully predicated ISA on a speculative architecture with register renaming. They use an additional source input to each instruction to allow copying of the old logical destination register value to the new physical destination register if the predicate is false. This reintroduces output dependencies (write-after-write) between instructions that produce the same logical destination register. Our solution avoids these artificial dependencies by using conditional moves to reconcile data dependencies.

Heil and Smith [9] and Tyson, Lick and Farrens [22] also discuss "dual path" execution. The paper by Tyson does not describe an implementation. The paper by Heil mentions duplicating the processor pipeline resources. By comparison, our method can pipeline multiple branch hammocks, meaning that more than two paths can be under evaluation at any one time. Furthermore, our modifications to the decoder, renamer and scheduler are much simpler than the proposal in [9]. In [24] Uht *et al.* describe Disjoint Eager Execution and minimal control dependencies, which allow execution of control and data independent instructions after the join, concurrently with instructions before the fork branch. Our approach has the same property but comes with a much smaller hardware overhead. Klauser *et al.* [12] describe multipath execution, which uses a path naming scheme that has similarities to our context tagging. However, their work only considers fork operations without rejoining paths.

Rau [17] proposed a mechanism to implement dynamically scheduled VLIW architectures that is similar to our injection of conditional move operations used to reconcile contexts. Our model is similar in that we insert a conditional move to "rename" predicated computation should the predicate be true, but we use dynamic dependence information to control the scheduling of the conditional move rather than using a fixed-delay queue.

Some current architectures implement limited support for predicated execution, e.g. the Alpha ISA [1] supports conditional move instructions and the HP-PA 2.0 ISA [11] supports conditional nullification of one instruction after most computational instructions. In comparison, our approach uses full predication in the microarchitecture without requiring additional instruction set support.

L1 Icache	64 kB, 32 byte lines, 2-way set-associative, 2 cycles hit latency
L1 Dcache	64 kB, 32 byte lines, 2-way set-associative, 2 cycles hit latency
L2 Cache Combined	512 kB, 64 byte lines, direct mapped, 6 cycles hit latency
Memory	128 bit wide, 26 cycles access latency
Branch Predictor	McFarling combined, gshare + bimodal 2k entry each
BTB	1024 entry, 4-way set-associative; 32 entry return address stack
TLB	64 entry (I), 128 entry (D), fully associative
Functional Units and Latency (total/issue)	8 Int.ALU (1/1), 2 Int Mult (3/1) / Div (20/19), 4 Ld./St. (2/1), 8 FP Add (2/1), 2 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)

Table 1. Machine configuration parameters.

				compress	gcc	perl	go	m88ksim	lisp	vortex	jpeg
		Data-Set		train **	amptjp	test ***	2stone9	dhry	train	train *	specmun*
Baseline	Dynamic	Instructions	(M)	80.4	250.9	227.9	548.2	416.5	183.3	180.9	252.0
		Cycles	(M)	38.3	197.7	174.1	507.0	197.5	96.4	53.7	84.7
		Branches	(M)	14.4	50.4	43.7	80.3	89.8	41.8	29.1	20.0
		Misprediction Rate		9.9%	11.9%	11.2%	24.1%	4.3%	7.0%	1.7%	10.4%
		Inst. Per Branch		5.6	5.0	5.2	6.8	4.6	4.4	6.2	12.6
		Inst. Between Mispred.		56.7	41.9	46.7	28.3	106.9	63.1	365.8	121.1
		Inst. Per Cycle		2.1	1.3	1.3	1.1	2.1	1.9	3.4	3.0
Simple Hammock Predication	Static	Cond. Forward Branches		1450	33568	7215	6765	3727	2022	11756	4992
		Hammocks	Single-sided	403	6128	1319	1743	857	420	1423	1248
			Double-sided	77	1640	380	321	218	101	343	202
			Simple	282	3822	968	1176	660	316	1207	734
			Max. Nesting	5	18	8	10	7	5	7	10
			Max. Size (Inst.)	50	243	297	562	237	50	297	302
	Dynamic	Instructions	(M)	87.0	257.6	235.5	575.3	465.2	184.3	186.8	259.7
		Fork Branches	(M)	1.8	2.6	3.4	7.0	21.4	0.7	2.1	1.8
		Inserted Cmoves	(M)	3.9	2.6	3.8	11.1	24.3	0.6	2.2	3.2
		Killed Pred.Inst.	(M)	2.7	4.1	3.8	16.0	24.4	0.4	3.6	4.4
		Committed Pred.Inst.	(M)	9.2	5.5	8.7	20.7	35.8	2.6	6.2	2.7
		Removed Uncond. BR	(M)	0.6	0.2	0.3	2.6	1.3	0.2	0.2	0.1
		Predicated Branches		12.7%	5.2%	7.7%	8.7%	23.8%	1.6%	7.1%	8.8%
		Avg. Hammock Size		6.5	3.7	3.7	5.2	2.8	4.7	4.8	4.0
		Avg. Cmoves Per Hmk.		2.1	1.0	1.1	1.6	1.1	0.9	1.1	1.8
		Avg. Committed / Killed		3.5	1.3	2.3	1.3	1.5	5.9	1.7	0.6

Table 2. Benchmark characteristics and baseline machine performance. The benchmark data sets (*) have reduced input sizes, (**) has increased input size, and data set (***) are the test inputs from the source distribution. Baseline statistics represent the baseline speculative architecture without predicated execution. Simple Hammock Predication statistics are taken from size-based hammock selection with a threshold of 32 instructions. All instruction counts represent millions (M) of retired instructions.

	Size-Based					Profile-Based		Perfect Branches			
						Basic	Enhanced	Simple Hammock		Complex Hammock	
Hammock Size	6	10	16	24	32	32	32	predict	+ fetch	predict	+ fetch
compress	8.72%	10.19%	10.19%	10.20%	10.20%	8.02%	8.84%	10.80%	10.93%	10.76%	11.76%
gcc	2.85%	3.04%	3.11%	3.08%	3.11%	3.07%	3.07%	3.06%	3.26%	6.60%	7.09%
perl	5.62%	5.73%	5.87%	5.82%	5.84%	6.16%	6.16%	5.88%	6.34%	8.70%	9.36%
go	5.95%	6.30%	6.45%	6.50%	6.50%	6.47%	6.45%	6.70%	7.14%	24.78%	25.97%
m88ksim	11.48%	10.92%	12.07%	12.28%	12.40%	12.58%	13.18%	12.91%	13.66%	21.96%	23.12%
xlisp	1.03%	1.05%	1.17%	1.17%	1.17%	-0.01%	-0.01%	0.94%	1.01%	0.86%	1.02%
vortex	0.22%	0.58%	0.50%	0.37%	0.21%	1.66%	1.65%	1.98%	2.03%	3.41%	3.80%
jpeg	3.33%	3.35%	3.37%	3.35%	3.35%	5.20%	5.23%	5.00%	5.83%	10.27%	12.11%
g-mean	4.84%	5.08%	5.27%	5.27%	5.27%	5.33%	5.50%	5.84%	6.20%	10.65%	11.49%

Table 3. Percent speedup for hammock predication with static hammock selection methods (left hand side) and perfect hammock branch execution (right hand side). Perfect prediction and fetch for simple hammock branches represents an upper bound model for hammock predication.

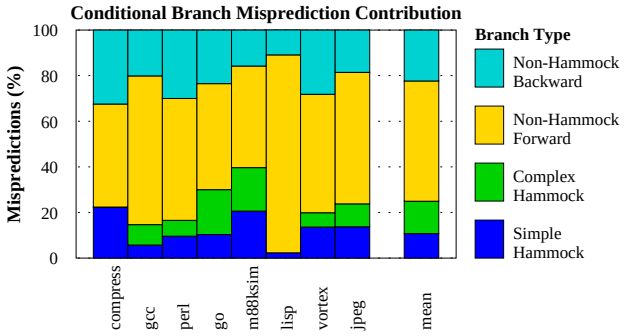

Figure 4. Conditional branch mispredictions.

4 Results

We have used the SimpleScalar tools [2] to build a pipeline-level simulator of our proposed dynamic predication architecture. The baseline machine architecture is a superscalar, out-of-order execution, in-order commit processor with architectural parameters as detailed in Table 1. The architecture can fetch, issue, and commit up to 8 instructions each cycle and has a 128-entry instruction window, a 64-entry load/store queue, and a 16 instruction fetch queue. The architected misprediction penalty is 16 cycles.

For our performance evaluation, we use the SPECint95 benchmark suite with scaled down input data sets to reduce simulation time. All benchmarks run to completion. Table 2 shows the benchmark characteristics and their performance.

4.1 Branch Misprediction Contributions

Tyson and Mahlke [23, 13] found that ≈30% of the dynamic branches could be removed by predication. We have measured the effect on performance if these branches were removed.

In Figure 4, we show the relative contribution of the following subclasses of conditional branches to the misprediction rate of all conditional branches:

- **Simple hammocks** are branches that start a hammock with a single basic block in the then- and else-path, i.e. there are no nested control instructions inside the hammock.
- **Complex hammocks** are branches that start the remaining hammocks, i.e. the hammocks with arbitrary embedded control-flow like function calls, loops, and nested hammocks.
- **Non-hammock forward** branches are all other forward branches that do not start hammocks.
- **Non-hammock backward** are all backward branches.

Our dynamic predication architecture can predicate the subclass of simple hammocks. On average, approximately 11% of all mispredictions of conditional branches fall into this class.

To assess the performance impact of these mispredicted simple hammock branches we simulated the performance of our benchmarks when simple hammock branches are executed perfectly. Perfect execution of hammock branches in this context has two components; avoiding misprediction penalties by perfect prediction and avoiding instruction fetch-block breaks by perfect instruction fetch across the branch. Table 3 shows the results for perfect prediction and perfect prediction with perfect fetch. Perfect fetch can accrue a small additional performance gain over perfect prediction.

Perfect prediction and fetch is useful as an upper bound model for hammock predication. Predicated hammocks can not be mispredicted, since both paths are executed. Also, fetches across hammock forks (former branches) do not break the fetch-block, since these fetches are always consecutive.

The table also shows the speedup for perfectly executed complex hammock branches. Note that for some benchmarks, like *compress*, there is little difference since the fraction of mispredicted complex hammocks is small. Other benchmarks, like *go* have a significantly larger portion of mispredicted complex hammock branches. However, our proposed dynamic predication architecture does not predicate complex hammocks, since it would require a much larger hardware investment than predicating only simple hammocks. As Table 2 shows, complex hammocks in *go* can reach a maximal depth of 10 nested hammocks and a size of 562 instructions, far beyond what could be efficiently predicated.

Note that this perfect branch execution model is sometimes slightly outperformed by our real dynamic predication models. This is an artifact of our perfect model, which does not include the effects of eliminating then-path terminal unconditional branches. Another uncertainty comes from our perfect branch prediction model, which introduces minor timing perturbations in some cache accesses. Also, mispredicted branches can in rare cases slightly increase performance by prefetching instructions which will be used in the near future. A case of this anomaly can be seen e.g. with *compress*, where perfect prediction for simple hammocks slightly outperforms perfect prediction of simple and complex hammocks.

4.2 Static Hammock Selection Methods

In the previous section, we have introduced an upper bound model for hammock predication. In the following subsections, we show the actual performance of our dynamic predication architecture with several hammock selection methods.

4.2.1 Size-based Hammock Selection

The simplest static selection method that can be used is only based on the hammock size, i.e. the number of instructions between the fork and join. Table 3 shows the performance of size-based hammock selection with varying hammock sizes. Each hammock with a size up to the threshold size is marked for predication; larger hammocks are executed with speculation instead. The intention behind this heuristic is that small hammocks are efficient to predicate since the whole hammock is fetched in one or two cycles. It would take the same amount of time to speculate across this hammock, with the potential for mis-speculation.

The results in Table 3 show that the performance increases up to a hammock size of 16 instructions and stays almost constant for larger sizes. Even with this simple selection mechanism, the geometric mean of 5.27% speedup is already close to the 6.2% speedup of perfect branch prediction and fetch for simple hammock branches. Only two benchmarks, *vortex* and *jpeg*, fall significantly short of their perfect execution performance. *Lisp*, on the other hand, slightly outperforms our perfect model due to the reasons outlined above.

Table 2 gives some insight into why this simple hammock selection technique works surprisingly well. The rows labeled ”Simple Hammock Predication” correspond to the performance of size-based hammock selection with a threshold size of 32 instructions. The table shows that, although up to 32 instructions are allowed in a hammock, the average hammock size is only 4 instructions or half a fetch-block. Also, on average less than 2 cmoves are injected to reconcile data dependencies. Due to these cmoves and predicated wrong-path instructions, the number of retired instructions

for dynamic predication increases on the average by about 5%. It is also interesting to note that for all but one benchmark (jpeg), the number of correct-path instructions in hammocks is larger than the number of wrong-path instructions. This program feature helps to limit the resource requirements of wrong-path predicated instructions, which increases the performance of predicated execution.

The overall performance improvement of the different benchmarks, as seen is Table 3, is roughly connected to the percentage of conditional branches that can be predicated (see Table 2). The benchmarks with the largest fraction of predicated branches, *m88ksim* and *compress*, show the highest performance improvements. The benchmarks with the smallest fraction of predicated branches, *lisp*, *gcc*, and *vortex*, show the lowest improvements.

4.2.2 Profile-based Hammock Selection

To enhance the quality of hammock selection, we have also considered profile-based hammock selection. We assume the binary modification tool has access to branch prediction profiles. These profiles contain information about correct predictions and mispredictions for each branch, and can be gathered with tools like ProfileMe [5]. We capture the predication and speculation effects by comparing their wrong-path instruction costs:

$$
\begin{aligned}
\mathit{ThenPathIncorrect} &= \text{number of times the else-path should have been executed} \\
\mathit{ElsePathIncorrect} &= \text{number of times the then-path should have been executed} \\
\mathit{PredicationCost} &= \mathit{ThenPathIncorrect} * \mathit{ThenPathSize} + \mathit{ElsePathIncorrect} * \mathit{ElsePathSize} \\
\mathit{SpeculationCost} &= \mathit{Mispredictions} * \mathit{MispredictPenalty} * \mathit{IssueWidth} \\
\mathit{PredicationCostRatio} &= \mathit{PredicationCost} \,/\, \mathit{SpeculationCost}
\end{aligned}
$$

The parameters *ThenPathIncorrect* and *ElsePathIncorrect* for each branch are extracted from the profile. This cost model estimates the number of instructions from the incorrect path that would enter the pipeline for predication or speculation. It does not capture secondary effects such as cmove injection and data dependence related stalls. With this method, a branch is marked as hammock branch if its predication cost-ratio is below a threshold.

Table 3, column Profile-Based/Basic shows the results for self profiling and the best threshold of 5. The optimal threshold is above 1 since the hidden, not modeled costs of speculation are higher than the hidden costs of predication. Comparing profile- and size-based hammock selection shows that the performance increases for *jpeg*, *vortex*, and *perl*, whereas it decreases for *compress* and *lisp*, resulting in only a small average performance gain for profile-based selection. The reason for the performance losses of two benchmarks can be attributed to the cost metric used for profile-based hammock selection. Our basic cost metric only models costs related to perfect prediction, not the costs related to fetch-block breaks. To alleviate this problem, we have modified the hammock selection algorithm.

The speculation of double-sided hammocks always results in a fetch-block break, either at the hammock branch to jump to the else-path, or at the terminal then-path branch to jump across the else-path. To avoid performance losses due to this fetch-block break, it would be advantageous to predicate very small double-sided hammocks, even if they are predicted with high accuracy. Our enhanced profile-based selection algorithm always predicates double-sided hammocks with then-path sizes up to 4 instructions (half a fetch-block); other hammocks are predicated based on the basic predication cost model as presented above. The results in Table 3 show that the enhanced selection algorithm performs the best hammock selections of all algorithms studies so far. It reaches a speedup of up to 13% for *m88ksim*, and an average of 5.5% over all benchmarks.

	Branch Predictor Update		
	none	history	history & counters
Perfect Branches	6.20%	5.54%	5.02%
Size-Based	5.27%	5.13%	4.55%
Profile-Based / Basic	5.33%	5.44%	4.79%
Profile-Based / Enhanced	5.50%	5.54%	4.95%
Saturating Counters	4.88%	4.91%	1.96%
Miss-Distance Counters	5.10%	4.97%	3.36%

Table 4. Overall percent speedup for different hammock selection methods and branch predictor update strategies for hammock branches.

4.3 Branch Predictor Influence

Hammock branches do not depend on the branch predictor anymore, since instruction fetch always follows the fall-through path and fetches contiguously until the join is reached. This property can be used to reduce the interference for other branches in the branch predictor.

We have looked at three branch predictor update strategies that differ in the amount of interaction between predicated hammock branches and the branch predictor: (1) hammock branches do not update the branch predictor at all, (2) they only update the global history, and (3) they update history and counters. Table 4 shows the results of this comparison for the different hammock selection methods. For perfect branch execution, (1) outperforms the other two approaches and we have shown the data for this case in Table 3.

Approach (2) eliminates destructive interference in the counters due to hammock branches, while still allowing other (non-hammock) branches to benefit from history correlation to hammock branches. Our results reveal that the performance of (1) and (2) only shows minor differences. Some benchmarks like *go* and *vortex* show a slight benefit from history correlation to hammock branches, whereas other benchmarks like *m88ksim* and *lisp* show a negative interference of hammock branch history with other branches. Overall, history update has a negligible effect on performance. In the data presented earlier, we have assumed no history update, since it reduces the complexity of the micro-architecture.

Approach (3) consistently displays the lowest performance improvements, since it does not benefit from reduced branch predictor interference. As seen for perfect hammock branch execution in Table 4, eliminating the interference in the branch predictor contributes approximately 1% to overall speedup.

4.4 Dynamic Hammock Selection Methods

We have also considered dynamic hammock selection methods that use a branch confidence estimator [8, 10] to dynamically determine if a hammock should be predicated. The two designs studied are "Saturating Counters" and "Miss-Distance Counters".

Saturating counters is a simple method that uses the state of the branch predictor counters to derive the confidence estimation. If both indexed counters in the McFarling branch predictor are saturated in the same direction, the branch prediction is of high confidence, otherwise it is low confidence. This scheme is called "Both-Strong" in [8]. It does not require any additional storage for the confidence estimator, since it uses the branch predictor counters.

The second scheme is more expensive and keeps a separate table of miss-distance counters (MDC), which count the number of correct predictions since the last misprediction for this index. The

MDC is incremented for each correct prediction, and reset to zero for each incorrect prediction. If the MDC value is below a threshold, the prediction is low confidence, otherwise it is high confidence. As suggested in [8], we have adapted the indexing mechanism of the MDC table to the McFarling predictor and keep a separate MDC table for each of the two component predictors. The McFarling meta-predictor selects between the two.

The overall performance for dynamic hammock selection can be seen at the bottom of Table 4. For MDC, we show the best performance, which was reached for an MDC-threshold of 7. As expected, the more expensive MDC outperforms the cheaper saturating counter method. However, dynamic hammock selection does not perform as well as static hammock selection. We believe that the reason for this is found in the information that the branch confidence estimator uses to derive its decisions. Confidence estimation only considers the merit of correct branch predictions, but not the advantage of contiguous fetch-blocks. It prevents well predictable hammocks from being predicated. However, it is advantageous to predicate small hammocks even if they are predicted with high accuracy, since predication allows for more efficient instruction fetch. This is supported by the fact that simple size-based hammock selection already performs very well. Therefore, we believe that dynamic confidence estimation is not necessary for dynamic predication.

5 Contributions

In this paper we have concentrated on developing the microarchitectural mechanisms needed to implement dynamic hammock predication. We developed an architectural mechanism called Dynamic Predication, that dynamically predicates and executes both paths after a hammock branch. We describe a context tag architecture to allow multiple paths of execution to coexist on a single threaded architecture, and modifications to the rename table allowing contexts for the before-fork, then, and else paths for proper register renaming in the presence of multiple path execution.

Dynamic predication provides robust speedups of up to 13% over an existing architecture that already has aggressive support for out-of-order speculative execution. On average, it can harness about 90% of the performance improvement that could be achieved through perfect execution of hammock branches.

Moreover, dynamic predication demonstrates consistent performance improvements with only modest hardware costs, replication of renamer hardware, and additional control logic. Dynamic predication exposes predication to the compiler without requiring conditional moves and speculative instruction support in the instruction set architecture. It is an effective measure to hide branch misprediction latency while not creating unnecessary barriers to dynamic scheduler performance.

Acknowledgements

We would like to thank Intel MRL for supporting Artur Klauser during the summer of 1997, Digital Equipment Corporation for an equipment grant that provided the simulation cycles, a grant from Hewlett-Packard, and the anonymous referees for providing helpful comments. This work was partially supported by NSF grants No. CCR-9401689 and No. MIP-9706286.

References

[1] D. P. Bhandarkar. *Alpha Implementation and Architecture*. Digital Press, 1996.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tools Set. Technical Report TR#1308, University of Wisconsin, July 1996.

[3] P.-Y. Chang, E. Hao, Y. Patt, and P. Chang. Using Predicated Execution to Improve the Performance of a Dynamically Scheduled Machine with Speculative Execution. In *Intl. Conf. on Parallel Arch. and Compilation Techniques, Limassol, Cyprus*, June 1995.

[4] D. Draper et al. Circuit Techniques in a 266-MHz MMX-Enabled Processor. *IEEE Journal of Solid-State Circuits*, 32(11), Nov. 1997.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *30th Annual Intl. Symp. on Microarchitecture*, Dec. 1997.

[6] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-System Design Considerations for Dynamically-Scheduled Processors. In *24th Annual Intl. Symp. on Comp. Architecture*, June 1997.

[7] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence Estimation for Speculation Control. In *25th Intl. Symp. on Computer Architecture*, Barcelona, Spain, June 1998.

[9] T. Heil and J. Smith. Selective Dual Path Execution, Nov. 1996. University of Wisconsin-Madison, http://www.ece.wisc.edu/ jes/papers/isca.sdpe.ps.

[10] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *29th Annual Intl. Symp. on Microarchitecture*, pages 142–152, Paris, France, Dec. 1996.

[11] G. Kane. *PA-RISC 2.0 architecture*. Prentice Hall PTR, 1996.

[12] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the PolyPath Architecture. In *25th Intl. Symp. on Computer Architecture*, Barcelona, Spain, June 1998.

[13] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. mei W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *27th Annual Intl. Symp. on Microarchitecture*, San Jose, CA, Dec. 1994.

[14] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *22nd Intl. Symp. on Computer Architecture*, pages 138–149, June 1995.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Intl. Conf. on Microarchitecture*, pages 45–54, Dec. 1992.

[16] D. N. Pnevmatikatos and G. S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *21st Intl. Symp. on Computer Architecture*, pages 120–129, June 1994.

[17] B. R. Rau. Dynamically scheduled VLIW processors. In *26th Annual Intl. Symp. on Microarchitecture*, Austin, Texas, Dec. 1993.

[18] R. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 Departmental Supercomputer. *IEEE Computer*, 22(1):12–35, Jan. 1989.

[19] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *24th Annual Intl. Symp. on Comp. Architecture*, pages 284–291, May 1997.

[20] E. Sprangle and Y. Patt. Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme. In *27th Annual Intl. Symp. on Microarchitecture*, San Jose, CA, Dec. 1994.

[21] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23nd Annual Intl. Symp. on Comp. Architecture*, May 1996.

[22] G. Tyson, K. Lick, and M. Farrens. Limited Dual Path Execution. CSE-TR 346-97, University of Michigan, 1997.

[23] G. S. Tyson. The Effects of Predicated Execution on Branch Prediction. In *27th Annual Intl. Symp. on Microarchitecture*, pages 196–206, San Jose, CA, Dec. 1994.

[24] A. K. Uht, V. Sindagi, and K. Hall. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *28th Intl. Conf. on Microarchitecture*, pages 313–325, Dec. 1995.