Compiler-Assisted Dynamic Predicated Execution of Complex Control-Flow Structures

Hyesoon Kim   José A. Joao   Onur Mutlu   Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2006-005
April 2006

Compiler-Assisted Dynamic Predicated Execution of Complex Control-Flow Structures

Hyesoon Kim José A. Joao Onur Mutlu Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{hyesoon,joao,onur,patt}@ece.utexas.edu

Abstract

Even after decades of research in branch prediction, branch predictors still remain imperfect, which results in significant performance loss in aggressive processors that support large instruction windows and deep pipelines.

This paper proposes a new processor architecture for handling hard-to-predict branches, the diverge-merge processor. *The goal of this paradigm is to eliminate branch mispredictions due to hard-to-predict dynamic branches by dynamically predicating them. To achieve this without incurring large hardware cost and complexity, the compiler identifies branches that are suitable for dynamic predication called* diverge branches. *The compiler also selects a control-flow merge (or reconvergence) point corresponding to each diverge branch to aid dynamic predication. If a diverge branch is hard-to-predict at run-time, the microarchitecture dynamically predicates the instructions between the diverge branch and the corresponding merge point by first executing one path after the branch, then executing the other path, and later merging the data-flow produced by the two paths using special select-uop instructions. The control-flow merge point is selected based on the* frequently-executed paths *in the program using profile information. Therefore, the control-flow from a diverge branch does not* have to *merge (but it usually does), which allows the dynamic predication of a much larger set of branches than simple hammock (if-else) branches .*

Our evaluations show that a diverge-merge processor outperforms a baseline with an aggressive branch predictor by 10.8% on average over 15 SPEC CPU2000 benchmarks, through an average reduction of 31% in pipeline flushes due to branch mispredictions. Furthermore, the proposed mechanism outperforms a previously-proposed dynamic predication mechanism that can predicate only simple hammock branches by 7.8%.

1. Introduction

State-of-the-art high performance processors employ deep pipelines to extract instruction level parallelism (ILP) and to support high clock frequencies. In the near future, processors are expected to support a large number of in-flight instructions [25, 32, 7, 4, 10] to extract both ILP and memory-level parallelism (MLP). The performance improvement provided by both pipelining and large instruction windows critically depends on the accuracy of the processor's branch predictor [31, 25, 32]. Branch predictors still remain imperfect, even though intensive research in branch prediction has been carried out for decades. Hard-to-predict branches not only limit processor performance but also result in wasted power consumption.

When a branch misprediction is detected, a current processor flushes all of the instructions following the branch in the pipeline and begins fetching from the correct program path. If the correct and the wrong paths merge, instructions on the control-independent section of both paths will be fetched, decoded and renamed again after the processor recovers from the branch misprediction. Figure 1 shows the percentage of control-independent and control-dependent wrong-path instructions out of all instructions fetched by the processor for the twelve SPEC CPU2000 integer benchmarks and three SPEC CPU2000 floating-point benchmarks.[1] On average, 52% of all instructions fetched by the evaluated processor are on the wrong path,

[1]This data is measured for an 8-wide, 30-stage-pipeline, 512-entry-window processor with a 300-cycle memory latency. The detailed machine configuration is provided in Section 3.

even with a 3.3% conditional branch misprediction rate.[2] Furthermore, about 33% of all instructions (i.e., 63% of the wrong-path instructions) are on the control-flow independent portion of the wrong path. Based on these results, we can conclude that near-future processors will spend significant time fetching wrong-path instructions and a significant number of wrong-path instructions are actually on the control-flow independent path.

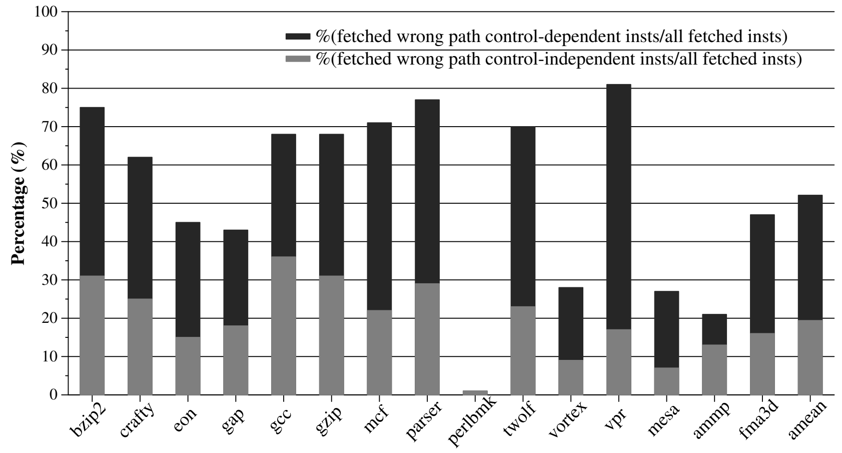


Figure 1. Percentage of control-independent and control-dependent wrong-path instructions

In order to save the useful work done by control-flow independent instructions on the wrong path, several techniques have been proposed to selectively flush the pipeline [28, 29, 8, 5, 13]. Such mechanisms try to flush only those instructions that are control-dependent on the mispredicted branch, thereby reducing the branch misprediction penalty. Unfortunately, the mechanisms proposed to exploit control-flow independence significantly increase the complexity of the hardware [28]. For example, they require non-trivial hardware support to insert the control-dependent correct-path instructions in the middle of the instruction stream during misprediction recovery in order to avoid the flush of the control-independent instructions. Furthermore, some instructions in the control-independent section of the wrong path can be data-dependent on results produced by instructions that are control-dependent on the mispredicted branch. For correct program functionality, such data dependent but control-independent instructions need to be supplied with correct source data values. Selective pipeline flush therefore requires complex hardware mechanisms to detect such instructions, fix their data dependencies, and re-execute them after recovery from misprediction.

Predication is another approach to avoid the pipeline flush due to branch mispredictions. Predication converts control-dependencies into data-dependencies [1]. With predication, the processor fetches instructions from both paths of a branch but only commits the results from the correct path, effectively avoiding the pipeline flush associated with a branch misprediction. However, predication requires significant support (i.e., predicate registers and predicated instructions) in the instruction set architecture (ISA). Furthermore, executing the compile-time predicated code on easy-to-predict branches can often times hurt performance. This is because the additional overhead of executing useless instructions could offset the benefit of avoiding

[2]perlbmk shows only a 0.3% misprediction rate in the reduced input set, which makes it an outlier in Figure 3. However, it shows a similar percentage of wrong-path instructions to the other benchmarks with the reference input set. We perform all our experiments with the reduced input set to reduce simulation time, since all other benchmarks show similar branch misprediction rates for the reduced and reference input sets.

the relatively infrequent pipeline flushes [6, 19].

Dynamic Hammock Predication (DHP) [20] was proposed to predicate hard-to-predict branches at run-time without ISA support. This approach is attractive not only because it does not require modifications to the ISA, but also because it applies predication selectively to each dynamic instance of a branch, thereby avoiding the additional overhead of predication when the branch predictor is likely to be correct. However, DHP can only predicate simple hammock branches (simple if-else structures with no other control flow inside), which account for a small subset of the mispredicted branches [20]. Our goal in this paper is to devise a technique that reduces the branch misprediction penalty by applying dynamic predication to more *complex control-flow structures* without requiring predication support in the ISA and at the same time to avoid the overhead associated with compile-time predication for easy-to-predict dynamic branches.

We propose a new microarchitecture, called the *Diverge-Merge Processor*. The diverge-merge processor uses some control-flow information provided by compiler hints. The compiler identifies and marks suitable branches as candidates for dynamic predication. These branches are called *diverge branches*. The compiler also selects a control-flow merge (or reconvergence) point corresponding to each diverge branch. If a diverge branch is hard-to-predict at run-time (as determined by a confidence estimator), the processor dynamically predicates the instructions between the diverge branch and the corresponding control-flow merge point identified by the compiler. Dynamic predication of the diverge branch is accomplished by first executing one path after the branch, then executing the other path, and later merging the data-flow produced by the two paths using special select-$u$op instructions. The control-flow merge point is selected based on the frequently-executed paths in the program using profile information. Therefore, the control-flow from a diverge branch does not *have to* merge (but it *usually* does), which allows the dynamic predication of a much larger set of branches than simple hammock branches. Hence, a diverge-merge processor avoids flushing the pipeline for a large set of mispredicted branches through cooperation between the compiler and the microarchitecture without requiring full support for predication in the ISA.

The major contributions and benefits provided by the diverge-merge processor architecture are:

1. It reduces the branch misprediction penalty by keeping the control-independent instructions in the instruction window without requiring complex hardware support.
2. It uses predication only for the hard-to-predict dynamic instances of statically-selected branches, thereby avoiding the overhead of predication when a dynamic branch is easy-to-predict.
3. It does not require full predication support in the ISA, thereby making the benefits of predicated execution applicable to existing ISA's with minimal changes.
4. Most importantly, **it can dynamically predicate complex, as well as simple, control-flow graphs**. Therefore, it makes the benefits of dynamic predication applicable to a much larger set of mispredicted branches than the previously-proposed dynamic predication approaches (e.g. [20]).

This paper first explains the diverge-merge processor architecture in Section 2. Then, we describe the simulation methodology and the selection heuristics of diverge branches and control-flow merge points in Section 3. Section 4 evaluates the performance of the diverge-merge processor on SPEC2000 benchmarks. Section 5 compares our work to related research in the areas of predication, multipath execution, and control-flow independence.

2. The Diverge-Merge Processor

2.1. The Basic Idea

A diverge-merge processor handles hard-to-predict branches through cooperation between the compiler and the microarchitecture. The goal of the processor is to eliminate branch misprediction-related pipeline flushes due to suitable hard-to-predict dynamic branches by dynamically predicating them without incurring very large hardware cost and complexity.

The compiler identifies conditional branches with control flow suitable for dynamic predication as *diverge branches*. Diverge branches are branches after which the execution of the program *usually* reconverges at a control-independent point in the control-flow graph, a point we call the *control-flow merge (CFM) point*. In other words, diverge branches form hammock-shaped control flow based on *frequently-executed paths in the control-flow graph* of the program but they are not necessarily simple hammock branches which *require* the control-flow graph to be hammock-shaped. The compiler also identifies a CFM point associated with the diverge branch. The diverge branches and CFM points are conveyed to the microarchitecture through modifications in the ISA.

When the microarchitecture fetches a diverge branch, it decides whether or not the branch is hard-to-predict. If the diverge branch is hard-to-predict, the microarchitecture dynamically predicates the branch. This is accomplished by first executing one path after the branch until the CFM point of the branch is reached, then executing the other path until the CFM point of the branch is reached. After the execution of both sides of the "hammock," the processor "merges" the data (i.e., register values) produced by both sides of the hammock by inserting select-uops to ensure that instructions dependent on the values produced in either side of the hammock are supplied with the correct data values which depend on the correct direction of the diverge branch.[3] If the hard-to-predict diverge branch is actually mispredicted and the processor is already at a control-independent point, the processor does not need to flush its pipeline since the branch is effectively *eliminated* through dynamic predication.

This section describes the detailed implementation of the diverge-merge processor and explains the tradeoffs and design choices that need to be addressed in designing a diverge-merge processor.

2.2. Overview

Figure 2 shows the overview of the diverge-merge processor. The shaded structures constitute the additional hardware required to support the diverge-merge processor and will be explained in this section. The uop logic inserts some uops (*enter.pred.path, enter.alternate.path* and *exit.pred*) at the front-end to support dynamic predication in later pipeline stages. The select-uop logic inserts uops after dynamically-predicated code to ensure that the control-independent instructions will have correct data dependencies. A confidence estimator is used to decide whether or not to enable predication for each

[3]In other words, the processor first "diverges" its execution to execute both control-flow paths that stem from a hard-to-predict diverge branch. The processor "merges" the values produced by both paths when the CFM point of the diverge branch is reached. Hence the name diverge-merge processor.

dynamic instance of the candidate branches. The logic and the storage for checkpointing the register alias table also have to be added if they are not already present in the baseline processor.[4]

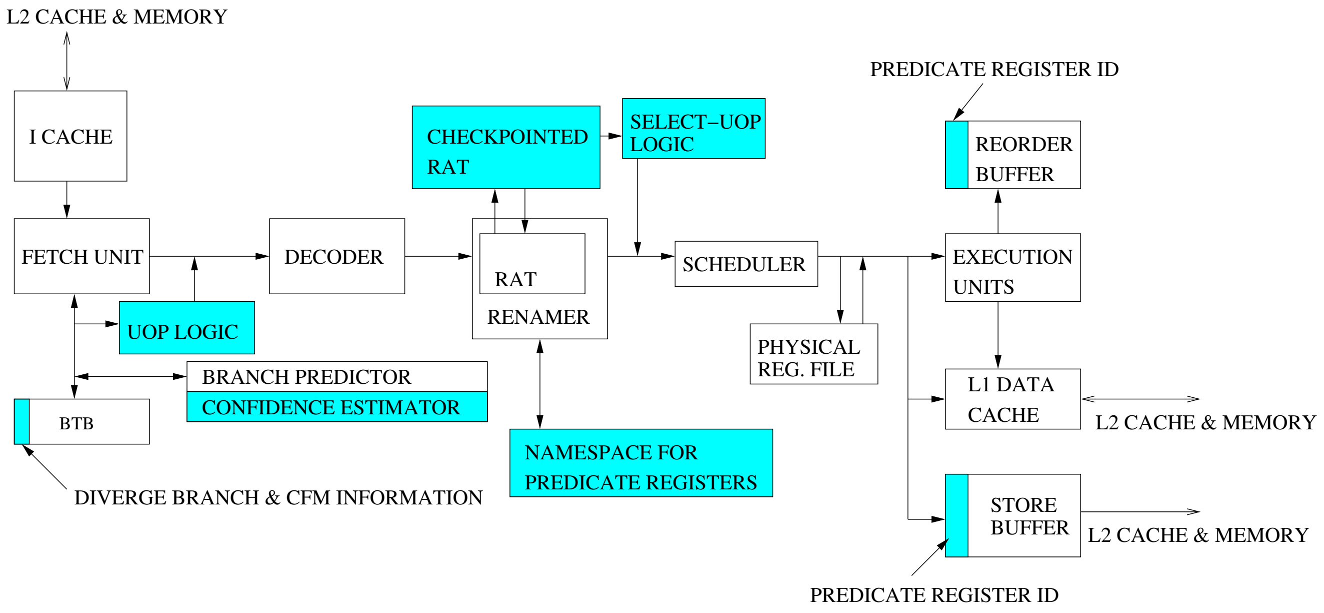


Figure 2. Simplified diagram of a diverge-merge processor showing the changes required in the baseline microarchitecture. The diagram focuses on the modified portions of the pipeline and it is not to scale.

2.3. The Fetch Mechanism

Figure 3 shows an example control-flow graph. The branch that ends block A is a diverge branch, and the first instruction at block H is the corresponding control-flow merge (CFM) point. The solid lines are frequently executed paths, and the dashed lines are the remaining control-flow paths. If there were no dashed lines, all the control-flow paths from A would reach the CFM point, and the CFM point would also be the immediate post-dominator of block A. The compiler finds a diverge branch and the corresponding CFM point considering only the frequently executed paths. Therefore, for many control-flow graphs, the selected CFM point is much closer to A than the immediate post-dominator. Frequently executed path information can be collected by profiling or compiler heuristics. The compiler can mark a diverge branch and the corresponding CFM point with a special instruction encoding at compile time.

When the diverge-merge processor fetches a low-confidence diverge branch, the processor enters *dynamic predication mode*. In this mode, the processor first stores the corresponding CFM point in a buffer (CFM register), and the front-end of the processor inserts an *enter.pred.path* $u$op into the pipeline. Then the processor continues to fetch instructions based on the outcomes of the branch predictor until it reaches the corresponding CFM point.[5] This path is called the *predicted path*. After that, the processor goes back to the diverge branch that started dynamic predication mode and begins fetching from the other path of the branch. This path is called the *alternate path*. When the processor starts fetching from the alternate path, the

[4]The uop logic at the front-end and the checkpoint mechanism of Register Alias Table (RAT) are already employed in most current processors.
[5]When the next predicted fetch address is the CFM point of the diverge branch, the processor reaches the CFM point.

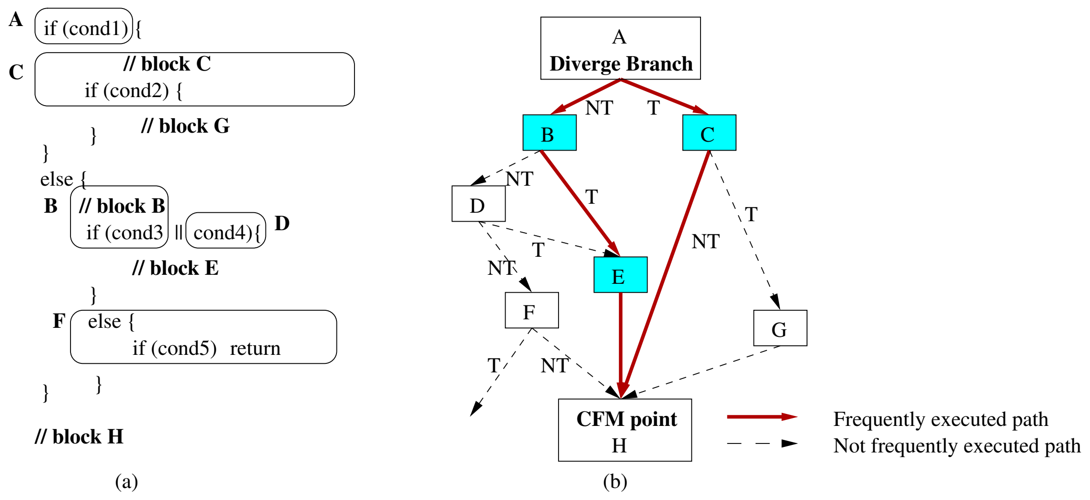


Figure 3. Control-flow graph example for the dynamic predication mechanism: (a) source code (b) control-flow graph

front-end inserts an *enter.alternate.path* $u$op into the pipeline. The processor also fetches instructions on the alternate path based on the outcomes of the branch predictor. When the processor reaches the CFM point again, the front-end inserts an *exit.pred* $u$op into the pipeline and exits dynamic predication mode. All the instructions fetched during dynamic predication mode are dynamically predicated. Dynamic predication is explained in Section 2.4.

For example, assume that, in Figure 3, the diverge branch in block A has low confidence, and the predicted direction is not-taken. The processor enters dynamic predication mode and fetches block B. The next branch is predicted taken, so the processor fetches block E. At the end of block E, the next fetch address is the address of the first instruction in block H (the CFM point), which means that the processor completed fetching instructions from the predicted path. Now, the processor goes back to the diverge branch, and it starts fetching instructions from the other path of the diverge branch, which is block C. The branch at block C is predicted not-taken, and the fall-through address is the address of the first instruction in block H, so the processor reaches the CFM point again. At this point the processor exits dynamic predication mode.

We checkpoint the global history register (GHR) of the branch predictor before entering dynamic predication mode (GHR1) in order to use it for both the predicted and the alternate path[6], and we keep the final GHR of the alternate path after exiting dynamic predication mode.[7] Since the pattern history table of the branch predictor is updated when a branch is retired, it is not polluted by the outcome of wrong-path branches (i.e., branches whose predicate values are FALSE).

2.4. Checkpointing and Select-$u$op Mechanisms to Support Dynamic Predication

The diverge-merge processor implements dynamic predication with checkpoints and a select-$u$op mechanism. Three $u$ops inserted at the front-end of the processor are used to support dynamic predication. Figure 4 and Figure 5 illustrate the dynamic predication process. Figure 4 shows the instruction stream for the example in Section 2.3.

The processor creates a checkpoint (CP1) of the register alias table when the enter.pred.path $u$op enters the rename stage. REGMAP1 in Figure 5 shows the contents of CP1. This uop also defines a predicate register (p1) according to the condition

[6]The last bit of the GHR1, which corresponds to the diverge branch, is set for the taken path and reset for the not-taken path.
[7]This design choice is based on simulation results.

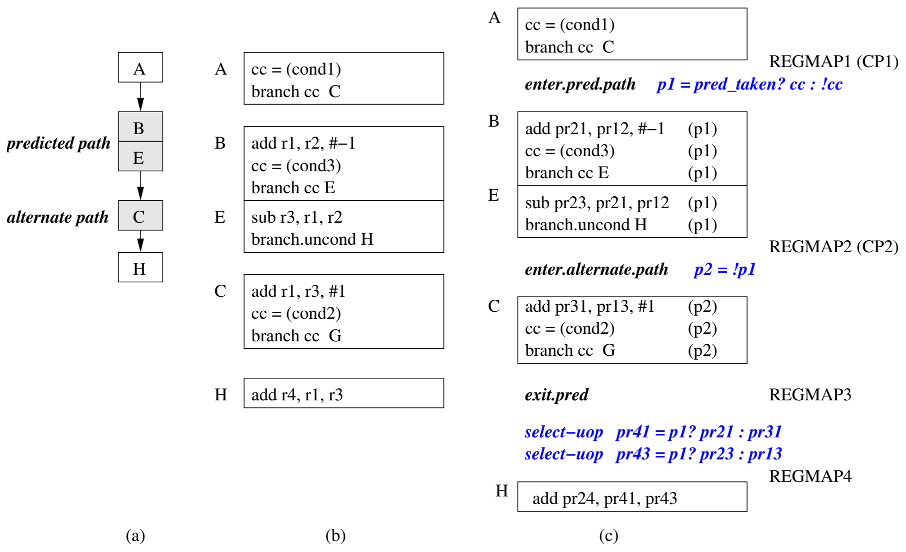

A
B
E
C
H
predicted path
alternate path
A cc = (cond1) branch cc C
B add r1, r2, #−1 cc = (cond3) branch cc E
E sub r3, r1, r2 branch.uncond H
C add r1, r3, #1 cc = (cond2) branch cc G
H add r4, r1, r3
A cc = (cond1) branch cc C
REGMAP1 (CP1)
enter.pred.path p1 = pred_taken? cc : !cc
B add pr21, pr12, #−1 (p1) cc = (cond3) (p1) branch cc E (p1)
E sub pr23, pr21, pr12 (p1) branch.uncond H (p1)
REGMAP2 (CP2)
enter.alternate.path p2 = !p1
C add pr31, pr13, #1 (p2) cc = (cond2) (p2) branch cc G (p2)
exit.pred
REGMAP3
select−uop pr41 = p1? pr21 : pr31
select−uop pr43 = p1? pr23 : pr13
REGMAP4
H add pr24, pr41, pr43
(a)
(b)
(c)

Figure 4. Instruction stream in the example of Figure 3: (a) fetched blocks (b) fetched assembly instructions (c) instructions after register renaming

Architectural	Physical	M
R1	PR11	0
R2	PR12	0
R3	PR13	0
R4	PR14	0

REGMAP1 (CP1)

Architectural	Physical	M
R1	PR21	1
R2	PR12	0
R3	PR23	1
R4	PR14	0

REGMAP2 (CP2)

Architectural	Physical	M
R1	PR31	1
R2	PR12	0
R3	PR13	0
R4	PR14	0

REGMAP3

Architectural	Physical	M
R1	PR41	0
R2	PR12	0
R3	PR43	0
R4	PR14	0

REGMAP4

Figure 5. Register alias table (REGMAP1) before renaming block B, (REGMAP2) after renaming block E, (REGMAP3) after renaming block C, (REGMAP4) after inserting select-$u$ops

of the branch and the branch prediction direction. If the branch is predicted to be taken, the predicate register value is the same as the branch condition. If the branch is predicted to be not-taken, the predicate register value is the complement of the branch condition. All instructions on the predicted path carry the predicate register identification (id) number in order to correctly update the architectural state at retirement.

The processor creates another checkpoint (CP2) when the enter.alternate.path $u$op enters the rename stage. REGMAP2 in Figure 5 shows the contents of CP2. Then, the processor restores the register alias table in CP1 into the processor's active register alias table that is used for renaming the incoming instructions.[8] It also assigns a new predicate register (p2), which is the complement of the predicate register value assigned to the predicted path (p1). All instructions in the alternate path carry the second predicate register id number after the rename stage. Note that an allocated predicate register id gets deallocated when the corresponding predicate value is computed and broadcasted.

Restoring CP1 into the active register alias table at the beginning of the alternate path ensures that the instructions in the

[8]The processor can optimize this process to use only one checkpoint. However, many current high performance processors already checkpoint the register alias table at every branch instruction to support fast branch misprediction recovery (e.g., [18, 34]). Our proposed mechanism does not add any special hardware on top of what exists in current processors to support saving/restoring of register alias table checkpoints.

alternate path (block C) have correct data dependencies. For example, the add instruction in block C should source the R3 that is produced before the diverge branch (i.e. PR13) and not PR23, which is produced in block E.

We extend the register alias table with one extra bit (M -modified-) per entry to indicate that the corresponding architectural register has been renamed in dynamic predication mode. Before entering dynamic predication mode, all M bits are cleared. Then, every time a register is renamed, i.e. the corresponding register alias table entry is modified, its M bit is set.

The select-$u$op mechanism inserts select-$u$ops when the exit.pred $u$op enters the rename stage. Select-$u$ops are similar to the Phi-nodes in the static single-assignment (SSA) form [11]. A select $u$op essentially selects the correct physical register for a given architectural register based on the resolved predicate value (i.e., condition of the branch). The select-$u$op insertion mechanism checks the register alias table in CP2 and the active (current) register alias table. If the physical registers assigned to a particular architectural register are different in the two register alias tables, a select-$u$op is inserted to choose between those two physical registers, according to the predicate register value. We determine the next register that requires a select-$u$op by simply or-ing the M bits for corresponding entries in the two register alias tables and then using a priority encoder to produce the register number to index into the register alias tables. If the logical-or of the M bits of an architectural register in the two register alias tables evaluates to 1, then a select-$u$op is required for that register and we reset the M bits on both entries after creating the select-$u$op. This mechanism can be extended to produce as many select-$u$ops per cycle as the number of ports in the register alias table allows. In this example, physical registers for R1 and R3 are different between REGMAP2 and REGMAP3. R1 is written in both paths, and R3 is only written in block E. Therefore, two select-$u$ops are inserted to choose between the two different physical registers assigned to each architectural register R1 and R3. New physical registers are allocated for the destinations of the select-$u$ops. REGMAP4 in Figure 5 shows the register alias table after the insertion and renaming of the select-$u$ops. The instructions in block H source the destination registers of the select-$u$ops. The processor releases all checkpointing resources after inserting the select-$u$ops.

2.5. Instruction Execution and Retirement

Dynamically predicated instructions are executed just like other instructions. Since these instructions depend on the predicate value only for retirement purposes, they can be executed before the predicate value (i.e., the diverge branch) is resolved. When the condition of the diverge branch that started dynamic predication mode is resolved, the predicate register values of the enter.pred.path and enter.alternate.path $u$ops are produced and broadcasted to the store buffer and the retirement logic (reorder buffer). The instructions on the incorrect path (either the predicted path or the alternate path depending on the correct branch direction) are still executed even though their predicate values are false, but they do not update the architectural state at the retirement stage. In order to support precise exceptions, results produced on the incorrect path should not be committed. When an instruction on the predicated-FALSE path is ready to be retired (i.e., when the predicate value for that instruction is known to be FALSE) the processor simply frees the physical register allocated for that instruction.[9]

[9]Note that this does not significantly increase the complexity in the retirement pipeline. In a current out-of-order processor, when an instruction is ready to be retired, the processor frees the physical register allocated by the previous instruction that wrote to the same architectural register. This is exactly how physical registers are freed in a diverge-merge processor for non-predicated and predicated-TRUE instructions. The only difference is that a predicated-FALSE instruction frees the physical register allocated by itself (since that physical register will not be part of the architectural state) rather than the physical

Dynamically predicated load instructions are also executed like normal load instructions. Dynamically predicated store instructions are sent to the store buffer with their predicate register id. However, a predicated store instruction is not sent further down the memory system (i.e., into the caches) until its predicate register value is ready. If the predicate value of a store instruction is resolved and it turns out to be false, the processor drops the store request and does not update the architectural memory state with that request. The dynamic predication mechanism requires the store buffer logic to check the predicate register value in addition to the branch misprediction signal before sending a store request to the memory system.

The dynamic predication mechanism could complicate the store-load forwarding logic. If there are two store instructions to the same address from both the predicted and the alternate paths, then a subsequent load after the CFM point does not know which store value needs to be forwarded. The processor could insert a select-$u$op to choose between these two store instructions. However, as a design choice to simplify the hardware in this case, the processor instead just waits and delays the execution of the load instruction until the predicate register values of the older store instructions are broadcasted. Since all instructions on each path have an associated predicate register id, the forwarding logic can determine that two store instructions are on different paths if they have different predicate register ids. We can only forward from: (1) a non-predicated store to any later load, (2) from a predicated store whose predicate register value is ready to any later load, or (3) from a predicated store whose predicate register is still not ready to a later load with the same predicate register id (i.e. on the same dynamically predicated path). If forwarding is not possible, the load waits.[10]

2.6. Resolution of the Diverge Branch

So far, we have described the diverge-merge processor for the cases when the processor reaches the CFM point on both the predicted path and the alternate path. However, since the CFM point is determined based on frequently executed paths instead of using the immediate post-dominator, there are cases when the processor does not reach the CFM point before the diverge branch is resolved. Table 1 summarizes the exit cases.

Table 1. Six exit cases of dynamic predication mode

case	predicted path	alternate path	branch prediction	processor action	compared to branch prediction
1	reach CFM	reach CFM	correct	normal exit	overhead of alternate path (-)
2	reach CFM	reach CFM	mispredicted	normal exit	reduce the branch misprediction penalty (++)
3	reach CFM	no reach	correct	re-direct fetch	overhead of alternate path (- -)
4	reach CFM	no reach	mispredicted	no special action	reduce the branch misprediction penalty (+)
5	no reach	—	correct	no special action	same
6	no reach	—	mispredicted	flush the pipeline	same

The processor exits dynamic predication mode normally in case 1 and case 2. Both cases are common cases because the control-flow paths that lead to CFM points are frequently executed paths. In case 2, dynamic predication eliminates a pipeline flush due to a branch misprediction. However, in case 1, the diverge-merge processor incurs the overhead of

register allocated by the previous instruction that wrote to the same architectural register.

[10]Note that it is possible to speculatively forward the data value from a predicated store whose predicate value is not ready to a later load in case (2), but if the forwarded value is incorrect, this option requires the re-execution of the load and its dependents, which complicates the hardware.

executing instructions on the alternate path without gaining any performance benefits compared to the branch prediction mechanism. Note that the occurrence of case 1 can be reduced by designing better confidence estimation mechanisms.

In cases 3, 4, 5, and 6, the processor resolves the diverge branch before exiting dynamic predication mode normally. Therefore, when the branch is resolved dynamic predication mode ends, and the resources associated with dynamic predication mode (e.g. checkpoints and the CFM register) are released.

In case 3, the processor has already finished fetching the correctly predicted path, and it is fetching instructions from the wrong path (which is the alternate path in this case) when the diverge branch is resolved. This is the worst-case scenario for the dynamic predication mechanism because the processor spent a significant amount of time fetching wrong-path instructions, whereas the processor could have fetched only the correct-path instructions with the normal branch prediction mechanism. When the branch is resolved, the processor does not flush the pipeline because the pipeline still has instructions from the correctly predicted path, and all the instructions in the alternate path will become NOPs (i.e., their predicates will be FALSE). The processor simply restores the register alias table from the checkpoint that was taken at the end of the predicted path during dynamic predication mode and starts fetching from the CFM point.

In case 4, the processor is still fetching and executing instructions from the correct path (which is the alternate path in this case) when the branch is resolved. In this case, the processor does not require any special action other than deallocating the dynamic predication resources. Case 4 also reduces the branch misprediction penalty.

The performance of cases 5 and 6 is the same as the baseline branch prediction mechanism. The processor is still fetching instructions from the predicted path, and it has not started fetching from the alternate path. In both cases, the processor exits dynamic predication mode and releases the associated resources.

If branches other than the diverge branch are mispredicted on the predicted path or the alternate path in dynamic predication mode, the processor flushes the instructions after the mispredicted branch just like in the case of a normal branch misprediction. For example, if either branch in block B or C in Figure 3 is mispredicted, then the processor flushes the instructions younger than the mispredicted branch.[11]

2.7. Enhanced Diverge-Merge Mechanisms

2.7.1. Multiple CFM Points The basic diverge-merge processor we have described so far supports only one CFM point. Some branches have several frequently executed paths that could lead to more than one CFM point. In the enhanced diverge-merge processor, the compiler can insert more than one CFM point for each diverge branch. The hardware stores the possible CFM points into a multiple-entry content-addressable memory (CAM), and the processor compares the next fetch address to the multiple possible CFM points, instead of comparing to only one CFM point, to determine whether or not it has reached a CFM point. The processor uses the first seen CFM point to end the predicted path. Then, this CFM point becomes the only CFM point that can end the alternate path. The rest of the dynamic predication mechanisms described for the basic

[11]The register alias table is checkpointed before each branch to simplify the branch misprediction recovery, and we include the state of dynamic predication mode in the checkpoint (i.e., the CFM register and two bits to indicate if the processor is on the predicted path, on the alternate path, or already done with dynamic predication mode). Therefore, if the diverge branch is still unresolved when a branch is mispredicted on either the predicted or the alternate path, we can restart fetching in dynamic predication mode from the correct path of the mispredicted branch, after flushing the pipeline and restoring the checkpoint.

diverge-merge processor do not require modifications to support multiple CFM points.

2.7.2. Early Exit From Dynamic Predication Mode To reduce the negative impact on performance due to case 3 described in Section 2.6, the processor can use simple heuristics to exit dynamic predication mode before reaching the CFM point. These simple heuristics are used only when the processor is on the alternate path. The purpose of these heuristics is to predict the cases when the processor will actually *not* reach the CFM point on the alternate path before the resolution of the diverge branch. If the processor can accurately predict that the alternate path will never merge (reach the CFM point), it can exit from dynamic predication mode early. After exiting the dynamic predication mode early, the processor restores the state that was produced by the predicted path when it reached the CFM point, and it restarts execution from the CFM point.[12] Because only the predicted path is completely executed, the processor does not need to insert select-$u$ops when it restarts execution from the CFM point. In essence, this prediction mechanism reverts the dynamic predication mode to the baseline branch prediction mechanism, which simply follows the predicted path for the diverge branch.

We use a simple heuristic to predict that the processor will not reach the CFM point on the alternate path. If the processor does not reach the CFM point before a certain number of instructions (N) are fetched on the alternate path, the processor exits dynamic predication mode. N can be a static threshold set at the design time of the processor or it can be chosen by the compiler for each diverge branch. We found that a compiler-selected threshold for each diverge branch performs slightly better than a static threshold that is the same for every diverge branch. This is because the number of instructions executed to reach the CFM point varies across different diverge branches. Better heuristics that take into account more information (such as the predicted directions of other branches on the alternate path or the average number of dynamic instructions executed to reach the CFM point) can be developed, and these can further improve the performance of the basic diverge-merge processor. Investigating such heuristics is a part of our future work.

2.7.3. Multiple Diverge Branches During dynamic predication mode, the basic diverge-merge processor ignores other low-confidence diverge branches. If the enhanced diverge-merge processor finds another low-confidence diverge branch while it is fetching *the predicted path* of a diverge branch, it ends dynamic predication mode for the current diverge branch, which becomes a normal predicted branch, and re-enters dynamic predication mode for the newly-found diverge branch. If the processor is already fetching from the alternate path, then it ignores other low-confidence diverge branches. We found that the CFM point of the diverge branch that caused entry into dynamic predication mode is less likely to be reached if another low-confidence diverge branch is encountered on the predicted path. Exiting dynamic predication due to the first diverge branch and re-entering dynamic predication due to the second branch is therefore more likely to eliminate a misprediction and improve performance.

[12] In addition, the predicate value produced by the enter.pred.path uop (p1) is broadcasted as TRUE, assuming that the predicted path is the correct path, so that instructions already predicated on the predicted path disregard their predicate values after the processor exits dynamic predication mode. Note that it is not necessary for correctness to broadcast the predicate value produced by the enter.pred.path uop right after exiting dynamic predication mode. However, if the predicate value is not broadcast early after exiting dynamic predication mode, there may be some performance loss compared to the baseline branch prediction mechanism because loads after the CFM point cannot forward from stores to the same address whose predicate values are not ready.

2.7.4. Future Work on Enhanced Mechanisms The performance benefits of the enhanced diverge-merge processor can be further improved via more enhanced mechanisms that provide better software and hardware support. We are planning to implement and evaluate the following mechanisms:

Diverge Loop Branches. The diverge-merge processor can distinguish between forward branches and backward branches (loop branches) in order to implement the dynamic predication of low-confidence loop iterations. Dynamically predicating backward branches will make the benefits of the diverge-merge processor applicable to hard-to-predict loop branches, similarly to the recently proposed wish loop instructions, a corresponding software construct that makes compile-time predication applicable to loop branches [19]. We are currently investigating compiler algorithms to identify good diverge loop branches and hardware mechanisms to estimate the confidence of loop branches.

Dynamic Predication of Multiple Diverge Branches. Instead of exiting dynamic predication mode for the current diverge branch when another low-confidence diverge branch is fetched on the predicted path, as in the enhanced mechanism in Section 2.7.3, the enhanced diverge-merge processor could continue in the dynamic predication mode by also predicating the newly-found diverge branch. The predicate register values for the second diverge branch should be and-ed with the predicate register corresponding to the previous branch (p1). Different policies can be researched to end dynamic predication mode in this case, as well as the maximum number of nested diverge branches that should be allowed. Another option for handling multiple diverge branches could be to re-evaluate the original branch prediction with additional information when another low-confidence diverge branch is found, in order to choose whether to exit or to continue dynamic predication mode for the current branch.

Selective Branch Predictor Update Policy. As noted by Klauser et al. [20] for Dynamic Hammock Predication, not updating the branch predictor counters for the low-confidence hammock branches eliminates destructive interference in the counters and improves performance. However, our current implementation of the diverge-merge processor updates the branch predictor counters for all the branches that retire. The performance of the diverge-merge processor can be further improved by optimizing the branch predictor update policy.

3. Methodology

3.1. Simulation Methodology

We use an execution-driven simulator of a processor that implements the Alpha ISA. Wrong-path effects are modeled faithfully. A large and aggressive branch predictor is used to avoid inflating the performance of the diverge-merge concept. The parameters of the processor we model are shown in Table 2.

The experiments are run using the 12 SPEC CPU2000 integer benchmarks and 3 SPEC CPU2000 floating point benchmarks. We selected the 3 floating point benchmarks, mesa, ammp and fma3d, as they are the only ones that show at least a 3% performance improvement with a perfect branch predictor over the baseline branch predictor. The rest of the SPEC CPU2000 floating point benchmarks have less than 3% potential improvement with perfect branch prediction. Table 3 shows the characteristics of the benchmarks on the baseline processor. All binaries are compiled for the Alpha ISA with the -fast

Table 2. Baseline processor configuration.

Front End	Fetches up to 3 conditional branches but fetch ends at the first taken branch I-cache: 64KB I-cache, 2-way, 2-cycle latency
Branch Predictors	64KB (59-bit history, 1021-entry) perceptron branch predictor [17] 4K-entry BTB; 64-entry return address stack; 64K-entry indirect target cache minimum branch misprediction penalty is 30 cycles
Execution Core	512-entry reorder buffer; 8-wide execute/retire
On-chip Caches	L1 data cache: 64KB, 4-way, and 2-cycle latency L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle latency All caches use LRU replacement and have 64B line size
Buses and Memory	300-cycle minimum memory latency; 32 memory banks 32B-wide core-to-memory bus at 4:1 frequency ratio
Confidence Estimator	1KB (12-bit history) JRS estimator [16]

optimizations. A binary instrumentation tool is used to mark the diverge branches and their respective CFM points. The integer benchmarks are run to completion with a reduced input set [22] to reduce the simulation time. Since it is important to fully exercise the control-flow graphs throughout the programs, we use the reduced input set to simulate the integer benchmarks until completion in a reasonable time. Floating point benchmarks are simulated for 250 million instructions with the reference input set after skipping the program initialization code using a SimPoint-like tool [30]. We use the train input data set to collect profile information from the binaries. In all the IPC (retired Instructions Per Cycle) performance results shown in the rest of the paper for the diverge-merge processor, the instructions whose predicate values are FALSE and additional uops inserted to support dynamic predication do not contribute to the instruction count.

Table 3. The characteristics of the baseline processor: base IPC, total number of retired instructions (Insts), total number of retired dynamic branches (Br.), total number of retired mispredicted dynamic branches (Misp.)

	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	mesa	ammp	fma3d
Base IPC	1.51	2.58	3.30	2.95	1.38	2.03	0.81	1.58	3.24	2.22	3.44	1.52	4.14	1.72	2.71
Insts	316M	190M	129M	404M	83M	248M	110M	255M	187M	101M	284M	75M	250M	250M	250M
Br.	32M	23M	13M	60M	16M	23M	21M	54M	28M	15M	56M	10M	20M	17M	40M
Misp.	2418K	657K	168K	334K	682K	1241K	595K	2085K	6K	525K	250K	698K	225K	136K	522K

3.2. Diverge Branch and CFM Point Selection

The diverge branch candidates are determined based on profiling the code. A branch is marked as a possible diverge branch if it is responsible for at least 0.1% of the total number of mispredictions on the complete simulation.

A second profile run is performed to find the CFM points for each diverge branch candidate, based on the frequently executed paths. We select CFM points that show up on both paths of the diverge branch candidate as a reconvergence point for at least 20% of the dynamic instances of the branch. In addition, the CFM points cannot be farther than 120 dynamic instructions from the diverge branch.[13] If no CFM point is found to satisfy these constrains, then the diverge branch candidate is not marked as a diverge branch. For the basic diverge-merge processor, only the most frequent CFM point is marked. For the enhanced multiple CFM-points mechanism in Section 2.7.1, all CFM points that satisfy the requirements are marked.

As explained in this section, we are currently using very simple algorithms to determine the diverge branches and their

[13]These thresholds were chosen after considering different combinations of alternatives.

corresponding CFM points. An important part of our future work is to develop profiling algorithms that can better exploit the potential of the diverge-merge concept.

4. Results

4.1. Characterization of Mispredicted Conditional Branches

In Figure 6 we classify the dynamic mispredicted conditional branches into three categories: simple hammock diverge branches (simple if and if-else structures with no other control flow inside), complex diverge branches, and other complex branches. The diverge-merge processor can dynamically predicate simple hammock diverge branches, and complex diverge branches. Dynamic Hammock Predication [20] allows only simple hammock branches to be converted into predicated code. The results show that diverge branches account for more than 50% of the mispredicted branches in bzip2, mcf, parser, twolf, vpr, mesa and fma3d. Simple hammock branches by themselves account for a significant percentage of mispredictions only in two benchmarks: mcf (44%) and vpr (11%). On average, 57% of the branch mispredictions are due to diverge branches (including simple hammock branches) whereas 9% of the branch mispredictions are due to simple hammock branches[14]. Therefore, the diverge-merge processor has potential to eliminate more mispredictions than DHP. In gcc, most mispredicted branches are other complex branches, which can be eliminated neither by DHP nor by the diverge-merge processor. The control-flow graphs of these other branches are so complicated that even if the compiler considers only frequently executed paths it cannot identify any suitable CFM point with our current CFM point identification mechanism.

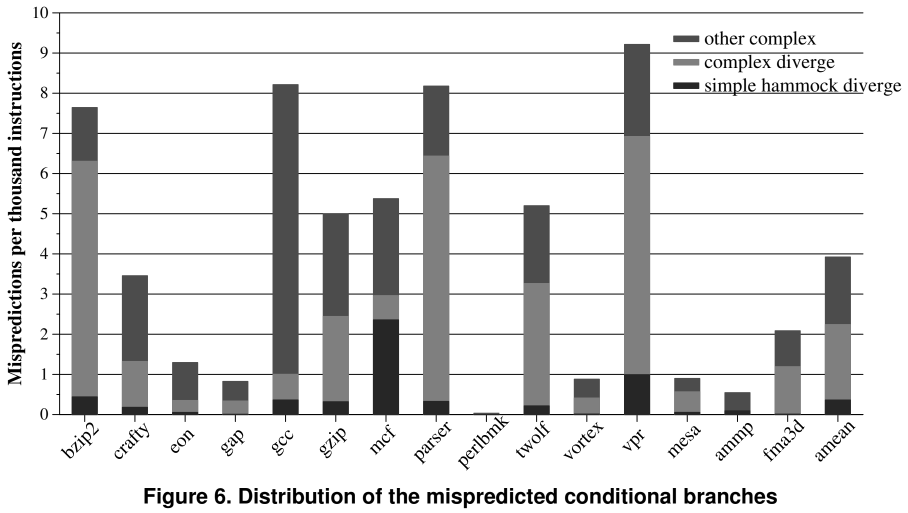


Figure 6. Distribution of the mispredicted conditional branches

4.2. Performance of the Basic Diverge-Merge Processor

Figure 7 shows the performance improvement over the baseline for DHP with a 1KB JRS confidence estimator [16] (DHP-jrs), DHP with a perfect confidence estimator (DHP-perf-conf), the basic diverge-merge processor (diverge-jrs), the basic diverge-merge processor with a perfect confidence estimator (diverge-perf-conf), and baseline with a perfect conditional branch predictor (perfect-cbp). With a perfect confidence estimator, DHP has 3.4% performance improvement potential, but

[14]We note that the fraction of simple hammock branches out of all mispredicted branches are similar to what was previously reported –about 10%– by Klauser et al. [20].

the diverge-merge processor has a 19% performance improvement potential over the baseline processor. As can be expected, the benchmarks with the highest performance improvement potential for the diverge-merge processor (bzip2, parser, twolf, and vpr) suffer from high branch misprediction rates and also have a higher percentage of diverge branches over the total mispredicted branches. The gcc benchmark also has a high branch misprediction rate, but neither DHP nor Diverge-Merge show performance potential for this benchmark due to the complexity of the control-flow graphs. Even though the conditional branch prediction accuracy of the baseline is 96.7%, the perfect branch predictor still can improve the performance of the baseline by 48% on average.

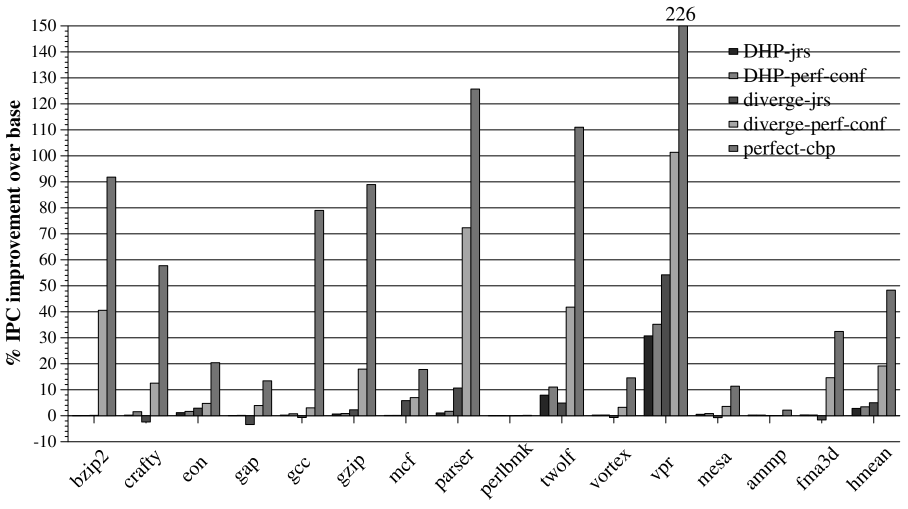


Figure 7. The performance of the basic diverge-merge processor

With a realistic confidence estimator, the average IPC improvement over all benchmarks is 2.8% for DHP and 5% for Diverge-Merge.[15] The diverge-merge processor improves the performance significantly on parser and vpr because more than 30% of the pipeline flushes due to branch mispredictions are eliminated with the dynamic predication mechanism. DHP with the realistic confidence estimator comes relatively close to the performance of DHP with a perfect confidence estimator. However, there is a significant difference between the performance of the diverge-merge processor with a real and perfect confidence estimator. One of the main reasons for this different behavior is that the number of instructions on the predicted and the alternate paths is small in DHP (since DHP predicates only simple hammock branches), so an incorrect confidence estimation does not significantly hurt performance. However, in Diverge-Merge, an incorrect confidence estimation could significantly hurt performance, especially if it leads to a large predication overhead as in case 3 in Table 1.

Figure 8 shows the distribution of exit cases in Table 1 for the basic diverge-merge processor. For some benchmarks (such as bzip2, gap, and gzip), cases 1 and 2 together account for less than 40% of all exits from dynamic predication mode even though diverge branches and merge points are selected based on frequently executed paths in the profiled code. The relatively infrequent occurrence of case 2, the best case for Diverge-Merge, is one reason why a realistic diverge-merge processor does

[15]Note that Klauser et al. [20] showed a 5.5% IPC improvement in IPC. However, their benchmarks, branch predictor, and the processor model were different from ours. They showed an average performance improvement of 5.5% over SPEC95 integer benchmarks with a 2K-entry hybrid (gshare+bimodal) predictor and 16-cycle misprediction penalty. To make a fair comparison, we implemented their mechanism in our machine model faithfully as described in their paper. Note that the fraction of simple hammock branches out of all mispredicted branches are similar to what was previously reported –about 10%– by Klauser et al.

not improve the performance of some benchmarks (such as bzip2 and gzip) even though they show potential. This problem can be reduced with a better compiler algorithm for identifying diverge branches and the corresponding CFM points. The multiple CFM-point mechanism in Section 2.7.1 could mitigate this problem as well. Diverge-Merge loses performance in gap because about 25% of the dynamic predication instances exit with case 3. The occurrence of case 3 can be reduced with the early-exit mechanism proposed in Section 2.7.2. A better confidence estimator will reduce the occurrence of cases 1, 3, and 5 because the processor would enter dynamic predication mode for fewer correctly predicted branches.

These results show that, for higher performance, Diverge-Merge requires techniques to reduce cases 1, 3, 5, and 6 in order to increase the occurrence of cases 2 and 4, which result in performance benefits. The following section shows the performance results of the enhanced diverge-merge processor that is designed to reduce the performance-degrading dynamic predication instances.

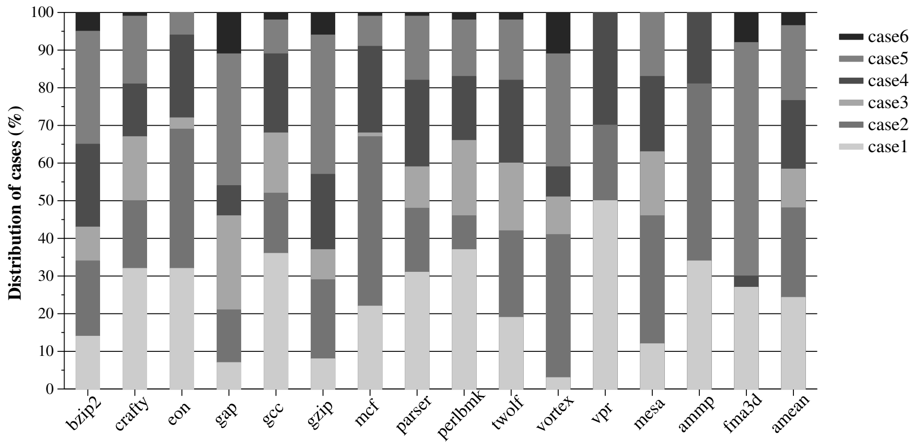


Figure 8. Distribution of exit cases for the basic diverge-merge processor

4.3. The Performance of the Enhanced Diverge-Merge Processor

Figure 9 shows the performance improvement of the enhanced diverge-merge processor. We show the % IPC improvement over the baseline for the basic diverge-merge processor (basic-diverge) and three variations of the enhanced diverge-merge processor after adding the enhancements described in Section 2.7 cumulatively to the basic diverge-merge processor: multiple CFM points (enhanced-mcfm), early exit policy (enhanced-mcfm-eexit), and multiple diverge branch policy (enhanced-mcfm-eexit-mdb).

Multiple CFM points increases the performance of the diverge-merge processor in bzip2, twolf, and fma3d, because this enhancement increases the probability of reaching a CFM point, which actually increases the occurrence of case 2 in these benchmarks, as shown in Figure 10. Using the early exit policy from dynamic predication improves the performance of especially crafty, gap, parser, twolf, and, mesa by reducing the occurrence of case 3. With the early exit policy, the frequency of case 3 is reduced from 10% to 3% on average as shown by comparing Figure 8 to Figure 10. The multiple diverge branch policy improves performance significantly for bzip2, parser, twolf, and vpr. Re-entering dynamic predication mode for a later diverge branch increases the probability of reaching the CFM point in these benchmarks.

With all the enhancements employed, the enhanced diverge-merge processor provides an average 10.8% IPC improvement over the baseline processor. The enhanced diverge-merge processor improves the IPC by more than 19% for the four benchmarks that have the highest IPC improvement potential as was shown in Figure 7 (bzip2, parser, twolf, vpr). Furthermore, using all the three enhancements eliminates the slight performance degradation incurred by the basic diverge-merge mechanism in three benchmarks (crafty, gap, fma3d). For the rest of the paper, *enhanced diverge-merge processor* refers to the diverge-merge processor that employs all the three enhancements (enhanced-mcfm-eexit-mdb).

Figure 11 shows the reduction in pipeline flushes due to branch mispredictions in the enhanced diverge-merge processor compared to the baseline processor. On average, 31% of the pipeline flushes are removed in the enhanced diverge-merge processor. In bzip2, parser, twolf, vpr, mesa, and fma3d, more than 40% of the pipeline flushes are eliminated. However, mesa does not show performance improvement because the number of control-independent instructions on the wrong path is relatively small, as we showed in Figure 1.

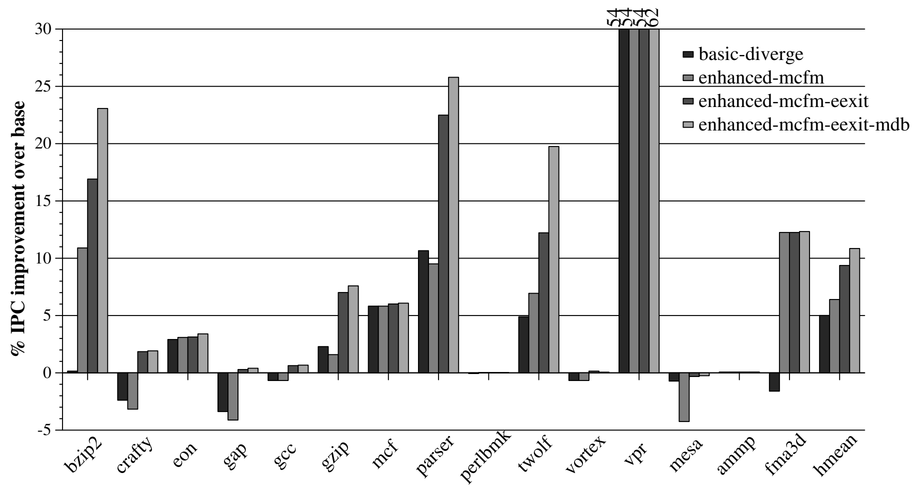


Figure 9. The performance of the enhanced diverge-merge processor

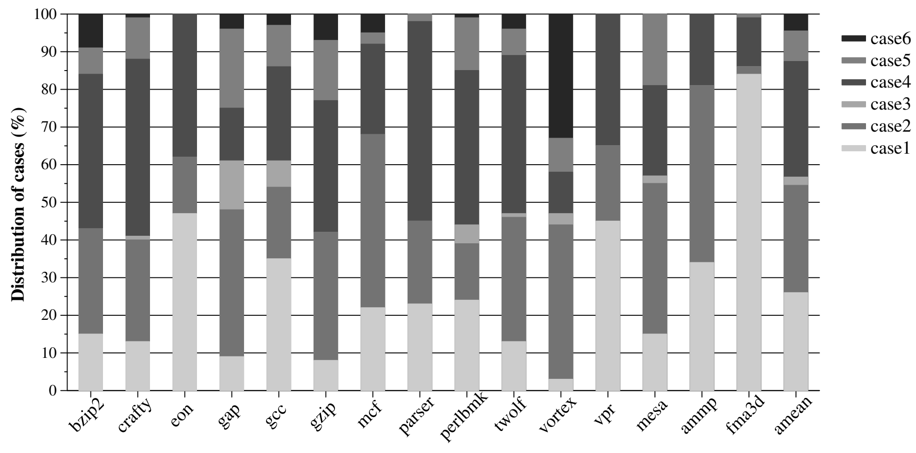


Figure 10. Distribution of exit cases for the enhanced diverge-merge processor

Figure 12 shows the overhead of dynamic predication on the enhanced diverge-merge processor. Even though the proces-

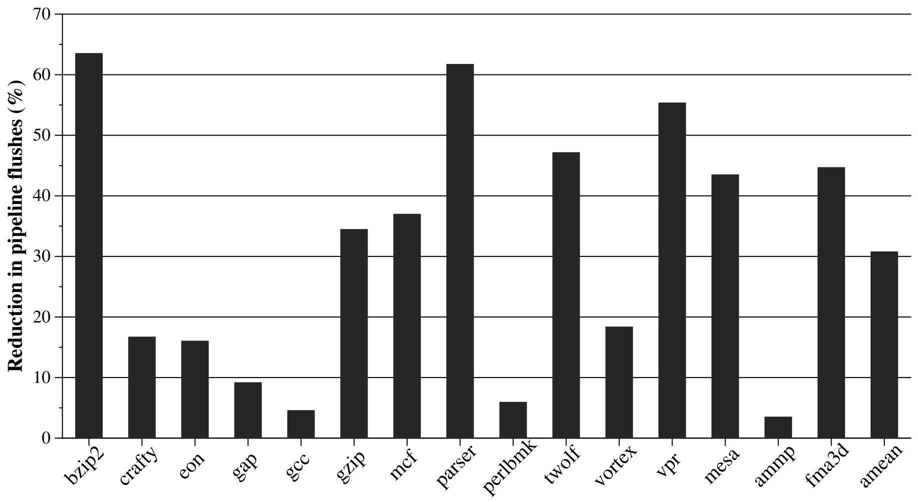
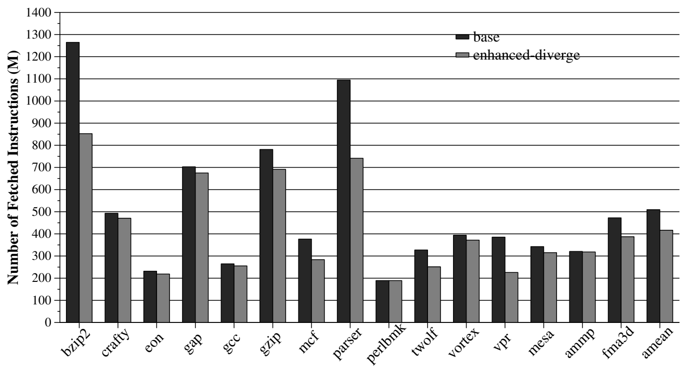


Figure 11. Percentage reduction in pipeline flushes on the enhanced diverge-merge processor compared to the baseline processor

sor has to fetch both paths of each dynamically predicated branch, the total number of fetched instructions decreases for every benchmark because the control-independent instructions are not flushed. On average, the enhanced diverge-merge processor reduces the number of fetched instructions by 18%. However, since the diverge-merge processor executes instructions even if their predicate values are FALSE, the processor executes more instructions than the baseline processor. Including the extra-uops used for dynamic predication (enter.pred.path, enter.alternate.path, exit.pred) and the select-$u$ops, the diverge-merge processor executes an average of 9% more instructions than the baseline. Improving the confidence estimator can reduce the number of correctly predicted branches that end-up being dynamically predicated, and therefore, it can reduce the overhead of executed instructions.



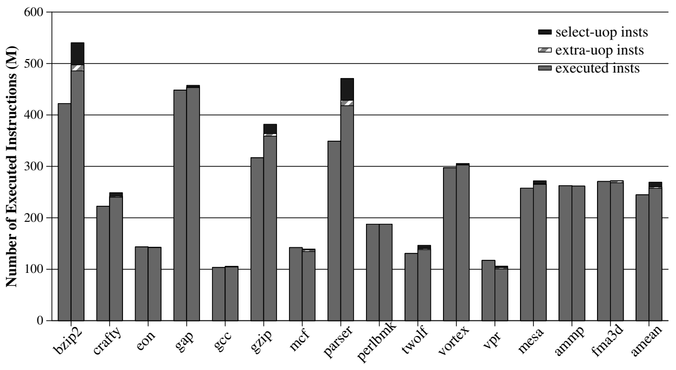


Figure 12. Number of fetched instructions (the left graph) and the number of executed instructions (the right graph) in the baseline (leftmost bars for each benchmark) and the enhanced diverge-merge processor (rightmost bars).

4.4. Effect of Instruction Window Size and Pipeline Depth

Figure 13a shows the performance of the enhanced diverge-merge processor and DHP on three different machines with 128, 256, and 512-entry instruction windows. The data is the average of all 15 evaluated benchmarks. Compared to the baseline processor, the enhanced diverge-merge processor improves the average IPC by 6.9%, 9.4%, and 10.8% respectively on a 128, 256, and 512-entry window processor. The enhanced diverge-merge processor provides larger performance improvements on processors with larger instruction windows because a large window can hold a larger number of control-independent instructions when the branch is mispredicted and the pipeline is flushed on the baseline processor. Therefore, the benefit of the enhanced diverge-merge processor increases because more useful work can be saved from being flushed.

Figure 13b shows the performance of the enhanced diverge-merge processor and DHP on three different 256-entry window processors with 10, 20, and 30 pipeline stages. Compared to the baseline processor, the enhanced diverge-merge processor improves the average IPC by 3.3%, 6.8% and 9.4% respectively on processors with 10, 20 and 30 pipeline stages. The performance benefit of the enhanced diverge-merge processor increases as the pipeline depth increases, since the branch misprediction penalty is higher on processors with deeper pipelines.

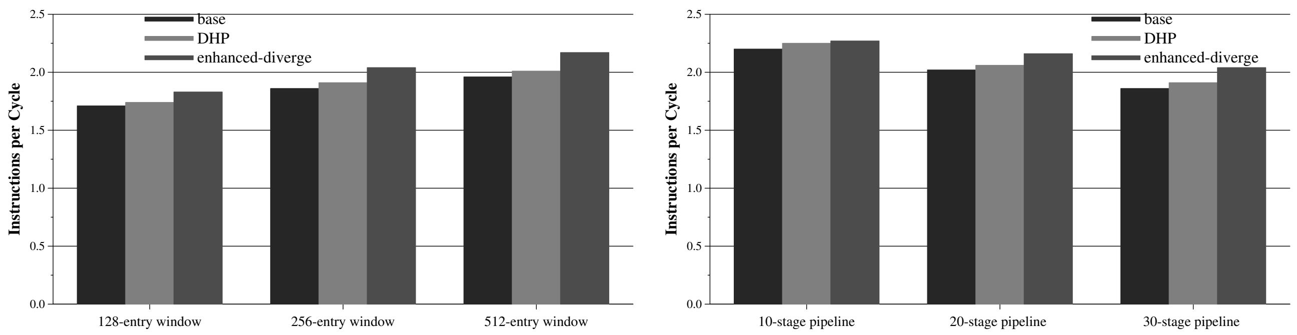


Figure 13. (a) Effect of instruction window size and (b) Effect of pipeline depth on diverge-merge processor performance

5. Related Work

5.1. Related Work on Dynamic Predication

Klauser et al. [20] proposed Dynamic Hammock Predication (DHP). After the DHP processor fetches a low confidence simple hammock branch (if or if-else structure with no other control flow inside), it fetches instructions from both paths of the branch. The instructions inside the hammock are dynamically predicated. After fetching from both paths, DHP inserts conditional-move (cmov) uops to form the correct data-flow dependencies. In contrast to the diverge-merge processor, which can dynamically predicate complex control-flow structures, DHP only works for simple forward branches, which constitute a much smaller subset of the control-flow graph and the branch mispredictions. Therefore, DHP provides smaller performance improvements compared to the diverge-merge processor as we showed in our results. In terms of hardware cost, both DHP and the diverge-merge processor require similar hardware complexity to support dynamic predication. Both mechanisms insert cmov or select uops to the pipeline and require the predicated instructions to carry the predicate register id.

5.2. Related Work on Compile-time Software Predication

Predicated execution has been studied intensively to reduce the branch misprediction penalty [24, 26, 33, 3] and to increase instruction-level parallelism [1, 15]. However, in a real IA-64 implementation, predicated execution was found to provide a small performance improvement [6]. This small performance gain is due to the overhead of compile-time predication, which sometimes offsets the benefit of reducing the pipeline flushes due to branch mispredictions. If a branch is predicated at compile-time, all the dynamic instances of that branch remain predicated, even though most of them would likely be correctly predicted by the branch predictor.

Recently, Kim et al. [19] proposed wish branches to reduce the overhead of predication by combining conditional branching and predication. With wish branches, the compiler generates code that can be executed either as predicated code or as code with conditional branches. At run-time, the hardware chooses between predicated execution and conditional branch prediction based on the run-time branch behavior. Like the diverge-merge processor, the wish branch mechanism also avoids the overhead of predication by using predication only for hard-to-predict branches at run-time. However, the diverge-merge processor overcomes the major disadvantage of wish-branches: the requirement for a predicated ISA. The wish branch mechanism is only applicable to ISA's that have full support for predication. In contrast to wish branches, the diverge-merge processor can be applied to any non-predicated ISA. Hence, the diverge-merge processor can be employed in a wider range of processors. Additionally, the diverge-merge processor provides the following benefits that are not possible with wish branches:

1. It can predicate complex control-flow graphs even if they include function-calls.
2. It doesn't fetch/execute ALL the basic-blocks between the low-confidence branch and the control-flow merge point (but wish branches have to). Diverge-merge fetches/executes only TWO paths.
3. It allows for dynamically choosing different control-flow merge points (see Section 2.7.1), whereas a wish branch has a single merge point that is determined statically.

5.3. Related Work on Multipath/Dual-path Execution

Starting with Riseman and Foster's eager execution [27] and the dual-path fetch in the IBM 360/91 [2], several contributions have been made in the field of multipath execution. See Uht [23] -a survey of multipath execution- for a good overview and comparisons. We will only review the work most relevant to the diverge-merge processor.

Any form of dual-path/multipath execution continues to fetch and execute from multiple paths even after a control-independent point is reached (for example, dual-path execution fetches/executes two copies of the same instruction, if the instruction is after a control-independent point). Compared to dual-path/multipath execution, the major contribution of our mechanism is that it exploits control-flow independence. In other words, the diverge-merge processor fetches/executes only one path after it reaches a control-independent point, whereas dual-path/multipath execution continues to fetch/execute from multiple paths. Thus, a dual-path/multipath execution processor wastes processor resources by keeping multiple copies of control-independent instructions in the processor, whereas the diverge-merge processor avoids this and allows processing resources to be available for instructions that are further in the instruction stream. As more than 60% of instructions after a mispredicted branch are control-independent (as was shown in Figure 1), the amount of work and processing resources wasted by dual-path/multipath execution is very significant especially if the processor's instruction window is large. Avoiding such waste using diverge-merge processing significantly improves performance.

Heil and Smith [14] and Farrens et al. [12] proposed a selective dual path execution mechanism that is restricted to low confidence branches. The processor starts fetching from both paths of a low confidence branch. The following low confidence branch either delays dual-path execution or is ignored until the first low confidence branch is resolved. When the low confidence branch is resolved, the instructions on the mispredicted path are discarded. These mechanisms have fairly high hardware cost because they increase the bandwidth and complexity of the fetch, decode, and execution stages of the pipeline to accommodate both paths simultaneously.

We simulated selective dual-path execution on our baseline processor model and found that it improves the average IPC by 2.6%. In contrast, the IPC improvement provided by dynamic hammock predication is 2.8% and by the diverge-merge processor is 10.8%, as we have shown in Section 4. Dual-path execution does not provide as significant a performance improvement as diverge-merge because it **always** wastes half of the fetch/execution resources even after a control-independent point. Dynamic hammock predication avoids this problem with dual-path execution, but it also does not provide as significant a performance improvement as diverge-merge because it is applicable only simple hammock branches rather than a large set of simple/complex control-flow structures. *Diverge-merge processing overcomes the major limitations of both dual-path execution and dynamic hammock predication by eliminating the waste of processing resources after a control-independent point AND by dynamically predicating a large set of mispredicted branches that exist in complex control-flow structures.*

Selective Eager Execution (SEE) on the PolyPath architecture was proposed by Klauser et al. [21]. PolyPath also fetches the most likely successors of each low confidence branch. Since it supports multiple outstanding divergent branches, it can execute along many different paths at the same time. The PolyPath architecture uses context tags to maintain path information for each instruction. A different tag is assigned to each path, so that only the instructions on the mispredicted paths are flushed when a divergent branch is resolved. Since there is no mechanism to maintain data dependencies, the instructions after the control independent point have to be fetched separately for each path, which significantly increases the contention for pipeline and execution resources, even more so than dual-path execution.

In general, multipath execution requires more resources than the diverge-merge processor to keep multiple paths in the instruction window, which increases the probability of having the correct path in the window when the branch is resolved, but also reduces the chance of getting a large number of control-independent instructions into the window. In contrast, a diverge-merge processor limits the execution to only two paths by following the branch predictor for successive branches in the dynamically-predicated paths. Furthermore, a diverge-merge processor makes more efficient use of the existing processor resources by time-multiplexing the two paths on the same hardware resources. Therefore, in contrast to proposals for multipath execution, there is no need to replicate or increase the bandwidth of portions of the pipeline.

5.4. Related Work on Control Flow Independence

Several hardware mechanisms have been proposed to exploit control flow independence [28, 29, 8, 5, 13]. These techniques aim to avoid flushing the processor pipeline when the processor is known to be at a control-independent point in the program when the branch is resolved. In contrast to the diverge-merge processor, these mechanisms require complex hardware to remove the control-dependent wrong-path instructions from the processor and to insert the control-dependent correct-path instructions into the pipeline after a branch misprediction. Hardware is also required to form correct data dependencies for the inserted correct path instructions. Furthermore, control-independent instructions that are data-dependent on the inserted or removed instructions have to be re-scheduled and re-executed with the correct data dependencies. The logic required for ensuring correct data dependencies for both control-dependent and control-independent instructions is complicated[28]. Exploiting control-flow independence in a trace processor is relatively simpler. Since traces already have live-in and live-out register information, the trace processor can easily detect the data-dependent instructions in the control-independent path [29]. However, instructions that had already been executed with incorrect data dependencies when the mispredicted branch was resolved still have to be re-executed with the correct data values.

Collins et al. [9] introduced Dynamic Reconvergence Prediction, a hardware-based technique to identify control reconvergence points (i.e., our CFM points) without compiler support. This technique can be combined with any of the mechanisms that exploit control-flow independence.

6. Conclusion and Future Work

This paper proposed the diverge-merge processor as an architecture for compiler-assisted dynamic predicated execution to reduce the branch misprediction penalty due to hard-to-predict branches. A diverge-merge processor dynamically predicates hard-to-predict instances of statically-selected diverge branches. The major contribution of our paper is a mechanism that enables the dynamic predication of branches that result in **complex control-flow structures** rather than limiting dynamic predication to **simple hammock branches**. We show that this benefit of the diverge-merge processor enables it to outperform a previously-proposed dynamic predication mechanism that can only predicate simple hammock branches by 7.8% on average for 15 SPEC CPU2000 benchmarks. Furthermore, the diverge-merge processor outperforms a baseline processor with an aggressive branch predictor by 10.8%.

The results presented in this paper are based on our initial implementation of the diverge-merge processor using relatively simple compiler and hardware heuristics and algorithms. The performance improvement provided by the diverge-merge processor can be significantly increased by future research aimed to improve these techniques. On the compiler side, better compilation and profiling techniques can be developed to select diverge branches and the corresponding control-flow merge points to maximize the benefits of dynamic predication. On the hardware side, better confidence estimators are worthy of research since they critically affect the performance benefit of dynamic predication. Hardware mechanisms to throttle dynamic predication based on dynamic information that becomes available (e.g., abandoning dynamic predication when the two control-flow paths are not likely to merge) are also promising to explore. In our future work, we intend to investigate these research areas, both on the compiler side and the microarchitecture side, to further improve the benefits of the diverge-merge processor by achieving larger reductions in the branch misprediction penalty.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[2] D. Anderson, F. Sparacio, and R. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, Jan. 1967.

[3] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Using predicated execution to improve the performance of a dynamically-scheduled machine with speculative execution. In *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1995.

[4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.

[5] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 4–15, 2001.

[6] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, 2001.

[7] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[8] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing*, pages 109–118, 1999.

[9] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *Proceedings of the 37th ACM/IEEE International Symposium on Microarchitecture*, 2004.

[10] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48, May 2005.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical Report RC14756, IBM, revised March 1991.

[12] M. Farrens, T. Heil, J. E. Smith, and G. Tyson. Restricted dual path execution. Technical Report CSE-97-18, University of California at Davis, Nov. 1997.

[13] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing branch misprediction penalty via selective recovery. In *Tenth International Symposium on High Performance Computer Architecture*, 2004.

[14] T. Heil and J. E. Smith. Selective dual path execution. Technical report, University of Wisconsin-Madison, Nov. 1996.

[15] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, 1986.

[16] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[17] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, pages 197–206, 2001.

[18] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[19] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches:combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th ACM/IEEE International Symposium on Microarchitecture*, 2005.

[20] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1998.

[21] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[22] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[23] A. K.Uht. *Multipath Execution*. CRC PRESS, 2005.

[24] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th Intl. Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

[26] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and dynamic branch prediction in dynamic ILP processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 120–129, 1994.

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

[28] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in superscalar processors. In *Proceedings of the Fifth IEEE International Symposium on High Performance Computer Architecture*, 1999.

[29] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, 1999.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[31] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, 2002.

[32] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[33] G. S. Tyson. The effects of predication on branch prediction. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 196–206, 1994.

[34] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, Apr. 1996.