UNIVERSITY OF CALIFORNIA, SAN DIEGO

Turning Predicate Information to Advantage to Improve Compiler Scheduling and Branch Prediction

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science and Engineering

by

Elizabeth A. Simon

Committee in charge:

Professor Jeanne Ferrante, Chair
Professor Bradley Calder, Co-Chair
Professor Larry Carter
Professor Pamela Cosman
Professor Fan Chung Graham

2002

The dissertation of Elizabeth A. Simon is approved, and it is acceptable in quality and form for publication on microfilm:

Co-Chair

Chair

University of California, San Diego

2002

To my parents, Lyle and Rita Simon, for providing the experiences and motivation that have made me the person I am today. I am especially grateful for the variety and diversity of the people to whom they exposed me while I was growing up. I consider that exposure to be the major contributor to my constant desire to consider, investigate, and then pursue exactly what it is I want in life. I am indebted to my entire graduate school experience for bringing home to me, yet again, what my parents instilled in me – to accept that what I want is a more important and meaningful pursuit than what other people might want for me.

To Jeanne and Brad, the tag-team of advisors who let me selfishly demand a co-advising relationship which allowed me access to each of their individual strengths. They'll be chagrined (I'm sure) to hear that I highly recommend it to anyone who finds themselves in a similar position to mine. I found the relative costs of the two-advisor system were vastly outweighed by the benefits of reduced branch mispredictions, ahem, I mean having such a variety of experience and expertise in my corner.

To the Arch Lab in all its incarnations up to this point in its history: Glenn, Chandra, Barbara, Lori, John, Tim, Eric, Suleyman, Jamison, Jeff, Wei, Erez, Floria, Satish, and Rakesh. As a great believer in the importance of the lab experience - you guys have been a great lab. But I'm taking the ball anyway.

To Chris, for everything. (OK, but especially, patience, bicycles, patience, laptop repair, patience, boring proof-reading assignments, patience, and patience.)

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgments

The text of Chapter IV is in part a reprint of the material as it appears in the International Journal of Parallel Programming, Volume 28, Issue 6, pp. 563-588. The dissertation author was a co-primary researcher and author (with Lori Carter) and the other co-authors listed on this publication ([18]) directed and supervised the research which forms the basis for Chapter IV.

The text of Chapter V is in part a reprint of the material as it appears in the proceedings of the 1st Workshop on EPIC Architectures and Compiler Technology. The dissertation author was the primary researcher and author and the co-authors listed on this publication ([40]) directed and supervised the research which forms the basis for Chapter V.

VITA

April 26, 1973	Born Columbus, IN
1991	High School Diploma, Brown County High School Nashville, IN
1995	B.S., University of Dayton Dayton, OH
1998	M.S., University of California San Diego, CA
1999-2000	Teaching Assistant, Computer Science and Engineering Department, University of California San Diego, CA
2000	Internship, Intel Corporation Santa Clara, CA
2002	Doctor of Philosophy, University of California San Diego, CA

PUBLICATIONS

"Incorporating Predicate Information Into Branch Predictors." Authors: B. Simon, B. Calder, J. Ferrante. In the Proceedings of the First Workshop on EPIC Architectures and Compiler Technology, Dec. 2001.

"Path Analysis and Renaming for Predicated Instruction Scheduling." Authors: L. Carter, B. Simon, B. Calder, L. Carter, J. Ferrante. In the International Journal of Parallel Programming, Dec. 2000.

"Predicated Static Single Assignment." Authors: L. Carter, B. Simon, B. Calder, L. Carter, J. Ferrante. In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 1999.

"Schedule-Independent Storage Mapping in Loops." Authors: M. Mills Strout, L. Carter, J. Ferrante, B. Simon. In the Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1998.

Field Of Study: Computer Science

ABSTRACT OF THE DISSERTATION

Turning Predicate Information to Advantage

to Improve Compiler Scheduling and Branch Prediction by

Elizabeth A. Simon

Doctor of Philosophy in Computer Science and Engineering

University of California, San Diego, 2002

Professor Jeanne Ferrante, Chair

Professor Bradley Calder, Co-Chair

Architectural support for predicated execution has been proposed as a manner of attacking performance bottlenecks resulting from modern processor pipeline design. Predication is the process of removing control flow in a program and replacing it with predicate define data flow. Predication has the potential to reduce performance penalties from mispredicted branches and to make greater use of functional unit resources. However, this potential can be squandered or even turned into a performance loss if both the compiler producing the predicated code and the hardware executing the predicated code are not cognizant of the relationships between and values expressed by the predicates.

In this dissertation we expose the importance of utilizing predicate information in both the compiler, for scheduling predicated code, and in the architecture for maintaining branch prediction accuracy in the face of predicated code. For the compiler, we define *Predicated Static Single Assignment* (PSSA). PSSA is a compiler-internal representation which maintains full-path information about predicates and their relationships. This complete information enables us to implement predicate-sensitive scheduling optimizations like *predicated speculation* and *control height reduction* that are not possible with less-complete predicate knowledge systems.

In the hardware, we show the negative impact that predicate region formation can have on the predictability of branches in the program. We specifically evaluate the impact of *misprediction migration* - the problem of increased mispredictions from branches located within predicated regions. We evaluate several predicate update branch prediction architectures targeted at both local and global prediction schemes with a goal of utilizing available predicate define information to restore high branch prediction accuracies.

Chapter I

Problem Statement

Architectural support for predicated execution has been proposed as a manner of attacking performance bottlenecks inherent in current processor pipeline designs. In order to achieve processors with high clock frequency, processor pipelines have increased in length - requiring more stages for a given instruction to complete execution and allowing more instructions to be in-flight in the pipeline at one time. This causes higher penalties for mispredicted branches as more work will have been fetched from the wrong path of execution by the time a branch is resolved at the end of the pipeline.

If-Conversion-based predication is the process of removing control flow in the form of branches and replacing it with predicate define data flow. Rather than predicting a branch and executing the predicted path, both paths of execution following a control flow branch point are fetched and executed. The instructions on these paths are tagged with a guarding predicate - implemented as a single-bit predicate register. The value of this register is used when the instruction reaches the end of the pipeline, at which point, a value of false in the guarding predicate causes the instruction to be nullified. In essence, predicated execution provides an alternative to branching in supporting the conditional execution of program code. The difference from traditional conditional execution is that knowledge of the condition is not required until the end of the pipeline. The potential benefit comes from branches where the conditional result frequently could not be guessed correctly at the beginning of the pipeline, known as *hard-to-predict branches*. However, the cost lies in executing both paths following a control flow branch, knowing that the instructions from one of the paths will be nullified. We call instructions from paths nullified at the end of the pipeline *spurious instructions* since their

execution is not required for program execution.

It is the role of compiler to decide what branches are transformed into predicate defines and, consequently, what instructions are guarded on predicates. This process merges together basic blocks from the original, non-predicated code into a larger *predicated region* of code - replacing control flow relationships with predicate data dependences. In this process the compiler tries to maximize the reduction of mispredicted branches (by translation of branches to predicate defines) while minimizing the impact of spurious instruction execution. This second goal is both critical to performance and complicated by the predicate region formation process itself. A compiler needs to aggressively optimize and schedule predicated code in order to mitigate the impact of spurious instruction execution. Traditional compiler analysis, optimization, and scheduling is centered around the control flow graph infrastructure. However, in predicated regions this infrastructure has been modified. Predicate relationships replace control flow and knowledge of these relationships must be obtained and used for correct, much less optimized, region code production. For example, a predicated region may contain two definitions of a variable `x` - one from the "then" portion of a conditional statement and one from the "else" portion. While these instructions will fall in some linear ordering of instructions inside the predicated region, one must know that these statements' guarding predicates are *disjoint* to realize that neither of the defines of `x` influences each other. In fact, one would want to allocate the same register to both of these defines so that uses downstream from them (in code reachable by both after the if statement) don't require a register move.

We will show that, more than just disjointness information, *full-path predicate* relationship information is needed to allow the most aggressive predicate region optimization and scheduling by the compiler. We define *Predicated Static Single Assignment* (PSSA), a compiler-internal representation which maintains this full-path information about predicates and their relationships. We show how to utilize this full-path information to implement predicate-sensitive scheduling optimizations like *predicated speculation* and *control height reduction* that are not possible with less-complete predicate knowledge systems.

However, the compiler is only part of the performance picture when it comes to predicated execution. It was original inadequacies in the processor's hardware branch prediction scheme that triggered the development of if-conversion via predicated execution. This branch prediction hardware will also need to utilize predicate information to achieve high performance when executing predicated regions.

In much the same way the compiler has assumptions about branches being the sole

indicator of control flow relationships, so has hardware. In order to predict branches hardware needs to manage control flow so as to optimize the time the processor pipeline stays full and productive. Branch prediction hardware relies on detecting current control flow information and using it to predict future control flow decisions.

Since predicated regions now encode control flow information in the form of predicate define statements, it follows that predicate information will be paramount for effecting branch prediction decisions.

The most obvious control flow information imparted by predicate defines is knowledge about spuriously executed instructions. Specifically, predicate information will be crucial for detecting *spurious branches*. More so than other spurious instructions (which simply take up pipeline resources), spurious branches can degrade performance by causing additional branch mispredictions. Additionally, control flow information now encoded in predicate defines may provide important correlative control flow history that is useful in prediction of even non-spurious branches in the code. In modern branch prediction schemes, a branch (`br1`) that, when taken, sets the variable `x` to zero, can provide information that allows a branch `br2 (x==0)` to be predicted very accurately. If `br1` is a hard-to-predict branch translated to a predicate define, then this information will be lost to the branch prediction hardware, yielding frequent mispredictions from `br2 (x==0)`. Both of these issues can lead to increased branch mispredictions of branches in predicated regions. This problem is known as *misprediction migration*.

We propose several schemes which utilize predicate information in order to improve branch prediction for predicated codes. We investigate Squash-FP - a low cost (in terms of implementation) scheme for reducing the impact of spurious branches by eliminating the prediction of branches that can be determined to be spurious at the beginning of the pipeline. The compiler enables this technique by scheduling code so that predicate defines can finish execution before branches guarded on them are fetched into the pipeline. We also explore methods of incorporating predicate define information into traditional branch prediction architectures to restore the control flow information hidden by the process of predicate define transformation. This reduces the impact of misprediction migration due to predicate region formation and execution.

Chapter II

Background

II.A Modern Computer Architectures as Pertains to Predicated Execution

In the search for ever increasing performance, processor clock frequencies have continued to increase. The need for high frequency processors has driven a general increase in processor pipeline depths. While pipelining is certainly beneficial to processor performance, increasingly deeper pipelines begin to exacerbate some of the current well-known performance bottlenecks.

Consider the two processor pipelines shown in Figure II.1. We will use the terminology `fetch, decode, execute, commit` to discuss the process of executing an instruction in a pipelined processor, though as we consider modern pipelines there may be more than one "stage" to each of these phases. One of the primary performance drawbacks to pipelined execution is that of the control hazard. A control hazard is caused by a control-flow-changing instruction in a program's otherwise sequential instruction stream. When a branch is executed, there is a chance (when the branch is "taken") that the next instruction we wish to process is *not* the instruction that sequentially follows the branch. In a non-pipelined processor, this is not a problem. Once we are done executing the branch instruction, we know if, indeed, the branch is taken and what the address of the next instruction to be fetched for execution is. However, consider the pipeline in Figure II.1(a). After a branch is fetched into stage 1, in the next cycle, it will be decoded (stage 2). While it is being decoded we would like to fetch another instruction into the first stage of the pipeline, but we don't know the address

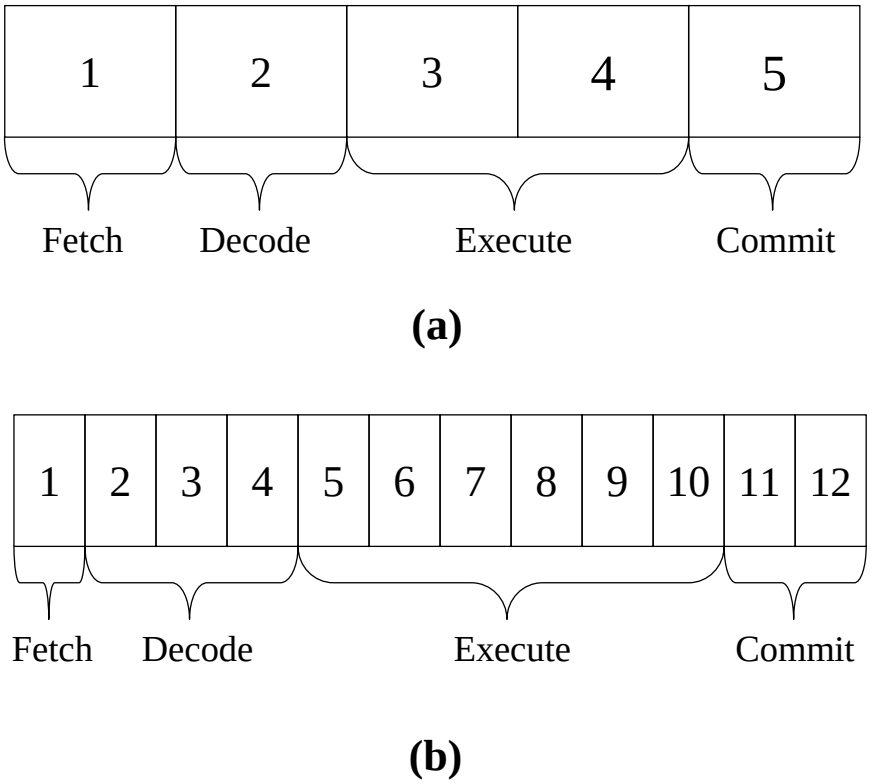


Figure II.1: Two fictional processor pipelines. Branches cause problems for pipelines since they determine the address of the next instruction that should be executed. In general, the longer the pipeline, the greater the hazard from branches.

of the next instruction to fetch. We must evaluate the branch condition in order to know if the instruction after the branch should be fetched next (i.e., the branch is not taken) or if we should jump to a different address in the code (i.e., the branch is taken). The value of the condition will not be known until the end of the execute phase, at the end of stage 4 in this example.

There are a variety of ways in which the pipeline can treat a branch in the fetch stage. The simplest choice is to do nothing, and simply wait for the branch to finish execution before beginning to fetch the next instruction. However, analysis on the SPEC integer benchmarks has shown that on average a branch is encountered every five instructions [16]. Clearly, for longer pipelines like that in Figure II.1(b) much time will be spent idle if we merely wait for a branch to finish execution. One solution to this is to *predict* the direction that a branch will go (i.e. will it be taken or not) upon fetching that branch. Then, at least in the case we predict correctly, we will continue to fill the processor with useful work in the face of a branch. Section II.C will discuss branch prediction in more detail.

Clearly, this indicates that the better we can predict branch behavior, the faster our code will execute. However, one can not always predict the direction of each branch perfectly. In those cases where we mispredict the direction of a branch, we suffer a *branch misprediction penalty*. This occurs when all of the instructions following a branch must be flushed out of the pipeline without committing - as they were not supposed to be executed. The size or length of this penalty is usually expressed in terms of the number of cycles that pass between the fetch of the branch and the fetch of the first instruction from the correct direction following the branch. For example, for the pipeline in Figure II.1(a), at the end of stage four we know if a branch was mispredicted, so the misprediction penalty would be three cycles. That is, the instructions currently being processed in stages one through three are instructions on the wrong path of execution and should be nullified. The next cycle will fetch the first instruction from the correct path into stage one and stages two through four will be idle. Correspondingly, the misprediction penalty for the longer pipeline in Figure II.1(b) would be nine cycles. Again, this cost is related to the length of the pipeline and, in general, increases as processor pipelines grow longer.

The combination of these effects makes poorly predictable branches an increasing performance bottleneck in modern processors and was a major catalyst for the development of predicated execution.

At the same time that clock frequencies are increasing, transistor process sizes are

decreasing, allowing more and more room on a chip. While there are many trade-offs to consider in deciding what additional resources to provide in this extra space, frequently the decision is made to increase the processor pipeline width - allowing multiple instructions to be processed in parallel in the pipeline. This widening of the pipeline includes the addition of multiple versions of functional units to serve instructions scheduled in parallel. While this provides the potential for suitably reducing the time required to complete the execution of a series of statements on the multiple functional units, this speedup is only possible if the work can be scheduled concurrently. If there are data dependences between the instructions (i.e. the input of one instruction is dependent on the output of another instruction) or if control flow restrictions due to pipelining occur, then the potential decrease in execution time may not be realized.

We will see in the next section that predicated execution takes advantage of functional units being available in the processor. It can provide "parallelism" across multiple control flow streams of execution, which does not lengthen the execution time of the longest of the streams if enough functional units are available.

II.B Background on Predicated Execution

Predication is the process of replacing branch instructions with predicate define instructions, then guarding the execution of the instructions originally control dependent on that branch with the values defined by the predicate define. In essence, it removes physical control flow instructions that change the address of the next sequentially executed instruction. Correctness is maintained by inserting data dependences via predicate values that can control the result of the "execution" of instructions. In an architecture supporting predicated execution each operation is guarded by a one-bit predicate register. An operation is fetched and executed regardless of the guarding predicate value, but committed only if the value of its guarding predicate is true. If an operation's guarding predicate is false, the result of its execution is ignored and not allowed to modify processor state. In this way, instructions may be fed into the processor pipeline before it is determined whether the results of these instructions should update the program state. Additionally, an instruction set architecture (ISA) supporting predicated execution must provide a set of predicate-defining instructions which evaluate a conditional expression and set the value(s) of predicate registers.

A simple example is shown in Figure II.2. The process of predication would transform

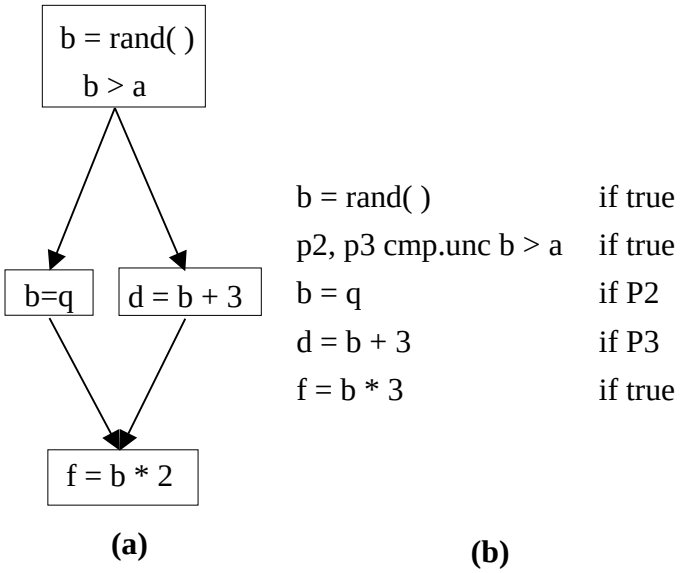


Figure II.2: A basic example of the transformation of non-predicated code into a predicated region. The branch determining flow of control in (a) is replaced by the predicate define which sets two predicate values (`P2` and `P3` in (b)). All instructions in (b) are fetched into the pipeline, but only those whose guarding predicate is true are committed at the end of the pipeline.

the Control Flow Graph (CFG) comprised of four basic blocks in Figure II.2(a) into the predicated code without control flow shown in Figure II.2(b). In this code, all operations are now guarded, either by a predicate register set to the constant value of true, or by a register that can be defined as either true or false by a `cmp` or *compare* operation. Operations guarded by the constant true, such as the operation `f=b*2` in Figure II.2, will be executed and committed regardless of the path taken. Operations guarded by a predicate register, such as the operation `b=q`, will be put into the pipeline, but only committed if the value of the operation's guarding predicate (`P2` for this operation) is determined to be true.

This process of replacing branches with predicate defining instructions and associating operations with a predicate defined by that compare is known as *If-Conversion* [7, 34]

II.B.1 Benefits of predicated execution

Predicated Execution has the potential to provide performance benefits in a variety of ways for architectures with pipelined processor cores.

One advantage is that predicated execution can eliminate the need to guess or predict a branch direction. In architectures with a deeply pipelined processor with multiple functional units available, it may be less costly to execute both paths of instructions following a branch

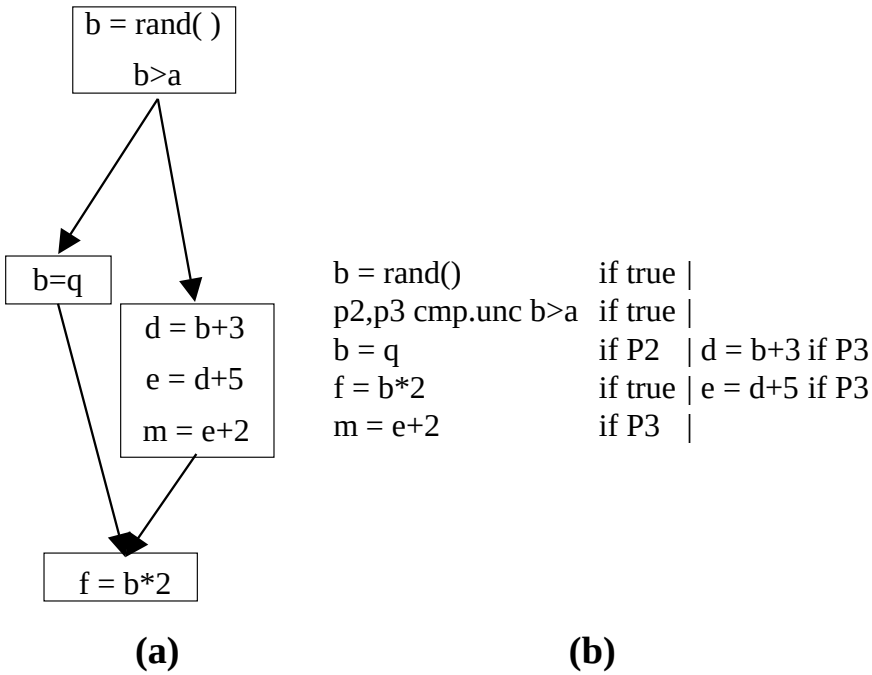


Figure II.3: A potentially poor region to predicate. Because of the uneven lengths of the dependence paths in blocks `P2` and `P3` forming these blocks into a single predicated region will cause a performance penalty whenever path `P2` should be executed. (b) shows the predicated region code scheduled for a processor with 2 issue slots per cycle. A possible optimization could alleviate the performance penalty for path `P2` if code from after `f = b*2` can be scheduled in parallel with `m = e+2 if P3`.

than to take the chance of incorrectly predicting the direction of a branch. For a processor with three ALUs and the pipeline shown in Figure II.1(b) this would be the case when executing the code from Figure II.2. In this case, the processor has the capacity to execute all the instructions corresponding to both paths following the branch within one cycle. These instructions (`b=q` and `d=b+3`) are independent of each other and may be executed in parallel. If branch `b>a` is *ever* mispredicted in the non-predicated code, then the predicated code will execute more quickly than the non-predicated code. The predicated code has traded the cost of never having to pay a branch misprediction penalty for the cost of executing additional instructions on every pass through this region.

In any given execution of the predicated code, some of the instructions will be executed but not committed (if their predicates were not true). We call these instructions *spurious instructions*. While, in this example, the execution of spurious instructions does not impact the scheduling of the code, an example like Figure II.3 shows a potential pitfall. In this case,

because of the un-even workload along the two paths following the branch, the execution of spurious instructions along path `P3` may impose a performance hit on the execution of the region when path `P2` is the true path. In this way, characteristics concerning the architectural pipeline depth, available functional units, and code characteristics must be considered when evaluating the potential benefit of predication.

Other advantages may be gained by combining both paths of a branch into a single path via predication. Combining several smaller basic blocks into one larger predicated region provides a larger scheduling region. [31]. This can allow for additional optimizations as well as providing a larger pool from which to draw *instruction level parallelism* (ILP) [17, 31]. In the example shown in Figure II.2(b) a processor with more than one ALU could execute both `e=d+5` and `f=b*2` in parallel, providing a decrease in the time required to execute this code. Additionally, code motion techniques may be more easily applied to predicated code. This will be discussed in depth in Chapter IV.

One specific scheduling optimization where predicated execution's ability to remove branches has been used successfully is software pipelining [21, 35] - a very profitable loop-scheduling optimization. In this case, predicated execution is used to remove branches in loop bodies, which then enable those loops to be scheduled with a software pipelining scheduler. Currently, software pipelining cannot correctly schedule loop bodies containing branches.

Removing hard-to-predict branches (and branches in general) is also beneficial in reducing dynamic execution time and is discussed in depth in Section II.C.

II.B.2 Architectural support

Machines with hardware to support predicated code include an additional set of one-bit registers called predicate registers. The process of predication replaces branches with compare operations that set predicate registers to either true or false based on the comparison in the original branch. Each operation is then associated with one of these predicate registers which will hold the value of the operation's *guarding predicate*. The operation will be committed only if its guarding predicate is true[1].

Additionally, hardware support must be provided for the execution of instructions that may need to be nullified. First, instructions can begin execution in the pipeline even when the value of their guarding predicate is not yet available. However, this guarding predicate

[1] One exception is the unconditional definition of a predicate. This is discussed later in the section.

must be known before any forwarding of the defined value is allowed in the pipeline. Before any instruction is allowed to modify processor state or commit (i.e. store to memory or update the register file), its guarding predicate must have evaluated to true. Finally, in order to avoid excess memory system traffic, any load which misses in the data cache will not trigger a cache miss until the load's guarding predicate has resolved to true.

EPIC Technology and PlayDoh versus IA-64

Predicated Execution has been implemented in the Explicitly Parallel Instruction Computing (EPIC) architecture technology as part of its design for achieving the ILP needed to keep increasing future processor performance [10, 24]. The EPIC architecture design was developed as a joint venture between Intel and Hewlett-Packard. The goal of the EPIC architecture design is to allow the compiler to more directly expose, enhance, and exploit parallelism by making it more explicit in the machine code [1]. An EPIC architecture issues wide instructions, similar to a VLIW architecture, where each instruction contains many independent operations. All of the operations in each instruction are potentially executed "in parallel".[2]

Additionally, EPIC supports predicated execution for removal of hard-to-predict branches to increase ILP and support code scheduling freedom that can result from larger predicated regions created from smaller basic blocks. EPIC architectures will provide compiler support for the definition of 1-bit registers in a predicate register set and the guarding of instructions with a predicate register value.

In the process of the development of the EPIC technology, two Instruction Set Architecture (ISA) implementations have been developed which fulfill a subset of the proposed EPIC architecture features. The first, and more complete, is the HP PlayDoh ISA. The second is the Intel IA-64 ISA. While less complete, it has been implemented in the first EPIC-technology processor, Intel's Itanium [2].

PlayDoh

PlayDoh is a parameterizable ISA developed for research into ILP. Here we will focus on PlayDoh's implementation as relates to predicated execution, but complete details on PlayDoh can be found in [26].

Branches in PlayDoh are broken into three steps (two for unconditional branches).

[2]Stop bits in an IA-64 instruction can delineate non-parallel work in a single instruction. This is discussed further later in the section.

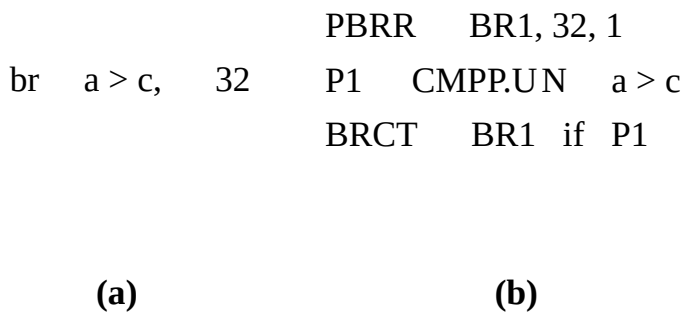


Figure II.4: An example of the branching mechanism employed by the PlayDoh ISA. A traditional branch is shown in (a) which branches to PC+32 if `a>c`. In PlayDoh, this branch is broken into three components. The first is used to set a branch register with the target address if the branch is taken. The second evaluates the condition and stores the result in a predicate register. The third causes the actual change in control flow by jumping to the address stored in `BR1` if `P1` is true.

This breakdown is designed to reduce the interruption in control flow that branches can cause by allowing the different pieces of information that contribute to a branch's outcome to be calculated as soon as possible. Figure II.4 shows the translation of a traditional PC-relative branch into a 3-part PlayDoh operation sequence. The first operation of this sequence (`PBRR` - prepare to branch) sets a special branch register with the target address of the branch if it is taken. The second operation performs the conditional evaluation and places the result (true or false) into a 1-bit predicate register. The third operation actually performs the control flow change in the case where the branch is taken. This third operation is only dependent on the value of the predicate register to control its operation.

In the work presented here, we do not investigate the issue of branch target detection. Hence, in later examples, the first part of the PlayDoh branch series (`PBRR BR1, 32,1`) is elided for brevity.

While PlayDoh already makes use of predicate registers in its implementation of traditional branch instructions, PlayDoh also supports predicated execution via the same "compare and put result in predicate register" instructions and by attaching a guarding predicate source operand to operations.

Consider a PlayDoh operation `B,C cmpp.un.ac a>c if A` as an example. The `cmpp` operation can define one or two predicates. This operation will define predicates `B` and `C`. The first tag (`.un`) applies to the definition of the first predicate `B` and the second tag (`.ac`) to `C`. The first character of a tag defines how the predicate is to be defined. The character `u` means that the predicate will unconditionally get a value, whether the guarding predicate (`A` in this

Guarding Predicate	Result of Comparison	Result				Complement of Result			
		UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	-	-	-	0	-	-	-
0	1	0	-	-	-	0	-	-	-
1	0	0	0	-	0	1	1	1	-
1	1	1	1	1	-	0	0	-	0

Figure II.5: Outline of the behavior of predicate defining instructions in PlayDoh. Every predicate define can potentially define two predicates. Each predicate's definition is determined, in part, by one of eight tags: UN, CN, ON, AN, UC, CC, OC, or AC. The result of the definition is determined by the value of the define's guarding predicate, the evaluation of the condition, and the meaning of the tag. A - in the table means the current value of the result predicate is unchanged.

case) is true or false. If A is false, then B is set to false. Otherwise, A is true and the value of B depends upon the evaluation of a>c.

The character a in the second tag (.ac) indicates that the full definition of the related predicate C is contingent on the value of A, the evaluation of a>c, AND the prior value of C. If A is false, the value of predicate C does not change. If A is true and C has previously been set false then C remains false. Additionally, the second character of a tag defines whether the normal (n) result of the condition (a>c) or the complement (c) of the condition must be true to make the related predicate true. If A is true and C is true and !(a>c) is true then the new value of C will be true[3].

A listing of the predicate defining instructions supported in PlayDoh ISA is shown in Figure II.5.

IA-64

Relative to PlayDoh, IA-64 implements a somewhat reduced set of predicate defining instructions. Not all possible combinations of conditional, unconditional, disjunctive, and conjunctive predicate definitions are available. In addition, the defining condition may be limited (most notably, sometimes to a compare to zero). However, expressiveness is not curtailed, though a predicate define that may be done in one operation in PlayDoh, may require a series of operations to be implemented in IA-64. A subset of the predicate defining instructions provided in IA-64 and their nomenclature is presented in Figure II.6. Please note that in IA-64

[3]Conversely, if A is true and C is true and !(a>c) is false, then the new value of C will be false.

tag	GP == 0		GP == 1			
			comp == 0		comp == 1	
	P1	P2	P1	P2	P1	P2
.unc	0	0	0	1	1	0
.and	N/C	N/C	0	0	N/C	N/C
.or	N/C	N/C	N/C	N/C	1	1

Figure II.6: Outline of the behavior of a subset of the predicate defining instructions available in the IA-64 ISA. Here a single tag applies to the entire define instruction and determines the setting of two predicate registers. N/C means the current value of the result predicate is unchanged.

the guarding predicate is referred to as the "qualifying predicate", though we will prefer to use the first term throughout the rest of this work.

Two other differences between IA-64 and PlayDoh will have bearing on the work presented here. First, IA-64 does not utilize the multi-phase branch instruction that PlayDoh does. In a multi-phase branch, the branch's execution may be dependent on both the guarding predicate and a conditional instruction - all found in the one branch instruction. In IA-64, for a branch to be "taken" both the guarding predicate and the condition must evaluate to true. If the condition is true but the guarding predicate is false, evaluation of the branch is "nullified" meaning that the correct behavior for the branch is to fall-through.

Second, while all operations in a traditional VLIW instruction are supposed to be independently executable, IA-64 introduces the *stop bit* which can be inserted to show sections of the VLIW instruction which are not independent. In essence, considering a stream of IA-64 bundles, any sequential operations not interrupted by a stop bit are independent of each other - regardless of bundle location. In actual execution, after instructions are fetched into the pipeline, they are dynamically divided in stop bit locations into *instruction groups* which contain independently executable operations. This annotation device allows static code to be stored more compactly, but is important to understand when reading IA-64 style code.

In most of the examples in this work, we tend to show the IA-64 style operations independently (sequentially) rather than as they would be bundled into an IA-64 instruction.

This more clearly matches the traditional manner of viewing a stream of machine instructions and is easier to read.

II.B.3 Compiler Support for Predicated Execution

Though we have to this point focussed on the necessary hardware support required for the implementation of predicated execution, we now turn to the actual producer of predicated code – the compiler. As proposed in the EPIC technology goals, predicated execution is one way for the *compiler* to make additional parallelism directly visible to the architecture. No change in high-level coding languages is required or even suggested to support use of predicated execution. It is the job of the compiler to decide when it will be beneficial to predicate and what paths of execution should be combined to best utilize the resources of the machine.

The introduction of support in hardware for predicated execution has a dramatic effect on the role and job of the compiler. First, it will be the job of the compiler to decide when and where to predicate code. Since predication involves trading the cost of execution of multiple execution paths for the reduction of branch misprediction penalties, not all branches should be if-converted to predicate defines. The process of *predicate region formation* (or just region formation) involves deciding what branches to if-convert and how many subsequent basic blocks to guard on those predicate values. A compiler's decisions with regards to region formation can make the difference in whether the application of if-conversion is a performance improvement or impediment.

Additionally, traditional compiler techniques for analyzing, manipulating, optimizing, and scheduling code have revolved around the control flow graph format. In this, code is organized into *basic blocks* which are straight-line sequences of instructions connected by edges that represent the order of execution of these basic blocks. This guarantees that once one instruction in a basic block is executed, that all instructions in that basic block are executed. Additionally, this separates the concern of non-linear control flow into dealing with the inter-relationship of basic blocks only. Predicated execution replaces some control flow with predicate register defines and uses. This can confuse the traditional analysis of code as a series of basic blocks and control flow "edges".

Mahlke et al. [31] defined the hyperblock as a single-entry, multiple-exit structure to help support effective predicated compilation. This structure replaces the use of basic blocks in the portions of a program's control flow graph that have been predicated. These hyperblocks are formed via selective if-conversion [7, 34] – a technique that replaces branches with predicate

define instructions. Once control reaches the beginning of a hyperblock, all instructions in that block will be fetched and executed, though only those whose guarding predicates are true will be committed. This entails a semantic change in what a compiler perceives as the data relationships in a given "block" of code. While in a traditional basic block, any two statements using the same variable have some data flow relationship (def-use, use-def, or def-def) this is not so for hyperblocks. Two statements both defining a variable `x` in a hyperblock may not have any data flow relationship if their guarding predicates are never on the same execution path.

II.C Background on Branch Prediction

In Section II.A, we introduced the problem of branch resolution delay in the face of modern pipelined processors. We also briefly discussed the option of guessing or *predicting* the outcome of a branch as it was fetched in order to be able to continue to fill the pipeline with possibly useful work while the branch was being resolved. Additionally, we motivated the importance of correct branch prediction and outlined predicated execution as an option for branches that were not able to be predicted well.

In this Section we provide a general overview of recent work in branch prediction focusing on the general tactics employed. The details of these methods will be important when we consider the interaction of branches (and their prediction) and predicated execution.

II.C.1 Static Branch Prediction

The simplest form of branch prediction is static branch prediction where a common prediction is made ("not taken", for example) for all branches. A minor modification to this which performs well is the "backward-taken, forward-not taken" static prediction scheme. This scheme relies on the fact that loop controlling branches frequently branch to a relative PC address less than the current PC (i.e., backwards in the sequential code layout). Loop branches are "taken" the number of times that the loop iterates - usually a rather large number. It is only on the last execution of the loop branch that this method will mispredict. When we are done executing the loop we wish this branch to fall through. The "forward-not taken" methodology is less compelling, but relies on the human tendency to put the more common of cases following an if-statement first (as the then clause).

This static prediction scheme, while implemented in hardware, relies on assumptions

about how the compiler (software) lays out code. Compiler-directed static branch prediction allows the compiler to directly specify to the hardware the prediction that should be made for a particular static branch. This scheme requires ISA support for the compiler to pass its predictions to the hardware and support in the hardware for reading the prediction from the instruction. The advantage is that it provides direct support from the compiler for branch predictions, rather than relying on hardware assumptions about the compiler. Additionally, it is possible to improve the branch behavior of programs with a simple recompile of the program rather than an upgrade of the hardware.

II.C.2 Dynamic Branch Prediction

While static branch prediction schemes are better than no branch prediction at all, the fact remains that a given static branch does not always exhibit the same behavior every time it is dynamically executed in a program or over the complete execution of a program.

The basic tenet that all dynamic branch prediction schemes attempt to take advantage of is that of likely repeated behavior. To capture this effect, two-bit saturating counters are traditionally used. The state diagram for a two-bit saturating branch prediction counter is shown in Figure II.7. If one could assign a counter to every static branch, then as long as branch behavior occurs in a repeated string with interruptions of no more than one different prediction, the branch will always be correctly predicted.

However, in practice it is difficult to build hardware for one 2-bit counter per static branch. Instead a fixed size table of counters is built which is indexed by a subrange of the branch address (usually the lowest `n` bits, where `n` is the number of bits needed to index the number of counters we have). This implementation attempts to provide the "per-static-branch" counter that we desire, with the caveat that branches whose `n` lowest bits are the same will end up predicting from and updating the same counter. This interference (usually called "aliasing") can have several flavors: neutral, constructive, or destructive.

Utilizing History Patterns

A further development in branch prediction recognizes that more accuracy can be gained by making a prediction that is based not just on the previous behavior of this branch, but based more specifically on the previous behavior of this branch *when it recently performed a certain way*. What this tries to capture is even more particular "patterns" or "histories" of branch behavior in predicting the likely behavior of the branch on the current execution. Fig-

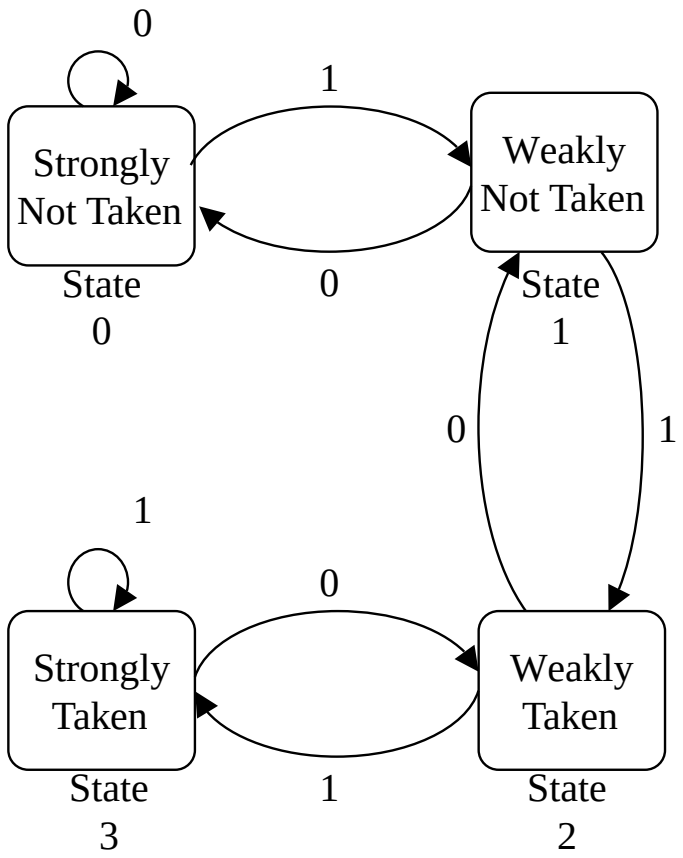


Figure II.7: A 2-bit saturating counter is frequently used to predict branch behavior. Branches which have recently been taken, will be predicted taken. Branches which have recently been not taken, will be predicted not taken.

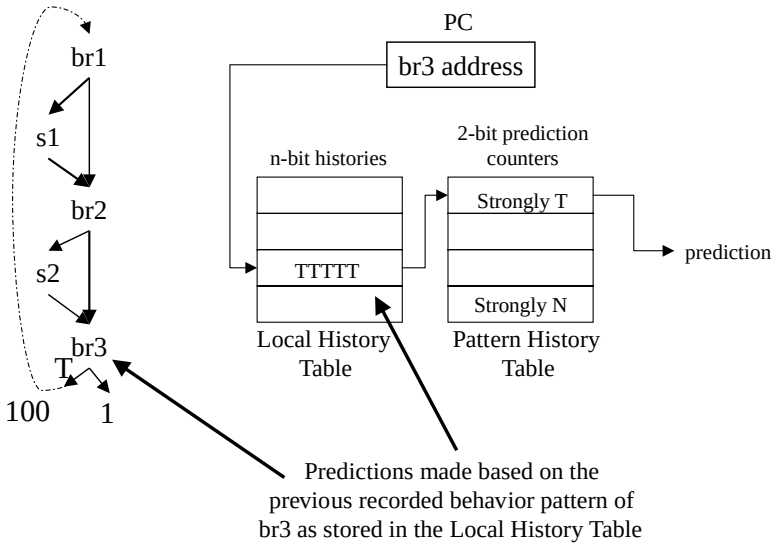


Figure II.8: A local history pattern branch predictor. The branch program counter (PC) is used to index into a table of per-branch histories. These histories are stored as n-bits of information about the last n occurrences of this branch (where 0 represents a not taken branch and 1 represents a taken branch). This pattern is then used to index into the local pattern history table where a 2-bit counter produces a prediction for that pattern. Here we see that `br3`, a loop controlling branch, has recently always been taken. This loop iterates 100 times before it exits, when the branch is not taken once. Since all branches with matching patterns (like all taken) index into the same entry in the pattern history table, different branches that share the same history pattern will share the same prediction counter.

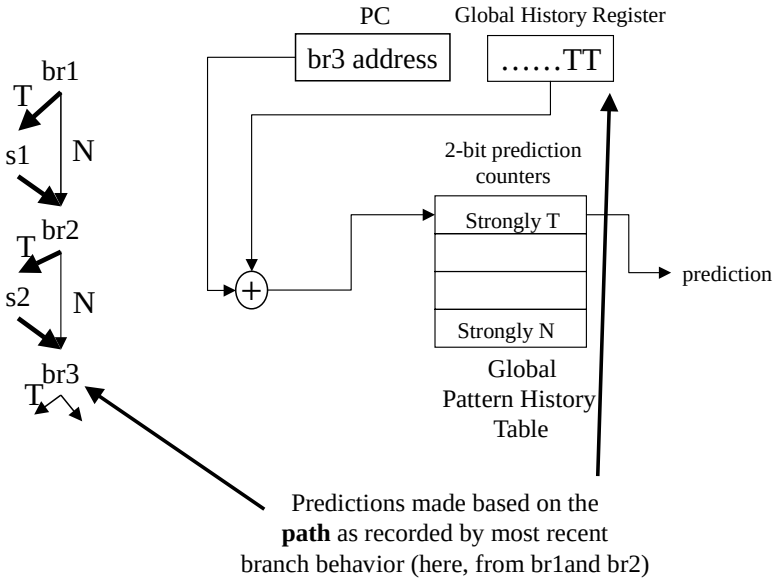


Figure II.9: A global history branch predictor. In the code example shown here, `br3` is only taken when both `br1` and `br2` are taken (the dark arrowed path). The global history predictor captures the behavior of recently executed branches in a global history register. Here we show the last two bits stored in the register for the highlighted path of execution, `TT`. The global history register is combined with the branch address to index into a table of 2-bit saturating counter predictors. Any other global history for this branch (ending in `NN`, `NT`, or `TN`) should predict not taken.

ure II.8 shows a schematic of a predictor that maintains a set of per-branch histories, recording the most recent `n` occurrences of this branch. In this example, we show the entries for `br3` a loop controlling branch that is taken 100 times before the loop terminates (and the branch is not taken). The local history recorded for `br3` shows that it has recently been taken. That history (all taken) is used to index into the pattern history table which uses a 2-bit saturating counter to make a prediction for a history of "all taken". Any branch which has this same history pattern, will use the same 2-bit saturating counter for its prediction.

Another method that tries to harness the power of branch history patterns is the global history branch prediction scheme. The idea behind a global history scheme is that a given branch may have a predictable behavior based on the behavior of the most recent `n` branches that occurred just before it. In Figure II.9 we show a control flow graph for a code where `br3` will be taken only when both `s1` and `s2` are executed. A global history prediction scheme can correctly capture that behavior by keeping a single register of history bits from the `n` most recently executed branches. Here we show the last part of the global history register when the the path of execution follows the darker arrows (i.e. `br1` and `br2` are taken). The

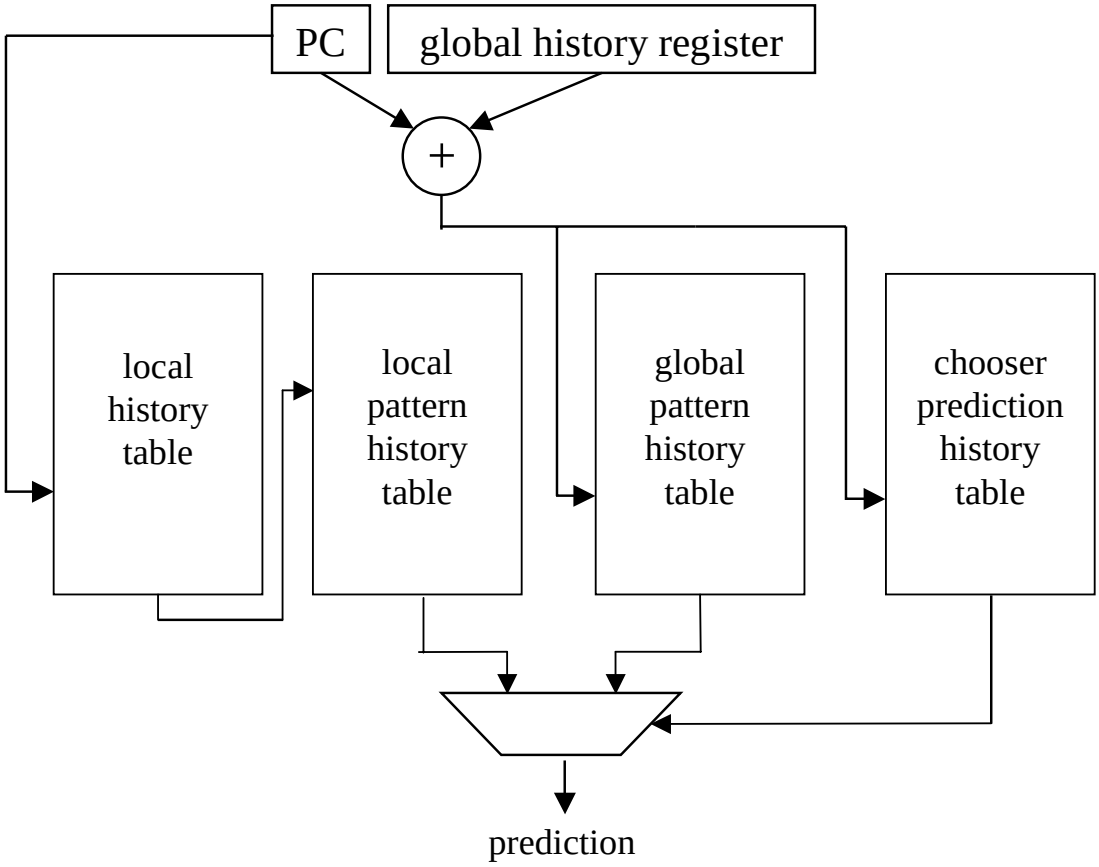


Figure II.10: Our baseline Meta Chooser branch prediction architecture. It combines a local history based prediction scheme with a global history prediction scheme. Each dynamic branch produces a prediction selected from either the local history or global history predictions as determined by the 2-bit saturating counter in the chooser table entry.

branch address is combined in some fashion with the global history register (usually using exclusive-or or concatenation) and the result is used to index into a table of two-bit saturating counters, producing a prediction for the branch. Note that for any other combination of global history register (ending in either `NN`, `NT`, or `TN`) and `br3`'s address, a prediction of not taken is desired. The global history register tries to identify and isolate path patterns which determine predictability for a given branch.

Meta Chooser Predictors: The Best of Both Worlds

Some of the best branch predictors currently found in processors attempt to take the best of both types of dynamic branch prediction methods. In real codes, some branches (such as loop branches) are more predictable based on the history of that branch (local history).

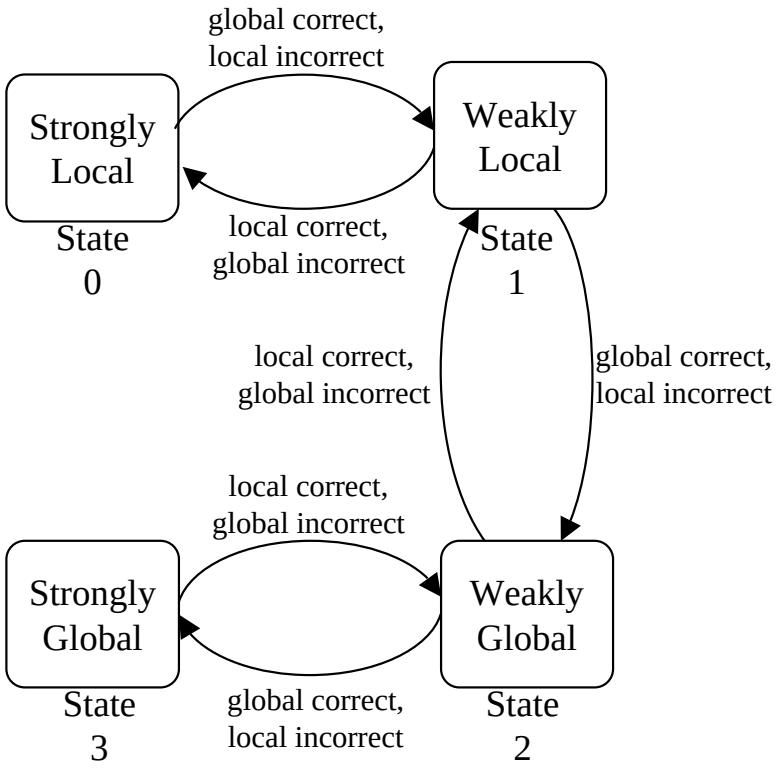


Figure II.11: A 2-bit saturating counter is used to select between a global-scheme branch prediction and a local-scheme branch prediction. For each dynamic branch (as identified by the PC and global history), if recent occurrences were better predicted with local rather than global history, then the local prediction will be selected. Conversely, if recent occurrences were better predicted with a global scheme, then the global prediction will be selected over the local prediction.

Some branches may be more predictable based on the history of recent branches (based on code structure or data values) and fare better with a global history predictor. Many modern branch predictors such as the DEC Alpha 21264 [27] maintain both a table of counters indexed with just the address of a branch and a table indexed with the global history register/address combination. For a given branch, both "local-style" and "global-style" predictions are made, then a separate predictor chooses whether to ultimately return the local or the global prediction. In essence, meta-data is stored in a chooser table which keeps track of and selects for the better performing predictor. Figure II.10 shows an implementation of a Meta Chooser style predictor. The chooser table is a table of 2-bit saturating counters which choose between the local and global predictions. These counters change their predictive state (local or global) to reflect the most recent correct predictor. The state diagram that controls the chooser 2-bit saturating counters is shown in Figure II.11.

Meta Choosers have benefits in other forms of flexibility as well. Frequently branch prediction is on the critical path in a processor. The local history based branch prediction scheme described here requires two serial table lookups to produce a prediction. In contrast, the global history scheme requires only one table lookup. A Meta Chooser scheme can help ease cycle-time restrictions by providing a multistage branch prediction. When a branch is fetched, a single table lookup can produce a global history style prediction which can be used to begin fetch of the next instruction. In the next cycle, when a prediction from the two table lookup local predictor finishes, it can, if indicated by the chooser, override the original prediction and begin fetching down the opposite path. This incurs an occasional cost of one cycle of mis-fetched code, but greatly eases the timing requirements of branch prediction implementation.

Chapter III

Related Work

In this chapter we address related work. The discussion is broken down into two areas: work related to compiler support for predicated execution and work investigating the effects of the interaction of predicated execution and branch prediction.

III.A Compiler Support for Predicated Execution

The challenges of doing data-flow and control-flow analysis on hyperblocks have been addressed. Since hyperblocks include multiple paths of control in one block, traditional compiler techniques are often too conservative or inefficient when applied to them. Methods of predicate-sensitive analysis have been devised to make traditional optimization techniques more effective for predicated code. Schlansker et al. [22, 25] use the predicate query system to define a predicate partition graph for a predicated region. This partition graph can represent disjointness between predicate values which is helpful in obtaining accurate liveness analysis to perform valid and optimized register allocation in predicated regions. However, this analysis does not extend across any join blocks (where multiple control flow paths merge) in a predicated region. Previous work has shown that path-sensitive analysis has been useful in the traditional data-flow domain [8, 14, 23]. Our work will extend path-sensitive analysis for predicated code and provide full relationship analysis including ancestor/successor and post-join relationships for predicate definitions.

August et al. [13] have subsequently proposed the program decision logic network as method of abstracting control flow. This network can then be used to produce equivalent predicated code using minimal numbers of predicate defining statements to improve instruction

level parallelism.

The success of predicated execution can depend greatly on the region of the code selected to be included in the region. August et al. [11] relates the pitfalls and potentials of hyperblock formation heuristics that can be used to guide the inclusion or exclusion of paths in a hyperblock. Their approach is to aggressively predicate early in compilation process and to perform partial reverse if-conversion during code scheduling in order to balance control flow and predication subject to architectural characteristics. They develop an equation which controls the inclusion or exclusion of a particular basic block into a region. This formula is based on excluding the following:

- Blocks which contain hazards to scheduling and/or optimization like procedure calls and memory accesses with no alias information.
- Blocks that are infrequently executed in comparison to the main "path" through the region.
- Blocks which contain disproportionately more instructions than other blocks also considered for incorporation into the region or which will disproportionately lengthen the longest data dependence chain through the region.

Additionally, factors such as the issue rate and available architectural resources and pipeline depth and cost of branch misprediction positively affect the inclusion of blocks into a predicated region. In our PSSA work we utilize the hyperblock developed by August el. al. [11] as implemented in the Trimaran system [3]. We will improve hyperblock scheduling via optimizations that are enabled by the more complete hyperblock information that we gather using PSSA. These techniques change the potential data dependence path considerations that should be used in hyperblock formation.

Moon and Ebcioglu [33] have implemented selective scheduling algorithms, which can schedule operations at their earliest possible cycle for non-predicated code. Our work in Predicated Static Single Assignment extends theirs for hyperblock-style predicated code, by allowing earliest possible cycle scheduling using predicated renaming with full-path predicates.

III.B The Interaction of Predicated Execution and Branch Prediction

Predicated execution is commonly proposed as a replacement solution for the problem of hard-to-predict branches. Mahlke et al. [30] investigated the interaction of predicated hyperblock region formation and branch prediction using two branch prediction architectures: a branch target buffer (BTB) with a 2-bit counter, and a BTB with profile-based direction prediction. Over a subset of SPEC92 benchmarks and UNIX utilities, they show a reduction in branch mispredict rate of 56% using hyperblock regions. However, since the replacement of branches with predicate defines comes at a cost of increased execution, not all branches will be replaced. This results in code that contains predicate defining statements and branches. Additionally, the branches left in the code can be found both "inside" predicated regions (eg, tagged with a guarding predicate themselves that provides an additional control on branch execution) and outside of predicated regions in portions of the program's control flow representation that were not targeted for predication. There are two ways in which these branches can be impacted by the process of predication. First is the impact that the removal of the branches transformed into predicate defines can have on other (possibly non-region) branches due to the global branch history they provide. Second is the impact of the increased fetch and execution (including branch prediction) of region branches.

Preliminary work by Klauser et al. [28] looked at a very restricted form of predication where only short, forward branches of the type that form `if-then` and `if-then-else` constructs can be transformed into predicate defines. They briefly examined the impact of allowing these new predicate defines to update the global history register even though they no longer required prediction via the branch prediction hardware. They found no conclusive evidence of benefit from global history update – but their scope of predication may have restricted the benefit to be gained. Our work will focus on exploring the options for global history update in the face of full, aggressive predication.

Acknowledging the problem of prediction of region branches in [12], August et al. presents a modified branch prediction architecture called PEP that incorporates predicate information into a local per-branch prediction scheme. They elaborated one of the problems of region formation by showing how the transformation of an unbiased branch into a predicate define could cause a previously predictable branch to become unpredictable. Their technique focuses on exploiting the relationship between a given predicate define and a branch guarded

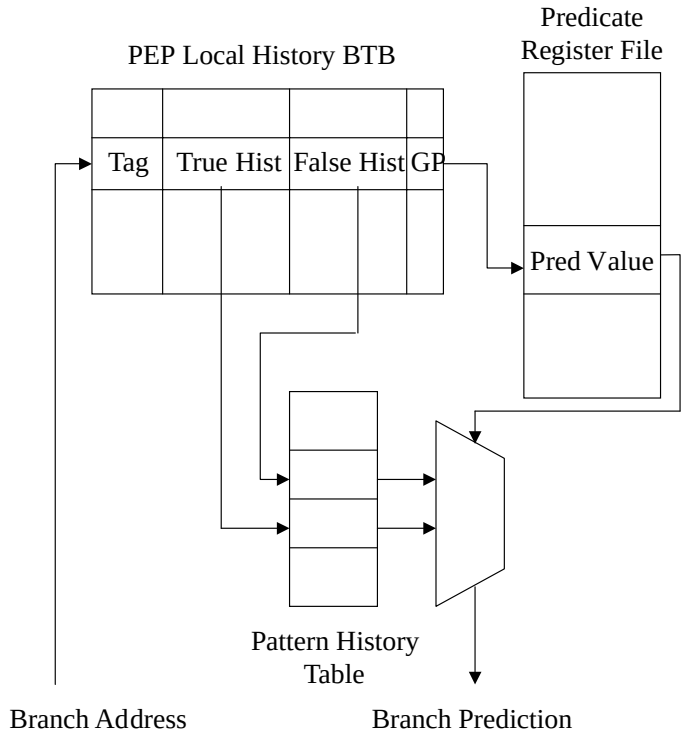


Figure III.1: High level design of the Predicate Enhanced Prediction scheme proposed by August et al. [12].

by that predicate to recover the original prediction pattern for that branch. This is accomplished by storing the guarding predicate of the branch instruction in the BTB. Additionally, two local histories are stored in the BTB entry – one associated with the branch behavior when the guarding predicate is true, and one when it is false. The "true history" should be used and updated with the same pattern that the original branch was accessed – since it will only be used and updated with the branch's guarding predicate is true. The "false history" should only be used and updated when the branch's guarding predicate is false – hopefully producing a prediction of "not taken". However, in cases where the predicate define guarding a branch has been issued to the pipeline, but not yet resolved, one may access the "false history" even when the branch's guarding predicate eventually resolves to true.

Figure III.1 shows a schematic of the PEP branch prediction architecture. A branch prediction takes two serial table lookups. The first lookup accesses the BTB providing the guarding predicate associated with the branch. Then another lookup is made in the predicate register file to find the currently available predicate value. The predicate value is then used

to choose between the predictions found by indexing a 2-bit saturating counter pattern history table using the histories from the BTB. We assume that the local histories are speculatively updated at fetch and correctly recovered on a branch misprediction.

Mahlke et al. [32] examined a new use of predicate registers for collecting information to assist in branch prediction via compiler synthesized information. They proposed new compiler techniques for statically examining register values to produce a dynamically executed function that would help guide branch predictions. They use predicate registers to hold the result of this dynamically executed function, then this predicate value is used in a modified version of the prepare-to-branch instruction that precedes a branch in their architecture.

Tyson [43] utilizes predicated execution to optimize short forward branches, showing that these constitute a significant percent of both integer and floating point branches and have relatively poor prediction rates. He shows up to a 30% reduction in misprediction rate for the SPEC92 benchmark suite – noting that most of the reduction comes directly from the branches translated into predicate defines that no longer require prediction. Tyson presents results for a region formation method that does not contain any changes in control flow. In addition, he examines an idealized region formation that allows any type of branch to remain in the predicated region, but states that it is unclear how to predict them. We assume for the results in [43], that branches left in these idealized regions are only being predicted when their guarding predicate is true.

Chapter IV

Using Predicate Information in the Compiler for Improved Code Scheduling

Compilers are large software systems traditionally organized into multiple passes, where the input is source code and the output is assembly or machine code. The source code is read in, parsed into an internal representation, analyzed, optimized, re-analyzed, additionally optimized, and so forth. Perhaps surprisingly, there is an almost universal basic internal representation used by the analysis and optimization passes of a compiler. This is a *control flow graph* of basic blocks with control edges between the basic blocks. Part of the challenge of supporting predicated execution in the compiler is that it re-defines part of this basic internal representation. In a traditional code representation, all control flow is expressed as relationships between basic blocks. Once control enters a basic block, all instructions in the basic block are executed in linear order. However, predicated execution defines control flow via predicate define statements and expresses the conditional execution of statements individually, based on a guarding predicate tagged on each instruction. Because of this basic infrastructure difference, analysis algorithms must be modified to query for control flow relationships that might exist in a single predicated hyperblock as well as between basic blocks. To provide full flexibility in analysis and optimization algorithms, full predicate relationship information needs to be defined as accurately as that which was available in the original control flow format.

Here we provide a *full-path predicate* relationship solution in the form of *Predicated Static Single Assignment* (PSSA). We show the potential of PSSA for two hyperblock scheduling optimizations: Predicated Speculation and Control Height Reduction.

IV.A Motivation

To produce the fastest predicated code via optimization opportunities and scheduling methods, compilers need access to complete information regarding predicate relationships in regions of predicated code. Previous techniques have offered only partial solutions. Our goal is to enable extremely aggressive scheduling of predicated code by exposing *full path predicate* relationships in the compiler. We will show how to use these relationships to develop optimizations that allow all statements to be scheduled at their *true data dependence length* - optimizing away control dependences and false data dependences. This will show the shortest possible schedule that can be obtained in terms of scheduled dynamic cycles, after which, we show benefit gained using restricted resources.

We define Predicated Static Single Assignment (PSSA) - a compiler transformation which makes explicit the full relationships between predicates in a region of predicated code. It goes beyond simple predicate disjointness knowledge to encode full-path relationships including: predecessor/successor relationships, post-join relationships, and branch-out-of-the region control relationships, which we define via an example in this section. As these predicate dependences encode the control flow dependences that were removed via predication, they will be important to correct and efficient compiler scheduling of the code.

A major task for the scheduler of a multi-issue machine is to find independent instructions. Unfortunately, predication introduces additional dependences that traditional code doesn't have to consider. In Figure II.2(b), there is a dependence between the definition of the guarding predicate `P2` and its use in the statement `b=q if P2`. Since predication combines multiple basic blocks, it may introduce false dependences between disjoint paths. For example, in Figure II.2(b), in the absence of predicate dependence information, we would infer a dependence between the definition of `b` in `b=q if P2` and the use of `b` in `d=b+3 if P3`. However, these two statements are guarded by *disjoint* predicates. Therefore, only one of the predicates (`P2` or `P3`) can possibly be true; only one of the statements will actually be committed and no dependence does in fact exist. Similarly for Figure IV.1(b), if we did not know that predicates `G` and `H` are disjoint, then we would falsely assume an output dependence between the statements

`r=5+x` and `r=x+8`.

Johnson et al. [25] devised a scheme to determine the disjointness of predicates using the predicate partition graph. This analysis allowed more effective register allocation as live ranges across predicated code could be more accurately determined [22]. Their approach was limited to describing disjointness with restricted path information. Path information that extended across join points was not collected. In Figure IV.1, the predicate partition graph would determine that the following pairs of predicates are disjoint: `G` and `H`, `B` and `C`, `D` and `!D`. However, no information regarding the relationship between `D` and `G` or `D` and `H` would be available. This is because of the presence of a *join block* where more than one path through the region share a common set of instructions (in this case `t=rand()` and `t>r`).

This "cross-join" information is needed to provide the scheduler full flexibility in scheduling statements such as `y=t+r`. If path information is not available, then `y=t+r` is guarded on true and the scheduler correctly assumes this statement is dependent on `t=rand()`, `t=t-s`, `r=5+x`, *and* `r=x+8`. However, since there are two possible definitions of each operand (`t` and `r`), there are 4 combinations of operands that could in fact cause the definition of `y` - each executable via one (or more) paths of execution through the region. If 4 versions of this statement could be made (one for each combination of the operands), then each could be scheduled at the minimum dependence length for that version. This differs from full-path enumeration in that though there are 6 different paths that can reach `y=t+r`, we recognize the need to only enumerate 4 versions of the statement, in accordance with the number of possible source operand combinations.

While disjointness information can maintain information regarding paths since the most recent join, we will need to combine path information across joins to remove unnecessarily conservative scheduling dependences. Figure IV.1(b) shows the cross-join path information that would be needed to guard each assignment of `y` so that the scheduler can know the precise dependences for each copy. This will allow the most flexibility in scheduling each statement.

Specific subsets of full-path predicate knowledge will be required for our predicate region scheduling optimization techniques. Specifically, we will take advantage of the predecessor/successor predicate relationships we collect to speculatively schedule instructions. If we wanted to speculatively schedule `t=t-s` we need to know that `D` is a successor of `C` whose `t>7` controls the execution of `t=t-s`. Additionally, branch-out-of-the-region relationships will also need to be explicitly specified for this same reason. If we attempt to speculatively schedule `t=t-s` in or above the second join region, we'll need to know that this definition should not

occur when `w>2` is taken out of the region.

Although precise dependence information can be determined from guarding predicate relations, we will also show that renaming techniques can be of additional use to achieve greater scheduling flexibility. By renaming variables that have more than one definition in a region, we will maintain path information even after optimizations which change the guarding predicate of a statement have been applied. For example, by a combination of renaming and utilization of precise guarding dependence relationships, we will be able to execute both `r=5+x` and `r=x+8` in the first cycle of the predicated hyperblock proposed in Figure IV.1(b). The renaming will allow the two statements to be scheduled in parallel, and full-path guarding relationships will allow us to identify and rename the later uses of each definition of `r` to maintain correctness.

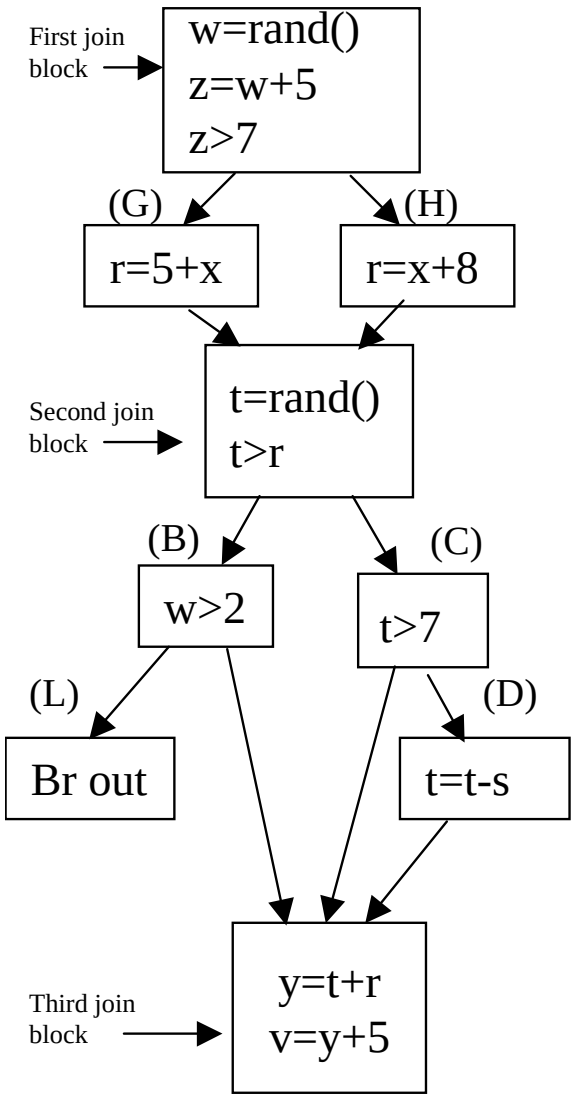
IV.B Predicated Static Single Assignment

Techniques such as renaming [6] and Static Single Assignment (SSA) [20, 19] have proved useful in eliminating false dependences in traditional code [44]. Removing false dependences allows more flexibility in scheduling since data independent operations can move past each other during instruction scheduling. This is especially important in predicated code since instructions from multiple basic blocks have been merged into a larger region to provide additional parallelism in scheduling.

In non-predicated code, SSA assigns each target of an assignment operation a unique variable. At join nodes a ϕ-function may need to be inserted if multiple definitions of a variable reach the join. The ϕ-functions determine which version of the variable to use and assign it to an additional renamed version. This new variable is used to represent the merging of the different variable names. Figure IV.2 shows a simple example of SSA form. In the assignment `b3->`ϕ`(b1,b2)`, the variable `b3` represents the reaching definition of `b` which is to be used after the join of definition `b1` or `b2`.

As discussed in section IV.A, eliminating false dependences is equally important and a more complex task for predicated code, since multiple control paths are merged. To address this problem we developed a predicate-sensitive implementation of SSA called *Predicated Static Single Assignment* (PSSA).

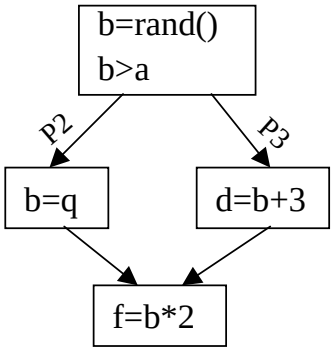
PSSA seeks to accomplish the same objectives as SSA for a predicated hyperblock. First, it must assign each target of an assignment operation in the hyperblock a unique variable. Second, at points in the hyperblock where multiple paths come together it must summarize



w=rand()		if true
z=w+5		if true
G,H	z>7	if true
r=5+x		if G
r=x+8		if H
t=rand()		if true
B,C	t>r	if G
B,C	t>r	if H
L,	w>2	if B
__,D	t>7	if C
t=t-s		if D
br out		if L
y=t+r		if !D&G
y=t+r		if !D&H
y=t+r		if D&G
y=t+r		if D&H
v=y+5		if true

(a) Original Control Flow Graph (b) Predicated Hyperblock with Paths

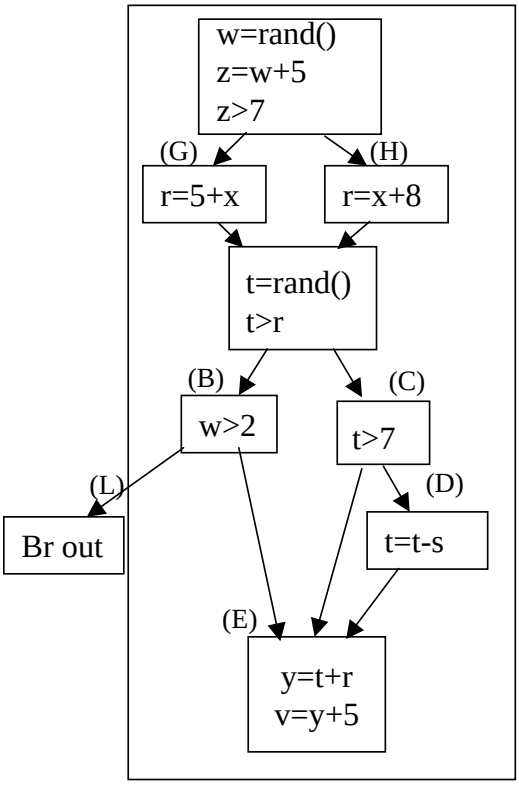
Figure IV.1: Code is duplicated when more than one definition reaches a use to maintain maximum flexibility for the scheduler. In (b), the statement y=t+r is duplicated for each pair of definitions that may reach this statement. Each copy is guarded by the predicates that defined the path along which those definitions would occur.




```
b1=rand()
if b1>a
   b2=q
else
   d1=b1+3
b3->Ø(b1,b2)
f1=b3*2
```

(a) Control Flow Graph (b) Code in SSA form

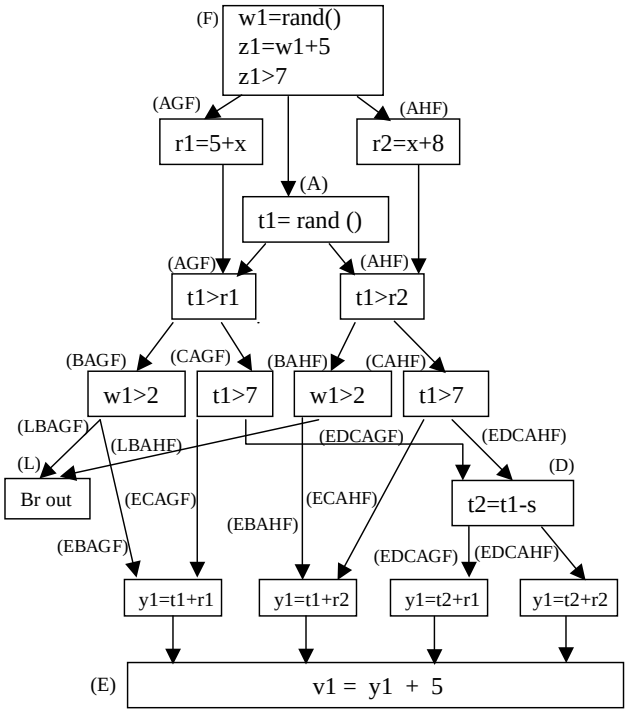
Figure IV.2: Static Single Assignment




```
w=rand()      if true
z=w+5         if true
G,H   z>7     if true
r=5+x         if G
r=x+8         if H
t=rand()      if true
B,C   t>r     if true
L,__  w>2     if B
__,D  t>7     if C
t=t-s         if D
br out        if L
y=t+r         if true
v=y+5         if true
```

(a) Original Control Flow Graph (b) Predicated Hyperblock

Figure IV.3: Extended example of transformation from non-predicated CFG to predicated hyperblock. This is a repetition of the motivating example in Figure IV.1



(a) PSSA dependence graph

F=true		if true	1
w1=rand()		if F	1
z1=w1+5		if F	2
AGF, AHF	cmpp.un.uc z1>7	if F	3
r1=5+x		if AGF	3
r2=x+8		if AHF	3
A=OR(AGF,AHF)		if true	4
t1=rand()		if A	4
BAGF,CAGF	cmpp.un.uc t1>r1	if AGF	5
BAHF,CAHF	cmpp.un.uc t1>r2	if AHF	5
LBAGF,EBAGF	cmpp.un.uc w1>2	if BAGF	6
LBAHF,EBAHF	cmpp.un.uc w1>2	if BAHF	6
ECAGF, EDCAGF	cmpp.un.uc t1>7	if CAGF	6
ECAHF, EDCAHF	cmpp.un.uc t1>7	if CAHF	6
D=OR(EDCAGF,EDCAHF)		if true	7
t2=t1-s		if D	7
L=OR(LBAHF,LBAGF)		if true	7
br out		if L	7
y1=t1+r1		if EBAGF	6
y1=t1+r2		if EBAHF	6
y1=t1+r1		if ECAGF	6
y1=t1+r2		if ECAHF	6
y1=t2+r1		if EDCAGF	8
y1=t2+r2		if EDCAHF	8
E=OR(EBAGF,EBAHF,ECAGF, ECAHF,EDCAGF,EDCAHF)		if true	7
v1=y1+5		if E	9

(b) PSSA-transformed code

Figure IV.4: The PSSA dependence graph shows the flow of data and control through the PSSA-transformed code. Blocks labeled with *full-path* predicates (indicated by multiple letters) contain statements that are only executed along that path. Blocks labeled with *block* predicates (single letters) contain statements that will be executed along several paths. The numbers on the right-hand side of the PSSA-transformed code shows the cycle that the instruction is available for scheduling.

under what conditions each of the multiple definitions of a variable reaches that join. The second objective is accomplished through the creation of full-path predicates and path-sensitive analysis.

Consider the sample predicated code shown in Figure IV.3 using traditional hyperblock predication [31]. In this predicated example, all branches have been replaced (except the one leaving the hyperblock) with predicate-defining operations using If-conversion. The predicates that are defined in this example correspond to the two edges exiting each conditional branch in the CFG in Figure IV.3. Figure IV.4 shows this example after PSSA has been applied and displays a graph showing the post-PSSA dependence relationships.

The PSSA transformation has 2 phases: pre- and post-optimization. Hyperblocks are converted to PSSA form before optimization. After optimization, PSSA inserts clean-up code on edges leaving the hyperblock, copying renamed variables back to their original names and then removing any unused predicate definitions.

IV.B.1 Converting to PSSA Form

When converting to PSSA form, each operation is processed in turn beginning at the top of the hyperblock and proceeding to the end. *Control PSSA* is applied to predicate-defining operations, and *Normal PSSA* is applied to all other operations. Figure IV.5 shows our PSSA algorithm.

We first describe Normal PSSA. If the operation is an assignment, the variable defined is renamed. The third operation in Figure IV.4(b), `z1=w1+5`, is an example. All operands are adjusted to reflect previously renamed variables (e.g. `w` becomes `w1`). If the operation is part of a join block, multiple versions of the operands may be live. The first operation (`y=t+r`) in the third join block of Figure IV.3 provides an example. Here, the operation will be duplicated for each path leading to the join and the correct operand versions for each path will be used in the duplicate statement as seen in Figure IV.4 (in the multiple definitions of `y1`). The duplicates are guarded by the full-path predicate (described below) associated with the path along which the operands are defined. Though there are 6 definitions of `y1` (only 4 are unique), there is only one definition of `y1` on any given path. These definitions are predicated on disjoint predicates; only one of them can possibly be true, and only one of them will be committed.

We next describe Control PSSA. The single `cmpp` operation that defined one or two block predicates (such as the definitions of `B` and `C` in Figure IV.3) is replaced by one or more `cmpp` operations, each associated with a particular path leading to that block. As can be seen

in Figure IV.4(b) there are now two `cmpp` operations: one defining `BAGF` and `CAGF`, and one defining `BAHF` and `CAHF`. These new predicates are called *full-path predicates* (FPPs). Each FPP definition has the appropriate operand versions for its path and each is guarded by the FPP that defined the path prior to reaching the new block. For example, the `cmpp` defining `BAGF` and `CAGF` is predicated on `AGF`. A FPP specifies the *unique* path along which an operation is valid for execution, enabling PSSA to provide correct guarding predicates for the duplicate statements previously described.

In the example in Figure IV.1 we pointed out that the definitions of `y1` needed guarding predicates that captured information about paths of execution. This same control flow graph is also shown in Figure IV.3. The first definition of `y1` needed to be guarded by a predicate representing a path of execution through block `G` but not block `D`. In addition, the predicate needs to reflect that the execution actually reached the block of the statement in question (`E` in this case). Register `y1` would be incorrectly modified if, for example, the branch out of the hyperblock is taken and block `E` is never reached. The new FPP `EBAGF` represents the precise conditions for correct execution.

In addition to the `cmpp` statements added to define FPPs, `cmpp` statements are included to rename join blocks whose statements were originally predicated on true. `A` and `E` and their associated FPPs are examples. The operations in Figure IV.3(b) predicated on true, are predicated on `F`, `A` and `E` in the PSSA version of the code shown in Figure IV.4. This is necessary to maintain exact path information.

Clearly, this has the potential to cause an exponential amount of code duplication. It might seem more reasonable to follow the example of SSA and insert ϕ-functions at join points to resolve multiple definitions. For example, an implementation of ϕ-functions resolving `r` and `t` in the definition of `y1` could be:

(1) `r=r1 if G`

(2) `r=r2 if H`

(3) `t=t1 if true`

(4) `t=t2 if D`

(5) `y1=r+t if true`

While this would have the advantage of decreasing duplication, it does not eliminate the need for predicate-sensitive analysis. Predicate relationship information is still needed to

determine the reaching definitions and associated predicates, and to determine the order of the copy operations. For example, both of the statements (3) and (4) defining `t` in the previous sequence could be committed. The literal predicate `true` is always true, and predicate `D` could be true as well. For the use of `t` in (5) to get the correct definition, statement (4) cannot be executed before statement (3). Moreover, other side effects that degrade performance are introduced. Most important is that the insertion of ϕ-functions adds data dependences. For example, a true dependence is introduced between the definition of `t1` and its use in (3). In addition, false dependences are re-introduced. An example is the output dependence between the two definitions of `t`. Thus, SSA and the usual ϕ-function implementation does not give the desired scheduling flexibility.

Block predicates are also important to the PSSA transformation. PSSA uses predicate `OR` statements to redefine the block predicates as the union of the FPPs associated with the paths that reach the block. PSSA does not simply duplicate every path through the hyperblock. Duplication only occurs when necessary to remove false dependences. When there is only one version of all operands reaching a statement, only one version of the statement is required. This is the case with `v1=y1+5` in Figure IV.4. The variable `y1` is the only version live in node E. This statement is guarded by E, a block predicate created by taking the logical OR of EBAGF, EBAHF, ECAGF, ECAHF, EDCAGF, and EDCAHF. As long as control reaches node E, regardless of the path taken, we will execute and commit the statement `v1=y1+5`.

IV.B.2 Post-Optimization Clean-up

After optimization is applied to code in PSSA form, a clean-up phase is run to remove unnecessary code and to assure consistent code outside of the hyperblock.

The PSSA implementation described in this dissertation generates `cmpp` statements for every path and block. These are entered into the PSSA data structure that maintains information about the relationships between the predicates they define, which provides maximum flexibility during optimization. However, some of these FPP definitions may not be used, and the corresponding `cmpp` operations will be discarded, reducing the code size significantly.

Finally, to assure correct execution following the hyperblock, PSSA inserts copy operations assigning the original variable names to all renamed definitions that are live out of the hyperblock. These are placed on the appropriate exit of the hyperblock. For example, the exit branch guarded by `L` in Figure IV.3 would include `t = t1 if L` if `t` was live out of the hyperblock at this exit.

```
Loop through all predicated statements in hyperblock:
{
  if (GP=TRUE)
      get_new_GP
  else
      reset GP
  if data statement, apply Normal PSSA
  {
      rename destination
      if operands have > 1 version
      {
          copy stmt for each path with correct operand versions
          predicate(path)
      }
      else
      {
          find correct operand versions for whole path
          predicate(whole predicate)
      }
  }
  if cntl stmt, apply Control PSSA
  {
      define new path predicates
      create cntl statements with correct operand versions for each path predicate(path)
  }
}
```

Figure IV.5: PSSA Algorithm.

IV.C Hyperblock Scheduling Optimizations

In this section, we describe how PSSA enables Predicated Speculation (PSpec) and Control Height Reduction (CHR) for aggressive instruction scheduling. PSpec allows operations to be executed before their guarding predicates are determined and CHR allows the guarding predicates to be determined as soon as possible, reducing the number of operations that need to be speculated. Used together with PSSA, we demonstrate that we can schedule the code at its earliest schedulable cycle, assuming a machine with unlimited resources.

IV.C.1 Predicated Speculation

This section describes how to perform speculation on PSSA-transformed code. In general, speculation is used to relieve constraints which control dependences place on scheduling. One can speculatively execute operations from the likely-taken path of a highly-predictable branch, by scheduling those operations before their controlling branch [29]. Similarly, Predicated Speculation (PSpec) will schedule a normal operation above the `cmpp` it is dependent upon, optimizing a hyperblock's execution time.

PSpec handles placement of the speculated predicated operation in a uniform manner. PSpec schedules a normal operation at its earliest schedulable cycle. When speculating an operation, the operation is scheduled earlier than the operation it is control dependent on, and is predicated on true. We assume that any exceptions raised by the speculated operations will be taken care of using architecture features such as poison bits [9].

Instruction Scheduling with Speculation

To demonstrate the usefulness of PSSA in enabling PSpec, Figure IV.6 shows the code from Figure IV.4 after the PSpec optimization has been applied. The assignments to `r1` and `r2` are examples of speculated operations. Notice that based on data dependences, they could both be scheduled at cycle one, which would have been impossible without renaming.

During predicated speculation, each operation is considered sequentially, beginning with the first instruction in the hyperblock. If it is a normal, non-store operation, PSpec compares its earliest schedulable cycle based on its data dependences with the cycle in which its guarding predicate is currently defined. If the operation can be scheduled earlier than its guarding predicate, the operation is predicated on true and scheduled at its earliest schedulable cycle.

Recall that PSSA has not performed full renaming, so further renaming may be required by PSpec. An example is the definition of `y1` in Figure IV.4. If we speculate any of the definitions of `y1` by predicating them on true without renaming, incorrect code can result. Consequently, we must rename the operations being speculated. The results of applying this to the 6 definitions of `y1` (now `y1`, `y2`, `y3`, `y4`, `y5`,and `y6`) appear in Figure IV.6. Speculation and renaming may require the duplication of operations using the definition being speculated, since there may now be multiple reaching definitions. When speculating `y1`, the operation `v1=y1+5` had to be duplicated and guarded on the appropriate FPP as shown in Figure IV.6. This is possible because PSSA already created all the necessary FPPs and path information.

If the guarding predicate has been defined by the operation's earliest schedulable cycle, we do not apply PSpec. It is again scheduled at the cycle equal to its earliest schedulable cycle, but guarded by the guarding predicate assigned by PSSA. The instruction `z1=w1+5` displays this characteristic. The algorithm for PSpec instruction scheduling is shown in Figure IV.7.

Using PSpec, the hyperblock can now be scheduled in 6 cycles as compared to 9 cycles in Figure IV.4. Since PSpec is applied whenever the definition of the operation's guarding predicate occurs later than the earliest schedulable cycle of the operation, we could reduce

F=true		f true	1
w1=rand()		if F	1
z1=w1+5		if F	2
AGF, AHF	cmpp.un.uc z1>7	if F	3
r1=5+x		if true	1
r2=x+8		if true	1
A=OR(AGF, AHF)		if true	4
t1=rand()		if true	1
BAGF,CAGF	cmpp.un.uc t1>r1	if AGF	4
BAHF,CAHF	cmpp.un.uc t1>r2	if AHF	4
LBAGF,EB AGF	cmpp.un.uc w1>2	if BAGF	5
LBAHF,EB AHF	cmpp.un.uc w1>2	if BAHF	5
ECAGF, EDC AGF	cmpp.un.uc t1>7	if CAGF	5
ECAHF, EDC AHF	cmpp.un.uc t1>7	if CAHF	5
D=OR(EDCAGF,E DCAHF)		if true	6
t2=t1-s		if true	2
L=OR(LBAHF,LB AGF)		if true	6
br out		if LBAGF	5
br out		if LBAHF	5
y1=t1+r1		if true	2
y2=t1+r2		if true	2
y3=t1+r1		if true	2
y4=t1+r2		if true	2
y5=t2+r1		if true	3
y6=t2+r2		if true	3
E= EBAGF,EBAHF, ECAGF, ECAHF,EDC AGF,EDC AHF)		if true	6
v1=y1+5		if true	3
v2=y2+5		if true	3
v3=y3+5		if true	3
v4=y4+5		if true	3
v5=y5+5		if true	4
v6=y6+5		if true	4

Figure IV.6: Extended code example after PSpec optimization has been applied. Statements (other than first statement) predicated on true have been speculated. The numbers on the right-hand side of the code are the cycles that the instructions are available to be scheduled.

```
PSpec(normal_op)
{
  if (normal_op.guarding_predicate not defined by
      normal_op.earliest_schedulable_cycle)
  {
      if (multiple defs of normal_op.target exist
      {
          rename(normal_op.target);
          update_uses(normal_op.target);
      }
      normal_op.schedule(earliest_schedulable_cycle);
      normal_op.set_predicate(true);
  }
  else
  {
      normal_op.schedule(earliest_schedulable_cycle);
  }
}
```

Figure IV.7: Basic PSpec Algorithm.

the number of operations that need to be speculated by moving the definition of the guarding predicates earlier. The goal of the next optimization, Control Height Reduction, is to allow predicates to be defined as early as possible.

IV.C.2 Control Height Reduction

Control Height Reduction (CHR) eases control constraints between multiple control statements. CHR allows successive control operations on the control path to be scheduled in the same cycle, effectively reducing control dependence height. For example, in the code in Figure IV.6, the control comparisons for `z1>7` and `t1>r1` are scheduled in cycles 3 and 4, respectively. However, the second comparison is only waiting for the definition of its guarding predicate `AGF`.

To schedule it earlier, consider the PSSA dependence graph in Figure IV.4. The definition of `BAGF` (defined by the condition `t1>r1`), is control dependent on the definition of `AGF` (defined by the condition `z1>7`). We could also define `BAGF` directly as the logical AND of the conditions `z1>7` and `t1>r1` removing the dependence on the definition of `AGF`. This AND expression could also be scheduled in cycle 3.

Control Height Reduction was proposed in [38]. It was successfully used to reduce the height of control recurrences found in loops when applied to superblocks. A *superblock* is a selected trace of basic blocks through the control flow graph containing only one path of

F=true		if true	1
w1=rand()		if F	1
z1=w1+5		if F	2
AGF,A HF	*cmpp.un.uc z1>7*	*if F*	*3*
r1=5+x		if true	1
r2=x+8		if true	1
A=OR(AGF|AHF)		*if true*	*4*
t1=rand()		if true	1
BAGF, CAGF	*cmpp.an.an z1>7*	*if true*	*3*
BAGF, CAGF	*cmpp.an.ac t1>r1*	*if true*	*3*
BAHF, CAHF	*cmpp.ac.ac z1>7*	*if true*	*3*
BAHF, CAHF	*cmpp.an.ac t1>r2*	*if true*	*3*
LBAGF,EB AGF	cmpp.an.an z1>7	if true	3
LBAGF,EB AGF	cmpp.an.an t1>r1	if true	3
LBAGF,EB AGF	cmpp.an.ac w1>2	if true	3
LBAHF,EB AHF	cmpp.ac.ac z1>7	if true	3
LBAHF,EB AHF	cmpp.an.an t1>r2	if true	3
LBAHF,EB AHF	cmpp.an.ac w1>2	if true	3
ECAGF,EDC AGF	cmpp.an.an z1>7	if true	3
ECAGF,EDC AGF	cmpp.ac.ac t1>r1	if true	3
ECAGF, EDC AGF	cmpp.an.ac t1>7	if true	3
ECAHF,EDC AHF	cmpp.ac.ac z1>7	if true	3
ECAHF,EDC AHF	cmpp.ac.ac t1>r2	if true	3
ECAHF, EDC AHF	cmpp.an.ac t1>7	if true	3
D=OR(EDCAGF,EDCAHF)		*if true*	*4*
t2=t1-s		if true	2
L=OR(LBAHF,LBAGF)		*if true*	*4*
br out		if LBAGF	3
br out		if LBAHF	3
y1=t1+r1		if true	2
y2=t1+r2		if true	2
y3=t1+r1		if true	2
y4=t1+r2		if true	2
y5=t2+r1		if true	3
y6=t2+r2		if true	3
E=OR(EBAGF,E BAHF,ECA GF, ECAHF,E DCAGF,E DCAHF)		*if true*	*4*
v1=y1+5		if EBAGF	3
v1=y2+5		if EBAHF	3
v1=y3+5		if ECAGF	3
v1=y4+5		if ECAHF	3
v1=y5+5		if EDCAGF	4
v1=y6+5		if EDCAHF	4

Figure IV.8: Extended example after PSpec and CHR optimizations have been applied. `Cmpp` instructions displayed in italics define predicates that are not used after optimization. Therefore, the statements can be removed from the final code. The numbers on the right of the code represent the cycle that the instruction is available to be scheduled.

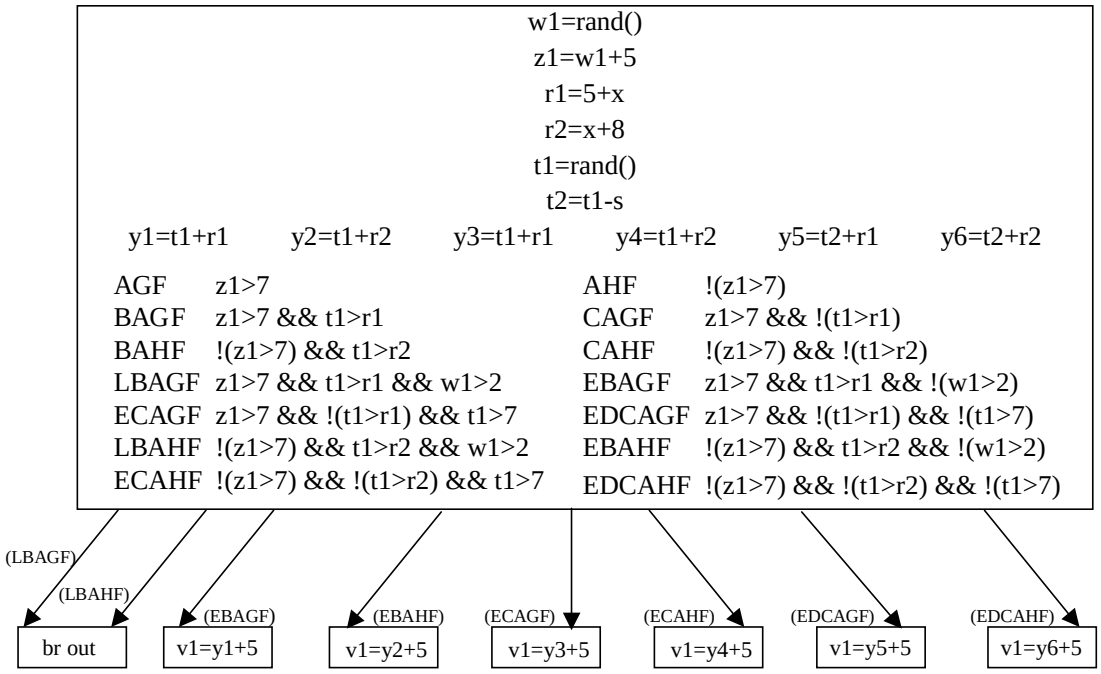


Figure IV.9: Dependence graph after PSpec and CHR have been applied.

control [37]. The path defining aspects of PSSA allow our algorithm to efficiently apply CHR to predicated hyperblocks, since the full-path predicates expose all of the original separate paths throughout the hyperblock.

Schlansker et al. [39] expanded on their previous research, applying speculation prior to attempting height reduction. Speculation can remove dependences between the branch conditions that need to be combined to accomplish the reduction. However, in that work, speculation was limited to operations that would not overwrite a live register or memory value if speculated, since they did not use renaming. In Figure IV.4, the `cmpp` operation defining `BAGF` and `CAGF` is shown scheduled at cycle 5 due to dependences on `t1` and `r1`. PSSA allows us to apply PSpec and schedule these definitions of `t1` and `r1` in cycle 1, making the `cmpp` available for CHR as shown in Figure IV.8.

Instruction Scheduling with PSpec and CHR

During instruction scheduling, PSpec is performed as described in Section IV.C.1. For each *control* operation (`cmpp`), CHR is performed if possible.

Recall that the operations in Figure IV.4 are scheduled in the order given in the PSSA hyperblock. Like PSpec, CHR compares when the operation could be scheduled based on its

```
CHR(cmpp_op)
{
    if (cmpp_op.guarding_pred defined
        by cmpp_op.earliest_schedulable_cycle)
    {
        cmpp_op.schedule(cmpp_op.earliest_schedulable_cycle)
    {
/* Apply Control Height Reduction */
    else
    {
        while (more_stmts_defining(cmpp_op.guarding_pred))
        {
            next_def=next_defining_stmt(cmpp_op.guarding_pred)
            copy=duplicate(next_def)
            copy.schedule(next_def.get_scheduling_time())
            copy.predicate_on(next_def.get_guarding_pred())
            copy.set_define(cmpp_op.get_pred_defined())
            copy.set_tag_to(a)
        }
        cmpp_op.schedule(next_def.get_scheduling_time())
        cmpp_op.predicate_on(next_def.get_guarding_pred())
        cmpp_op.set_tag_to(a)
    }
}
```

Figure IV.10: Basic Control Height Reduction Algorithm.

earliest schedulable cycle with when it must be scheduled if it waited for its guarding predicate to be defined. If it does not need to wait on the definition of its guarding predicate, it is simply scheduled at its earliest schedulable cycle. Without Pspec, the definition of `BAGF` was waiting on the definition of `t1` and `r1`. With Pspec, it is only waiting on the definition of its guarding predicate. Therefore, it is beneficial to control height reduce.

By ANDing the condition of the current definition with the condition that defined its guarding predicate, we can schedule this definition earlier. If the definition of the guarding predicate involved conditions that were ANDed as well, all of the conditions must be included, so the number of `cmpp` statements needed to define the current operation increases. The `.a` tag on each of these cmpp statements indicates that all of them are required for the final definition.

Consider the operations `z1>7`, `t1>r1` and `t1>7` in Figure IV.4. We control height reduce these operations in Figure IV.8, since they are all schedulable in cycle 3 based on our scheduling constraints. The new dependence graph resulting from PSpec and CHR is shown in Figure IV.9. The definition of ECAGF now describes the combination of `z1>7` being true AND `t1>r1` having a value of false AND `t1>7` having a value of true. We implement this logical AND, using the `.ac` and `.an` qualifiers. The definition of `ECAGF` requires that the conditions

`z1>7` and `t1>7` and the condition `!(t1>r1)` evaluate to true for the FPP to get a value of true. If any one of the requirements is not met, the FPP will be set to false. The compares can architecturally be performed in the same cycle [26] allowing multiple links in a control path to be defined simultaneously. The algorithm for CHR is found in Figure IV.10.

Using PSpec and CHR on PSSA-transformed code results in the 4 cycle schedule shown in Figure IV.8. Note that this last version of the code has fewer operations than the previous version in Figure IV.6 after the operations shown in italics are removed (because these operations define predicates that are never used). Using predicated speculation and control height reduction together on PSSA-transformed code allows every operation to be scheduled at its earliest schedulable cycle.

IV.D Results

We have implemented algorithms to perform PSSA, CHR and PSpec on hyperblocks in the Trimaran System (Version 2.00). We collect profile-based execution weights for operations in the codes and schedule operations with an assumed one-cycle latency in order to calculate execution time. Additionally, we conservatively assume that a load is dependent on all prior stores along a given path, and that a store is dependent on prior stores as well. We also ensure that all instructions along a path leading to a branch out of the hyperblock are executed prior to exiting the hyperblock.

Figure IV.11 shows normalized execution time when applying our optimizations for several Trimaran benchmarks: `fib`, `mm`, `wc`, `fir`, `wave`, `nbradar` (a media benchmark), `qsort`, `alvinn` (from SPECFP92), `compress` (from SPECINT95), and `li` (from SPECINT95). These codes are described in the Trimaran Benchmark Certification [3]. The original execution times are created from the default Trimaran settings, with the exception that the architecture issue rate is set to 16. Execution time is estimated by summing together the frequency of execution of each hyperblock multiplied by the number of cycles it takes to execute the hyperblock assuming a perfect memory system. Infinite results do not restrict the number of operations issued per cycle. 16-way results are obtained by dividing each cycle which has been scheduled with more than 16 operations into `ceiling(total_operations_scheduled_in_cycle / 16)` cycles. The results are normalized to the original schedule generated by Trimaran for a 16-issue machine and scheduled 16-way. The optimized results show the performance after applying PSSA, PSpec, and CHR. The results show that using PSSA with PSpec and CHR results in a significant

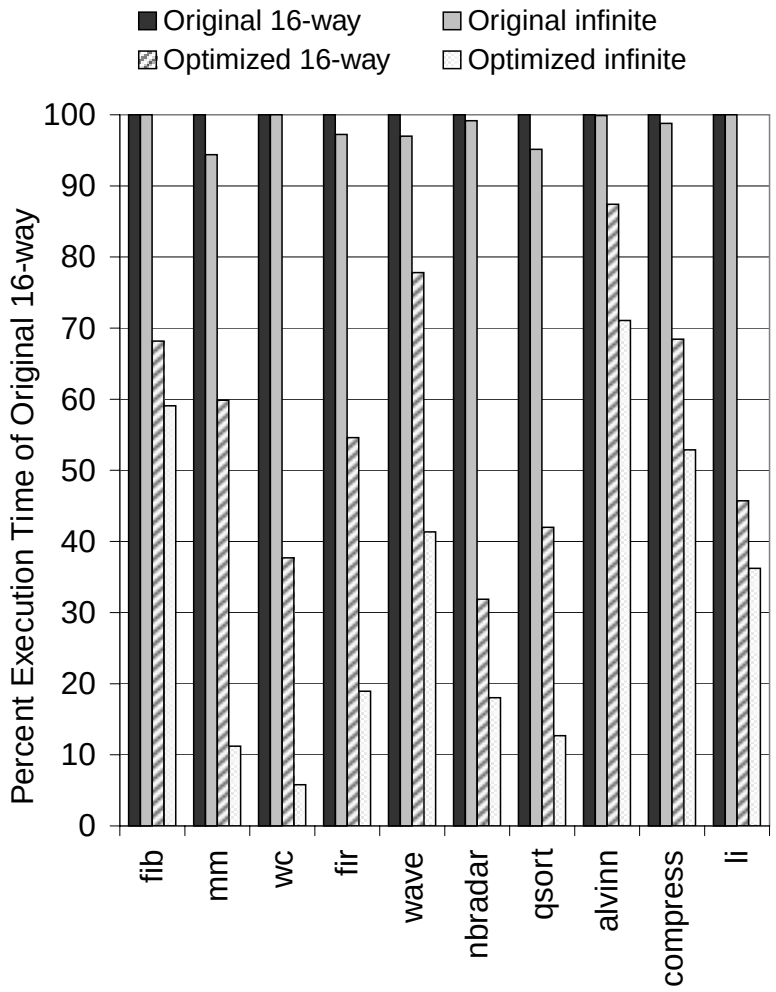


Figure IV.11: Executed cycles normalized to the number of cycles to execute the original code produced by Trimaran for a 16 issue machine.

reduction in executed cycles.

Figure IV.12 shows the average number of operations executed per cycle for the configurations examined in Figure IV.11. In comparing the two graphs for the 16-way results, 3-4 times as many instructions are issued per cycle after applying PSSA, PSpec, and CHR, and this resulted in a reduction in execution time ranging from 12% to 68%. Since PSpec and CHR as applied to PSSA code have the effect of removing the restrictions of control dependence, the optimized infinite results provide a picture of "best case" instruction level parallelism. Inspection of the optimized infinite results of `alvinn`, `compress`, and `li` show that, given current hyperblock formation, peak IPC is somewhat limited. For these benchmarks, even given infinite

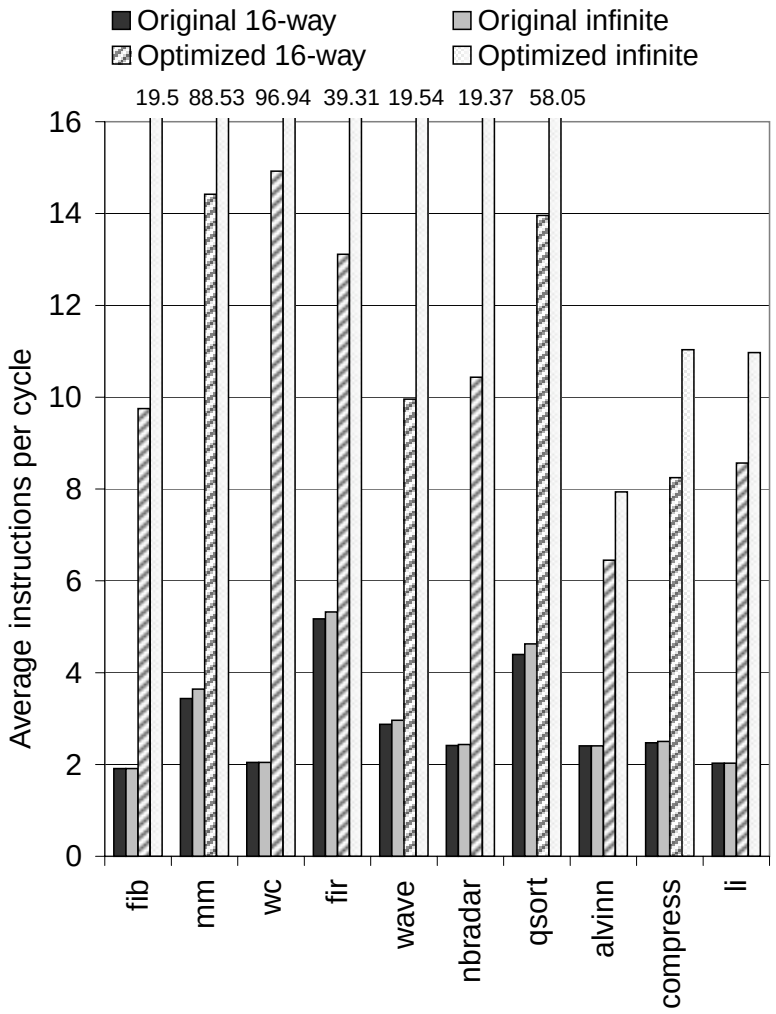


Figure IV.12: Weighted average number of operations scheduled per cycle for hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction. Note that several of the "Optimized infinite" results are greater than 16 – the issue width simulated in these experiments.

execution resources, the current region formation technique with our aggressive optimizations cannot find enough parallelism to significantly reduce execution time.

The renaming required by PSSA and PSpec also significantly increases register pressure. Trimaran's ISA (PlayDoh) supports 4 register files: general purpose, floating point, branch, and predicate [3, 26]. Figure IV.13 shows the average number of live registers for the original code and the optimized code using PSSA, PSpec and CHR. The average live register results are weighted by the frequency of hyperblock execution. For example, `matrix multiply` has on average 17 live general purpose registers in the original code, and 54 live general purpose registers after optimization. Though the increase in utilization of all these register files is notable, the weighted (by hyperblock execution frequency) average utilization mostly still remains within the reported IA-64 register file sizes (128 general purpose, 128 floating point, 8 branch, and 64 predicate) [4].

Additionally, PSSA combined with aggressive PSpec and CHR significantly increases code size – both static and dynamic. Aggressive and resource insensitive application of CHR and PSpec aims to reduce cycles required to schedule at the cost of duplicated code specialized for particular paths (in the case of PSpec) or duplicated code for faster computation of predicates (in the case of CHR). Figure IV.14 shows both the static and dynamic code expansion of the PSSA, PSpec, and CHR optimized code over the original code. We calculate static code expansion by comparing the number of static operations in the optimized code with the number of static operations in the original code. Dynamic code expansion is measured similarly, with the exception that each static operation is weighted by the number of times that it is executed (as calculated by Trimaran's profile-based region weights). This "dynamic code expansion" is intended to capture the run-time effect that the introduced duplicated code will have on the memory system. Dynamic code expansion indicates an increase in the working set size to be supported by the instruction cache.

IV.E Conclusions

This chapter motivated the need for renaming and for predicate analysis that extends across all paths of the hyperblock. It demonstrated how Predicated Static Single Assignment (PSSA), a predicate-sensitive implementation of SSA that implements renaming using full-path predicates, can be used to eliminate false dependences for predicated code. We showed the benefit of using PSSA to enable Predicated Speculation (PSpec) and Control Height Reduction

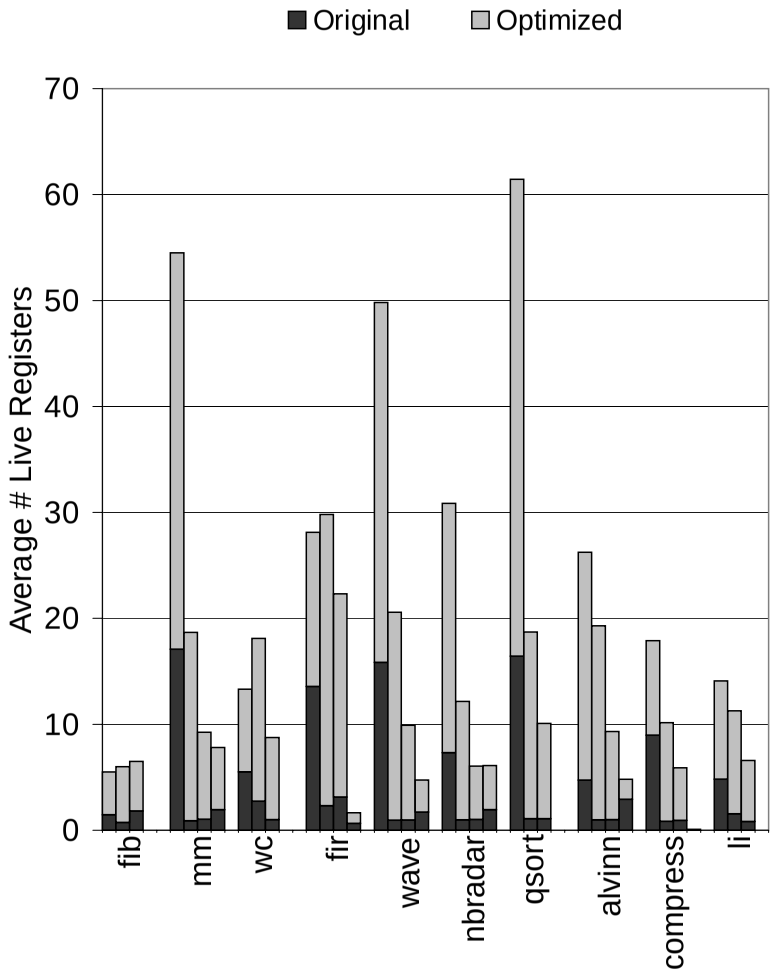


Figure IV.13: Weighted average register pressure in hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction. Shown from left to right for each benchmark is the general purpose file, predicate file, branch file, and floating point file (zero utilization for some benchmarks).

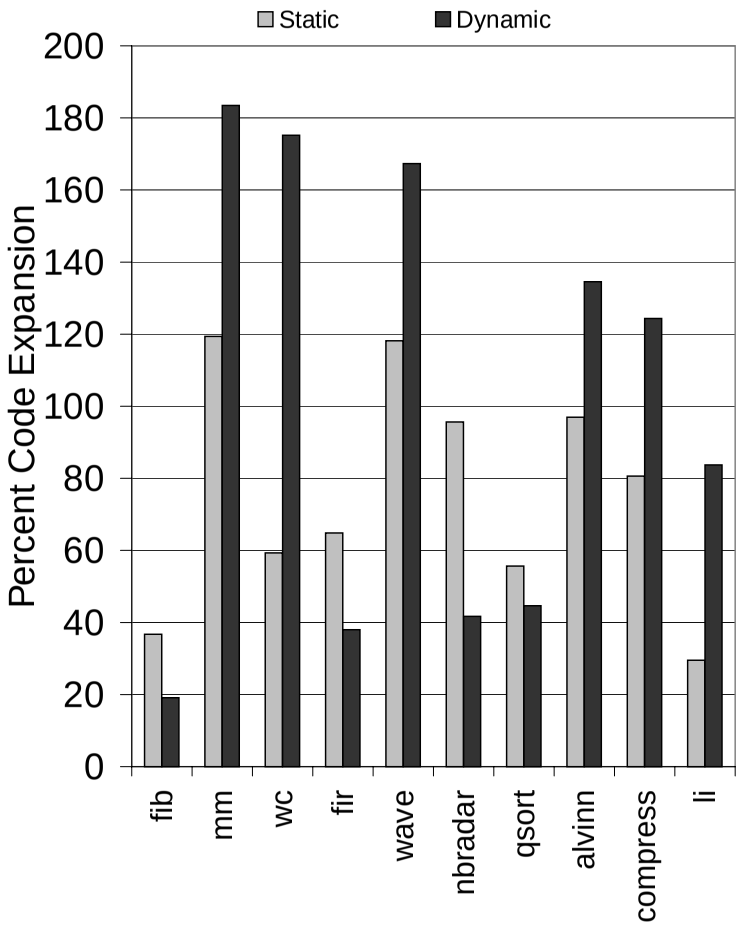


Figure IV.14: Static and Dynamic Code Expansion normalized to original code size. Dynamic code expansion indicates an increase in the working set size to be supported by the instruction cache.

(CHR) during scheduling. Predicated Speculation allows operations to be executed at their earliest schedulable cycle, even before their guarding predicates are determined. Control Height Reduction allows guarding predicates to be defined as soon as possible, reducing the amount of speculation needed.

By maintaining information about each of the original control paths in a hyperblock, PSSA can provide information that allows precise placement of renamed and speculated code, and allows the correct, renamed values to be propagated to subsequent operations. The renaming used by PSSA allows more aggressive speculation, as overwriting live values is no longer a concern. In addition, PSSA supports Control Height Reduction along every control path using full-path predicates, reducing control dependence depth throughout the hyperblock.

Our experiments show that PSSA is an effective tool for optimizing predicated code. We gave extended experiments that show using PSSA with PSpec and CHR results in a reduction in executed cycles ranging from 12% to 68% for a 16 issue machine.

Chapter V

Using Predicate Information in Hardware for Improved Branch Prediction

The application of if-conversion to form predicated regions has an impact on the ability of branch prediction hardware to accurately predict branches. First, predicated code from if-conversion increases the number of branches the hardware must predict. Since multiple paths of code are fetched in the execution of a predicated region, branches along falsely guarded paths will be fetched and predicted, even though they should never affect control flow. These *spurious branches* significantly increase region branch mispredictions. Second, if-conversion transforms control flow information, as indicated by branches, into predicate data flow relationships. Because branch prediction hardware discovers and makes use of control flow information by monitoring the executed code for branches and since branches are removed in the process of predication, some control flow relation information will be lost.

V.A Motivation

We deine *misprediction migration* as the problem of increasing branch mispredictions due to the formation and execution of predicated regions. It is a serious impediment to the benefit one expects to achieve from the process of predication. Misprediction migration stems from two sources: "new" mispredictions from spurious branch predictions and increased mis-

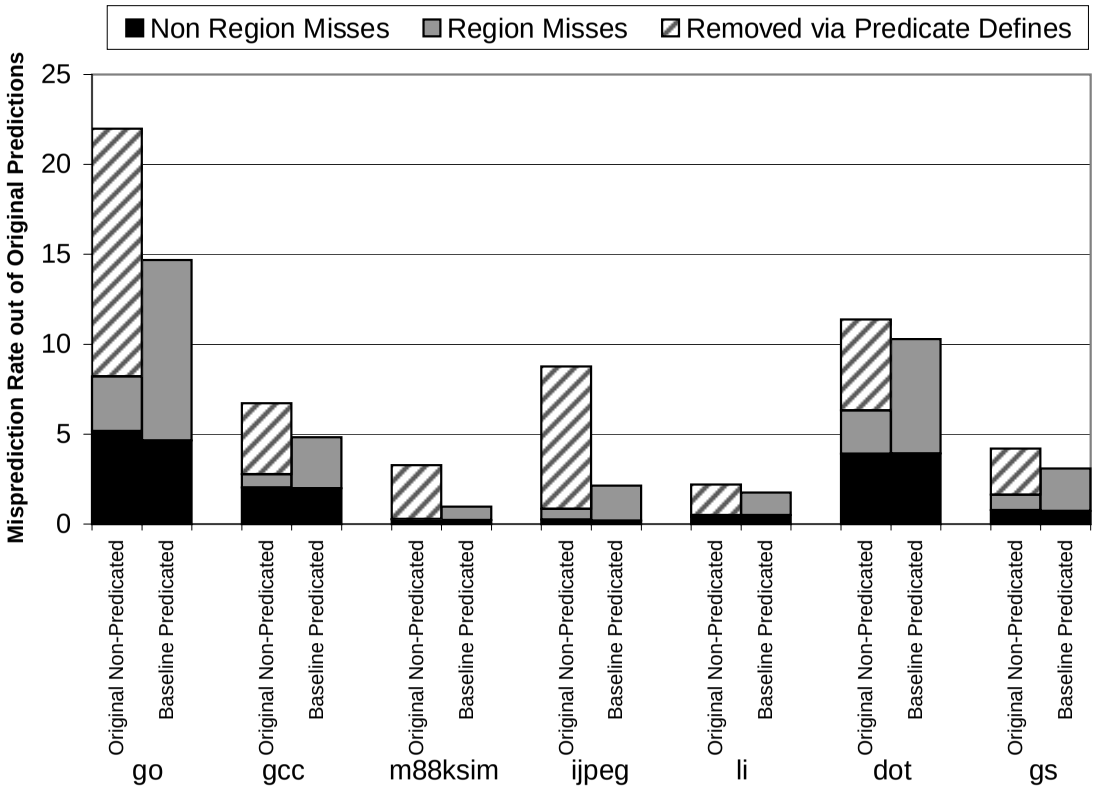


Figure V.1: The impact of misprediction migration in predicated regions. The first column for each benchmark shows a breakdown of where the mispredictions in the program lie: (from bottom to top) in code that will not be included in predicated regions, from branches that will be in predicated regions, and from branches that will be removed by the predication (if-conversion) process. The second column for each benchmark shows how the number of mispredictions from region branches increases by the process of predication. These benchmarks and the predicate region formation used are described in Section V.C.

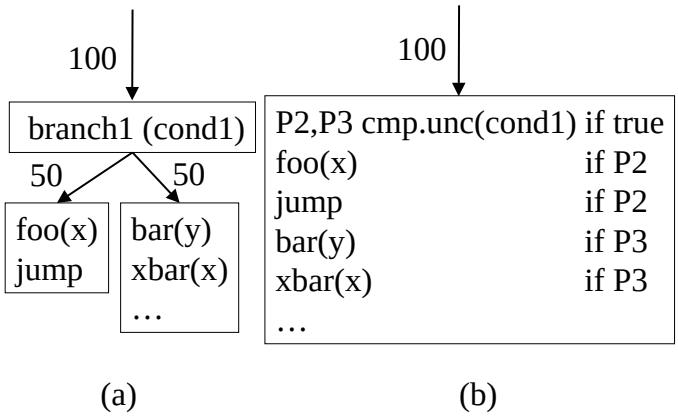


Figure V.2: An example of simple misprediction migration due to increased execution of a predicated region branch (here a jump).

predictions from non-spurious branches due to both spurious *and* removed branch information history.

In Figure V.1 we see the impact of the additional cost of misprediction paid in predicated regions. In the first column for each benchmark we show the miss rate for the original non-predicated code. We categorize these misses as follows:

- Non Region Misses: Misses from outside predicated region areas. These are misses that come from branches in basic blocks not targeted for improvement by the predication process (if-conversion).
- Region Misses: These are misses from branches that remain in predicated regions.
- Removed via Predicate Defines: Misses from branches that will be translated into predicate define statements. These misses will be completely eliminated in the predicated code run because these branches will no longer exist. The price of removing these predicate defines will be increased execution of some false path code.

In the second column we show the mispredict rate of the predicated code where the miss rate is calculated as the number of misses out of the number of branch predictions made in the original, non-predicated code. Clearly, the remaining branches in predicated regions experience a very notable increase in mispredictions for the predicated code. These mispredictions are a result of misprediction migration.

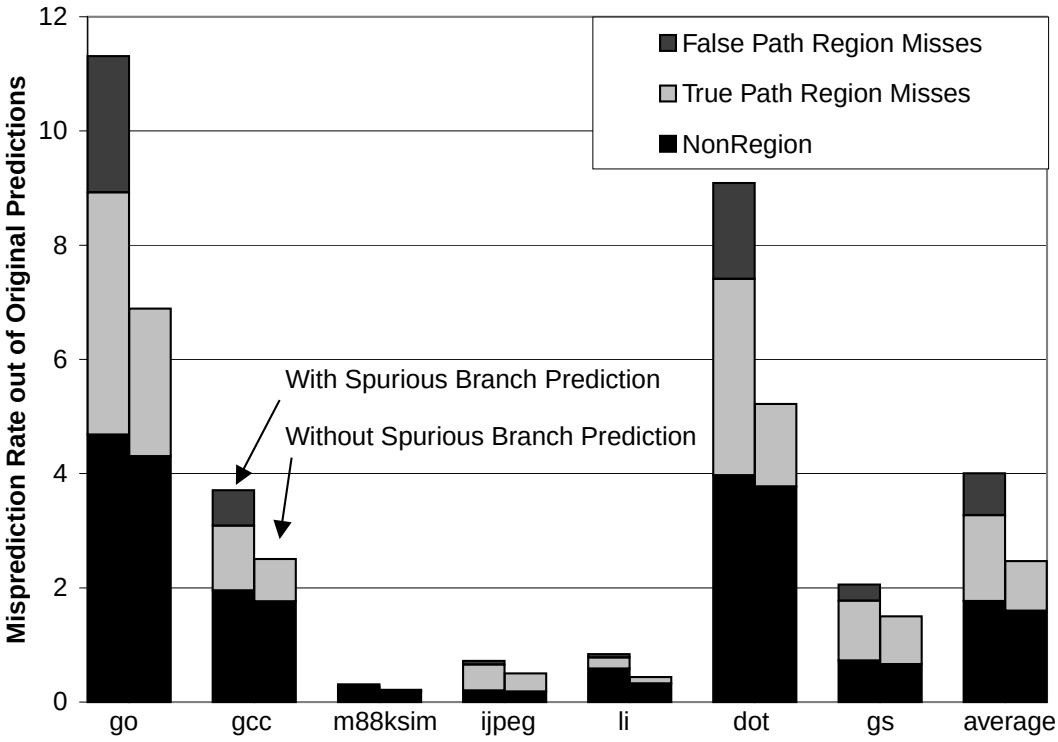


Figure V.3: The impact of only spurious branch prediction on the mispredict rate of a Meta Chooser predictor. In the first column only branches whose guarding predicates will be true (i.e. only the true path of execution through the predicated region) are predicted. In the second column, all branches in regions must be predicted. For both results, predicate define statements update the global history portion of the branch predictor. This isolates the impact of spurious branch prediction from the impact of global correlation modification.

V.A.1 Misprediction Migration from Spurious Branches

Consider Figure V.2. Here `branch1` is originally very unpredictable, but the `jump` instruction is always taken. However, after applying predication, we have caused the `jump` instruction to be transformed into the instruction `jump if P2` because not every execution of the jump will cause a control flow transfer to occur. Whether or not the instruction jumps is now dependent on the value of its guarding predicate. This is an instance where a very straightforward form of misprediction migration will occur. In the predicated code, the `jump` branch will be expected to mispredict in the same way that the `branch1` originally mispredicted. This is because of the increase in predictions of `jump`. While originally `jump` was only executed when `branch1` was taken (50 times in this example), now `jump` is fetched and predicted every time that `branch1` is fetched (100 times). Moreover, the pattern with which `jump if P2` should be predicted taken is *exactly* the same (unpredictable) pattern that `branch1` is taken.

This form of misprediction migration can be addressed by a branch prediction scheme which is cognizant of the branch's guarding predicate value. If one could identify, in fetch, branches whose guarding predicates are false and not predict them, one wouldn't have to "pay" the cost of spurious branch execution. In Figure V.3 we show just the impact that spurious branch execution from predicate region formation has on the number of branch mispredictions in a code. The first column in the graph shows the mispredict rate that is achieved by only predicting those branches whose guarding predicates are true. In the runs for the second column, all branches in a predicated region must be predicted regardless of their guarding predicate value - as is the actual case in current architectures. Clearly the cost of having to predict spurious branches is great - not only in terms of mispredictions from spurious branches, but also indirectly in terms of their impact on the predictability of other branches. In addition to the mispredicts caused directly by their branch execution, when spurious branches are predicted, mispredictions from true path region branches increase by 75% and mispredictions from non region branches increase by 10%, on average. In this comparison the branch predictor is updated with results of the (true path) predicate define operations which reflects the state of the original non-predicated code branch predictor. The only difference is that these branches, now predicate defines, don't have to be predicted. In the next section we explain the importance of this predicate update.

V.A.2 More Complex Misprediction Migration Sources: Global Correlation

While spurious branch prediction is a significant source of misprediction migration, there are other impacts on branch prediction other than extra execution of branches. The process of if-conversion to form predicated regions replaces hard-to-predict branches with predicate defines. Modern branch predictors utilize *global branch history* to identify dynamic relationships, also known as *global correlation*, between branches in order to make more accurate predictions. By removing some branches and replacing them with predicate defines, one removes potentially important branch behavior information from the view of these global correlation schemes.

Consider the code shown in Figure V.4(a). Assume that branch `b>a` is frequently mispredicted. Even so, given a branch predictor that utilizes at least 2 bits of global history, branch `a>100` would be easily predictable. The global history path ending in `T,T` would predict that `a>100` should be taken and any other values for the last 2 bits of history should predict that `a>100` not be taken. This is because `a` is only assigned a value greater than `100` when both occurrences of the statement `a = a+50` are executed. [1] The predictions that would be realized from a 2-bit global history pattern are shown in the left side of Figure V.4(c). Only a pattern with the last two bits of history set to "taken" should predict taken.

We may choose to form a predicated region starting from the hard-to-predict branch `b>a` that includes both the taken (`a = a+50`) and fall-through (`a>100`) paths. Figure V.4(b) shows this process whereby the branch `b>a` is replaced with a predicate define `P2 b>a` which sets predicate `P2` to true when `b>a` is true and false otherwise. Both statements `a = a+50` and `a>100` are always fetched and executed, but `a = a+50` is only committed when `P2` is true. However, since `b>a` is no longer a branch, the global branch history will no longer contain any information about the relationship between `b` and `a` which controls the eventual value of `a`. Clearly, the very bit of information which allowed us to predict branch `a>100` is no longer being recorded, and this branch will now mispredict in much the same way that that `b>a` mispredicted. This loss of branch history information is another cause of misprediction migration. The right side of Figure V.4(c) shows the new global history branch prediction table, where the loss of the second branch entry causes the "taken" prediction of branch `a>100` to become intermingled with a path that should predict "not taken".

[1] This execution path is highlighted with bold arrows in (a).

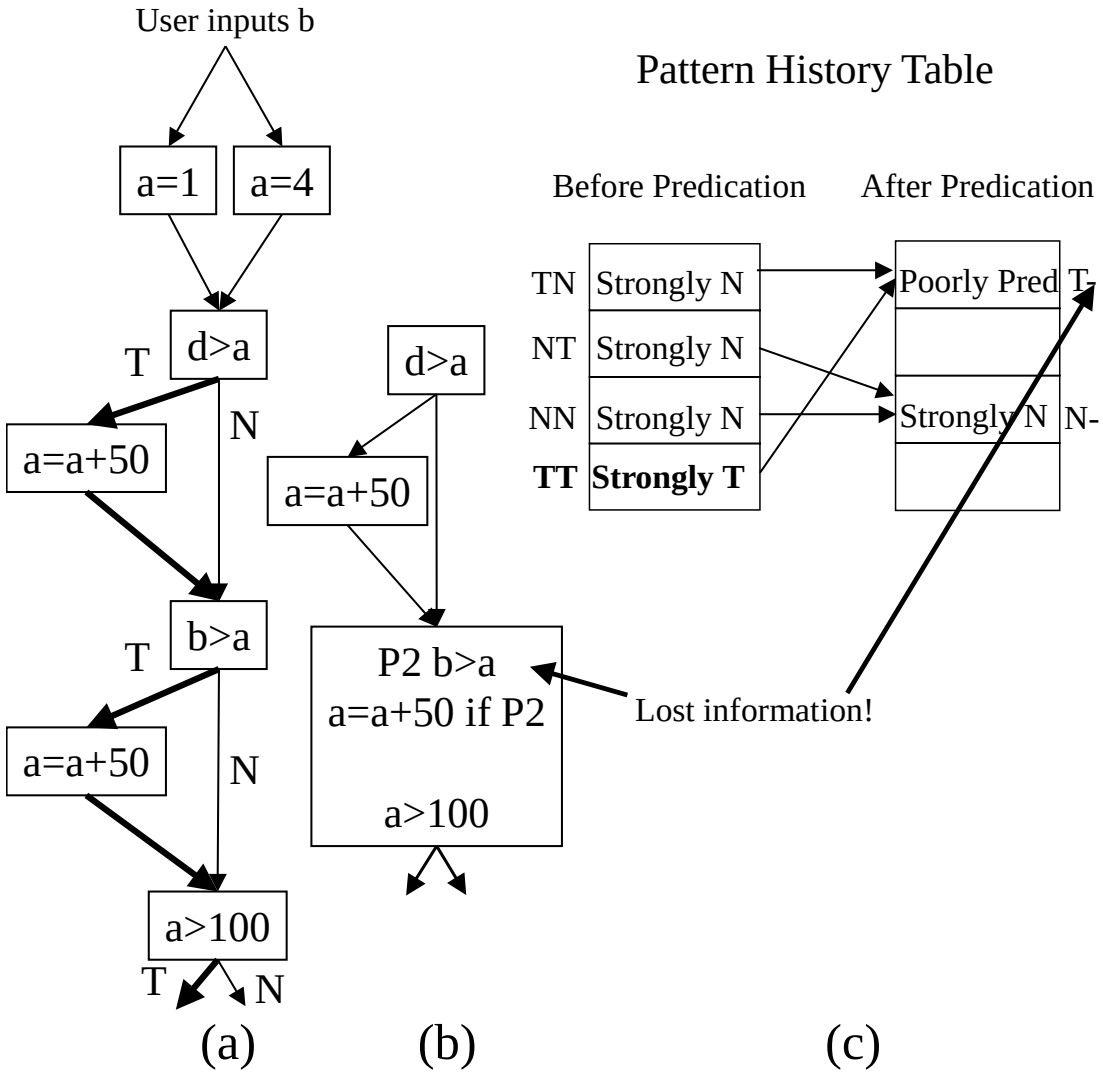


Figure V.4: Code showing the problem of misprediction migration due to the predication of a branch. (a) shows the original control flow graph where `b>a` is frequently mispredicted, but `a>100` can be readily predicted given two bits of global branch information. The left side of part (c) shows the prediction information that would be gathered for branch `a>100` from the code in part (a). (b) shows a portion of the code after the bottom three blocks are formed into a predicated region to alleviate the mispredictions from branch `b>a`. The right hand side of part (c) shows how this removes a bit of history from the branch prediction architecture, effectively merging two paths of history for branch `a>100` and making it very unpredictable.

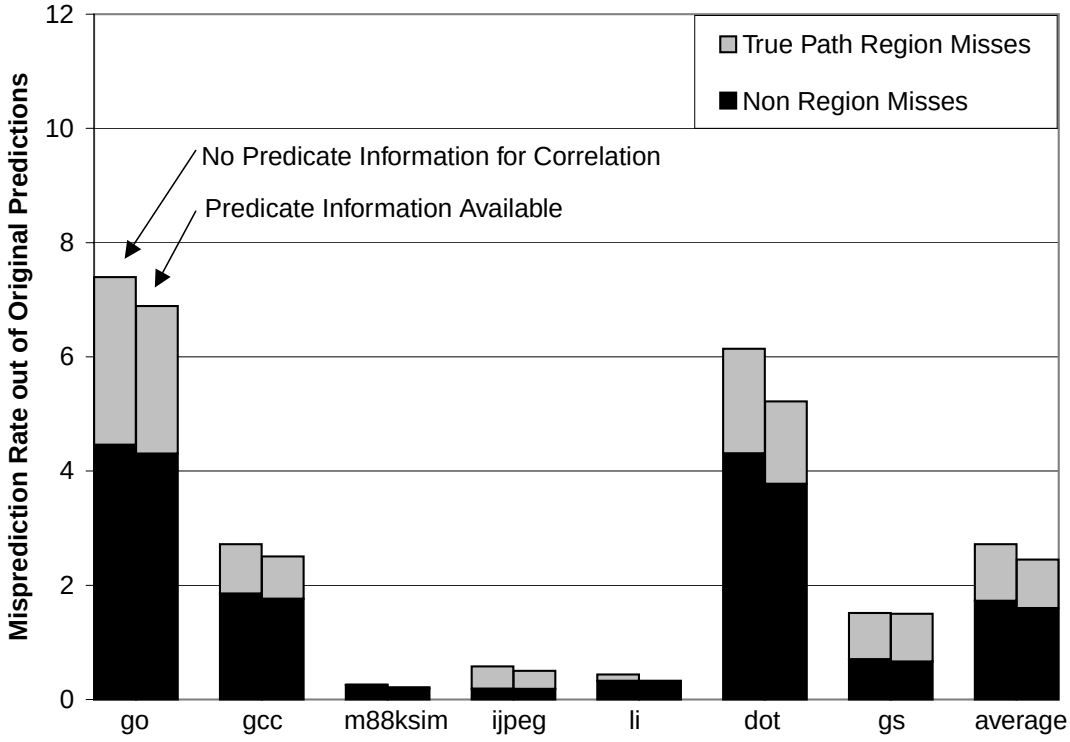


Figure V.5: The impact of the loss of predicate information on global correlation in a Meta Chooser predictor. For both cases only true path branches are predicted. This isolates the impact of predicate information for global correlation from the impact of spurious branch prediction. In the first column predicate defines are allowed to update the global history register of the Meta Chooser as if they were branches. In the second column, we show the default behavior for a Meta Chooser in a predicated region - predicate defines do not interact with the branch prediction architecture in any way.

In Figure V.5 we examine the impact of the removal of branch history by the process of predicate region formation. In the first column we show the mispredict rate for a predicated code where the results of predicate define statements are used to update the branch predictor. We assume that these results can be known immediately when the predicate define enters the pipeline so as to model the exact history that would be available if these statements had remained branches. In the second column, we show that by not allowing predicate defines to update the branch prediction scheme additional branches mispredict. In both of these experiments, only the instructions guarded on true predicates predicted and/or updated the branch predictor so as to exclude the effects of spurious branches. Therefore we can show that the increase in misprediction rate shown here is due solely to the loss of global correlation information from predicate defines.

V.A.3 Outline of the Chapter

In the rest of this chapter we will investigate various methods which allow predicate define information to update the branch predictor.

In section V.B, we begin by concisely outlining the different predicate update branch prediction schemes we'll motivate in the rest of the chapter. Motivation for the development of these predictors will follow later in chapter in sections V.D through V.E.

Next, we provide a full description of the methodology of our experiments. Section V.C gives a complete description of the methods we used for forming and simulating predicated codes and branch prediction architectures. We also describe the benchmarks studied. A summary is provided in section V.C.5.

In section V.D, we begin our investigation of predicate update branch prediction schemes by attempting to identify spurious branches at prediction time. We show how to use specific branch/guarding predicate relationship information to augment both local history and global history schemes to separate out spurious branch predictions from true path branch predictions. Then, in section V.D.4 we'll show how to obtain more accurate predicate information for use in these same schemes. Finally, in section V.D.5, we propose a predicate-aware branch filtering technique based on accurate branch/guarding predicate knowledge which can reduce the impact of spurious branches for any type of branch prediction scheme.

In section V.E.1 we explore a new method of utilizing predicate define information by incorporating predicate information directly into the global history register. This eliminates the necessity of storing direct branch/guarding predicate information and of a two table lookup to

produce a predicate-aware branch prediction, which previous techniques require. This method can also be augmented with a predicate-aware filtering scheme. Finally, in section V.E.5 we show how this method has the effect of restoring lost global correlation information, but that the impact of spurious branch execution conceals the potential benefit from global correlation.

Section V.F compares the proposed architecture designs from the point of implementation cost, potential performance problems, and comparative benefit and in section V.G we make some concluding remarks.

V.B Overview of Predicate Update Branch Prediction Architecture Designs

In this section we provide a very high-level overview of the predicate update branch prediction architectures that we present in the rest of this chapter. We investigate four basic predicate update branch prediction designs. All can be incorporated as part of or used in conjunction with a Meta Chooser branch prediction architecture. Various designs can be combined with each other to produce complementary effects. Here we highlight the basic purpose of each scheme in order to provide the intent of each scheme without regard to its implementation details:

- *Local History Predicate Update with PEP:* The Predicate Enhanced Prediction (PEP) scheme proposed by August et al. [12] utilizes predicate information in a local history branch prediction scheme. The goal of PEP is to read the value of a branch's guarding predicate when the branch is being predicted and use that to separate out true path predictions from spurious (false paths) predictions. This is done by maintaining two separate local histories for each branch - one which is used when the branch's guarding predicate is true and one which is used when the branch's guarding predicate is false.

- *Global History Predicate Update with APPEND:* The APPEND predictor we propose has the same goal and basic methodology as PEP, but for a global history prediction scheme. In APPEND, we separate out true path and spurious path predictions by appending the value of a branch's guarding predicate onto the current global history register to form the history that is used to index into the global pattern history table. In this way we produce different indices into the global pattern history table depending on whether a branch's guarding predicate is true or false.

- *Spurious Branch Filtration with Squash-FP:* The Squash-FP branch prediction filter does not update the traditional branch prediction architecture with predicate information at all. Rather, it directly identifies branches whose guarding predicates are known to be false and predicts them to be not taken, since a spurious branch should never affect control flow in a program. This has the effect of filtering spurious branches from traditional branch prediction architectures, reducing the contention in those structures.

- *Global History Predicate Update with PGCU:* The Predicate Global Correlation Update predictor (PGCU) utilizes predicate information very differently that either PEP or APPEND. It directly incorporates all true path predicate define information into the global history register by allowing predicate define statements to update the global history register with their result. It provides spurious path information indirectly through entries made by predicate defines into the global history register, rather than explicitly maintaining or creating separate histories for true and spurious executions of a branch.

Next we present the experimental setup we used in developing and evaluating the effectiveness of these predicate update branch prediction schemes.

V.C Experimental Setup

In order to investigate the impact of various branch prediction architectures on codes containing predicated regions, we would like to have a detailed processor pipeline simulator that supports the IA-64 instruction set as well as source code for an optimizing compiler that supports general if-conversion. A detailed pipeline simulator measures the time spent in each stage of the pipeline and gives accurate information regarding execution time of the code produced by the compiler on the simulated architecture. However, neither of these were available. Generating either, much less both, of these components "in-house" was beyond the scope of our available time-frame. Instead, we develop a solution that allows us to simulate (and control) predicated region formation and then simulate without a detailed pipeline simulator just the branch prediction hardware we develop. Consequently, our results are presented in terms of change in branch misprediction rates, rather than change in execution time - which is only possible given a detailed pipeline simulator. In the next four sections we provide a detailed description of the simulation process we follow. An overview of this information is summarized in Section V.C.5; the details of implementation given here are not necessary to understand the results presented in the rest of the chapter.

V.C.1 Using the ATOM Binary Instrumentation Tool to Emulate Branch Prediction Hardware

The simulator we use is ATOM [42] which supports instruction-level simulation of a code. This is done by modifying an Alpha binary of the code to be simulated with instrumentation calls before or after the instructions to be simulated. In our case, to model a baseline branch predictor on original, unpredicated code, we insert an instrumentation call immediately before every branch instruction. At this point, all of the information that is available to the branch is available for our instrumentation function. This uses the branch's address (PC) to index into our simulated branch prediction architecture. The prediction we produce can be compared for correctness against the actual direction that the branch follows as determined by the branch condition and we can record the branch prediction rate.

The first step in our experimental process is to gather branch prediction information for the original non-predicated binaries. From this we create a list of the most frequently mispredicting branches for each benchmark we study. Since the goal of our work is to reduce branch mispredictions, we want to replace frequently mispredicting branches with predicate defines. We create a list of the top 10% most frequently mispredicting branches for each benchmark. This list will be input to our predicate region formation algorithm.

V.C.2 Emulating Predicated Execution in ATOM

ATOM only supports instrumentation of Alpha binaries. The Alpha ISA does *not* support predicated execution. We modify the Alpha ISA by adding a guarding predicate to every instruction and by creating predicate define instructions. We then simulate this new ISA.

Because the original Alpha ISA does not support predicated execution, there is no compiler we can use to create a binary with if-converted sections of predicated code that we can use as input to our simulator. Instead, we develop our own predicated region formation scheme. This scheme uses a list of branch addresses that we have identified as hard-to-predict. For each branch in the list from most to least frequently mispredicting, we try to form a "predicated region". Since the Alpha ISA doesn't actually support predication, in effect we produce a list of basic blocks associated with the hard-to-predict branch that triggered the formation of this predicated region. We say that the blocks in this list are part of the predicated region that will be emulated when control reaches the given hard-to-predict branch. All of the instructions from these basic blocks are guarded on a predicate register.

Region Formation in ATOM

The formation of this list of region blocks is malleable. Here we present the scheme that is used to present the results in this chapter. Several schemes were evaluated. Section V.C.4 provides more details on our choice of region formation algorithm. In the results provided here, regions were formed by walking the control flow graph of basic blocks following an identified hard-to-predict branch for a depth of up to five basic blocks after the branch. This walk may end prematurely if one of two things occurs: we try to walk to a block that has already been incorporated in another predicated region or if we encounter an indirect branch (which can have more than two successor blocks and cannot be transformed into a predicate define). Additionally, the walk may be lengthened beyond five basic blocks if, as we include basic blocks in the region, we encounter another branch on our list of hard-to-predict branches. At that point, the depth count at that basic block is reset to zero and region formation continues.

For each block we choose to include in the region, we label it as one of the following:

- Non Control Region Block: A block that ends with no control flow changing instruction (i.e. a branch or jump). These blocks have only one successor block.
- Predicated Block: A block that ends in a conditional branch and both of whose control flow successor blocks have also been included in region. These branches will be "transformed" into predicate defines. In our simulation, we mark these branches as predicate defines and do not predict them.
- Loop Footer Block: A block that ends in a conditional branch whose taken successor (i.e. the head of the loop) is also in the region. This is a branch that would potentially be transformed into a predicate define, but the branch is required because of the loop.
- Region Leaving Block: A block that has one or more of its control flow successors that is *not* part of the region. This may happen because the block ends in an indirect branch (i.e. procedure call or return) or because one of the successors of the conditional branch is already a member of another predicated region.

Region Scheduling in ATOM

In the process of including multiple basic blocks into one "region" of code to be executed, we need to redefine a schedule for that section of code. In the original code, our "schedule" is to execute all the instructions in a basic block, follow a control flow edge leaving

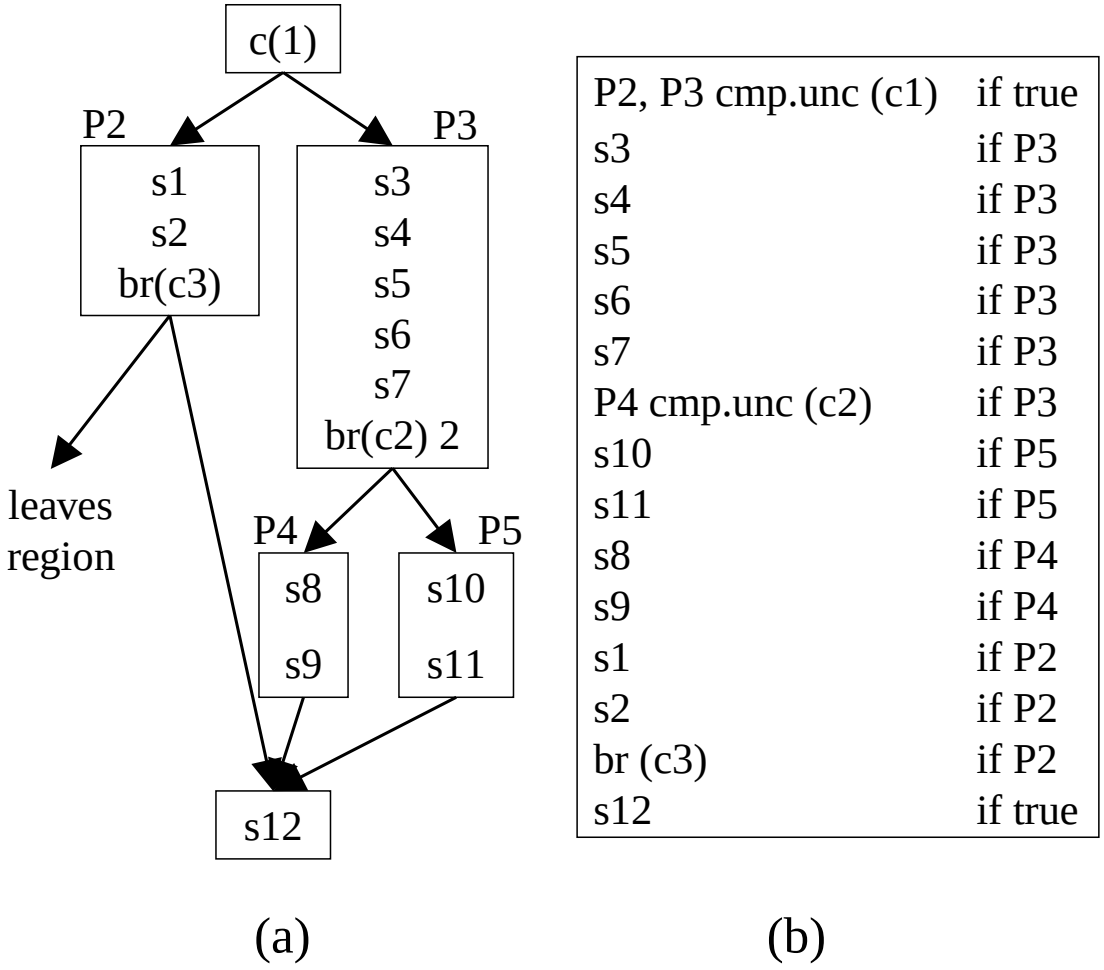


Figure V.6: An example of the impact predicate region formation has on scheduled code. The blocks shown in (a) are selected for inclusion in a predicated region. This means that all instructions from these blocks will be fetched, executed and then either committed or nullified every time program execution reaches statement `s1`. The order of the instructions in the predicated region (specifically as relative to the placement of region branches) has an impact on the amount of spurious execution of instructions that will occur.

the block, and repeat the process. In emulation of a predicated region, we need to consider the "ordering" of the basic blocks that we have included in the region. While a linear ordering of the blocks could be defined and used to emulate the region, the results would be very unrepresentative of the kind of code a compiler that supports predicated execution would produce. Figure V.6(b) shows what a straight forward ordering of basic blocks in a region would mean for emulation of the predicated region. When `P2` is true, we may spuriously execute statements 3-11 before we reach the scheduled location of the code guarded on `P2`, causing significant wasted work in terms of spurious instruction execution. While other options exist, we chose to:

- Assign predicate values to basic blocks (and the instructions in them) via a modified Predicated Static Single Assignment method. The modification we use is that no join blocks ever have their block predicate materialized. This is so that no "predicate define" statements are created in any location other than that of a branch ending a Predicated Block.
- Parse the blocks included in the predicated region for data flow dependences.
- Schedule for an 8-wide processor pipeline with no resource constraints (i.e. there is no restriction on the types of instructions that can be scheduled in any 8-wide instruction). This scheduling does not actually rearrange instructions in the Alpha binary. It creates a list of instruction PCs indicating the new order (or schedule) in which they are executed.
- After scheduling we annotate branches that have targets in the region (i.e. loops) with the schedule cycle of their taken successor.

Additionally, for our predicate update branch prediction architectures we would like to allow as much time as possible between predicate defines and the branches that are guarded by them. Within the confines of data dependences and loop boundaries, we try to schedule predicate defines as early as possible and branches as late as possible within a predicated region. This is not optimal and alternatives are discussed in Section V.D.6. However, our implemented region formation and scheduling techniques allowed us a wide range of exploration with regards to predicate define-branch scheduling placement. We produced codes with widely ranging numbers of branches whose guarding predicate is resolved: in `li` only 7.5% of all region branches have their guarding predicate resolved at fetch, while in `m88ksim` all region branches have their guarding predicate resolved.

Region Run-Time Emulation in ATOM

Because the ordering of branches (and branches now marked as "predicate defines") in our region schedule may differ from the order in which these branches will be processed in ATOM emulation, we have to change how we trigger our branch-prediction-emulating instrumentation routine. There are three basic steps to predicate region emulation: 1) recognize when the start of a region is encountered, 2) identify when execution leaves the region, and 3) simulate execution of the region.

As we execute our instrumented ATOM binary, we recognize when we enter a predicated region by identifying the top block that contains the re-scheduled code we have created for the region. At this point, we begin creating a trace of region execution that captures all branch (control flow) behavior of the basic blocks in the region - but does not update our branch prediction architecture. We continue this trace collection until we encounter a basic block that is not part of the region. We then trigger a "replay" of our region trace in conjunction with our region schedule that effectively emulates our region schedule of instructions for this dynamic instance of the predicated region.

In this replay, every instruction in the schedule has a guarding predicate which indicates whether it is "really executed" or not. When we encounter true path predicate define instructions (whose guarding predicates are true), they update our emulated predicate register file and potentially trigger branch predictor functionality. When we encounter branches, they are predicted. However, we note that a "correct" prediction for a spurious (falsely guarded) branch is always not taken (do not leave the region or do not branch to the top of a loop).

V.C.3 Providing Some Detailed Pipeline Information via SimpleScalar

We do have one compelling need for detailed pipeline timing information. We want to allow predicate define statements to update the branch prediction hardware, but realize that while branches can update in the "fetch" stage of a pipeline, predicate defines can't update until "commit". This means that any simulation of branch prediction hardware update needs to support a delayed update by predicate define statements. Since our simulations are run on an instruction-level granularity, we chose to model this delay in terms of instructions. We use SimpleScalar 3.0a [15] as a detailed pipeline simulator that processes Alpha binaries. For the SimpleScalar runs we simulate an 8-wide issue out-of-order machine with a 128-entry RUU. The L1 data cache is 64K 4-way associative, L1 instruction cache is 32K 2-way associative, and

we use a unified 1 MB 4-way L2 cache. The L1 miss with L2 hit latency is 12 cycles and there is a 120 cycle latency for an L1 and L2 miss. The minimum branch misprediction penalty is eight cycles, and we use a 32-entry return address stack for predicting return instructions.

We cannot gather fully accurate information of the number of instructions that are fetched between the time a branch (one which we'll translate into a predicate define) fetches and the time it commits. This number is variable for each dynamic execution of the branch depending on processor pipeline events such as data or instruction cache misses. Additionally, choose not to emulate in SimpleScalar the predicated regions that we form and emulate in ATOM. However, we attempt to model an average latency for each branch by gathering the average number of instructions that are fetched after a branch is fetched and before it commits. Discussion of other ways to model latency for deterministic update of predicate defines is discussed in more detail in section V.E.2.

This information will be used for branches transformed into predicate defines in our ATOM simulations to delay any update of branch prediction hardware by predicate define statements. Essentially, in ATOM, branches update branch prediction hardware immediately upon being replayed in the schedule. When a predicate define is replayed in the schedule, it submits its information to a queue in the branch prediction table and tags that entry with the delay gathered via the SimpleScalar run. Every instruction passed in the replay of the region schedule decrements the delays of the predicate define queue, and upon the delay reaching zero, that predicate define information updates the branch prediction hardware.

V.C.4 Predicated Region Formation for Hard-to-Predict Branches

Our region formation algorithm starts from a list of hard-to-predict branches that we want to target for translation to predicate defines. For the experiments presented here, we start from a list of the top 10% most frequently mispredicting static branches in each benchmark. Original mispredict values are gathered with a baseline Meta Chooser predictor [27] which is detailed in Section II.C.2. For the SPEC95 program `go` 87% of all mispredicts in the program can be attributed to the top 10% most frequently mispredicting static branches.

For a given hard-to-predict branch, we walk the control flow graph following the branch incorporating basic blocks into the predicated region. We continue adding successor blocks in a breadth-first fashion until we reach a depth of five basic blocks from the hard-to-predict branch. If, while walking, we encounter another member on the list of hard-to-predict branches, the depth count along that path is reset to zero. This method attempts to target the

most frequently mispredicting branches for removal and provides sufficient quantity of post-branch work to overlap the execution of the branch converted to a predicate definition in the pipeline. Additionally, this method provided sufficient scope for our scheduler to investigate a range of region schedules as discussed in Section V.C.2.

If a block in the region originally ended in a branch and both of its control flow successor blocks have also been included in the region, then the branch is translated into a predicate define and the successor blocks are assigned the appropriate guarding predicates. If either of the block's successors were not included in the region, then the branch becomes a region branch.

There are additional measures we use in controlling region formation. First any successor block which is reached less than 10% of the time that the region is entered is excluded from the region. This keeps cold blocks from unnecessarily bloating the region with infrequently useful work. Second, any block ending in an indirect branch or return automatically stops region formation along that path and similarly for any branch with a successor that has already been included in a previously formed region.

V.C.5 Overview of Instrumentation Setup

We gather results for a subset of the SPEC95 benchmarks (`go`, `gcc`, `m88ksim`, and `ijpeg`), SPEC92 `li`, as well as two other benchmarks. `Dot` is a project from AT&T for plotting graphs, and `gs` is a run of ghostscript translating a paper from postscript to jpeg format. We chose these benchmarks because they had a reasonable number of mispredictions. We used the same input to the applications to generate the profile to guide region formation and to gather the misprediction results via simulation.

We quantify our branch prediction architectures examining the improvements in branch misprediction rates, which are all normalized to the number of branch mispredictions in the original non-predicated code. In addition, we examine the percent increase in instructions executed for the hard-to-predict predicate regions formed.

For all the results in this chapter, the misprediction "rate" is calculated as the number of mispredicts divided by the number of branch predictor accesses in the *original* execution of the program code. This allows us to look at one consistent metric as the number of branch predictor accesses will change with the predicated code. In addition, the miss rates we show include the misprediction rate for all branch types. This includes conditional branches, returns, unconditional, procedure calls, and indirect branches.

We did not have available a detailed pipeline simulator which supports predicated execution or source code of a compiler that produces predicated code. Instead, we simulate predicated execution for the purposes of branch prediction behavior in the ATOM [42] binary instrumentation tool using select processor pipeline timing information from SimpleScalar3.0a [15]. This simulation infrastructure is described in detail in Section V.C.1 - Section V.C.4. Both ATOM and SimpleScalar support Alpha binaries.

Our SimpleScalar simulated parameters are:

- out-of-order execution
- 8-wide issue
- 128-entry RUU
- 64K 4-way set associative L1 data cache
- 32K 2-way set associative L1 instruction cache
- 1MB unified (instruction and data) L2 cache
- 12 cycle L1 miss-with-hit-in-L2 latency
- 120 cycle miss-in-L1-and-L2 latency
- 8 cycle minimum branch misprediction penalty
- 32-entry return address stack for predicting return instructions

In ATOM we implement a Meta Chooser branch prediction architecture (defined in Section II.C.2). Except where explicitly mentioned, we use the following simulation parameters for our Meta Chooser branch prediction architecture implementations:

- 12-bit global history register
- 4K-entry local pattern table (storing 12-bit per-branch histories)
- 4K-entry local pattern history table
- 4K-entry global pattern history table
- 4K-entry chooser table

V.D Spurious Branch Prediction via Direct Guarding Predicate Information

While if-conversion is recognized as trading the cost of branch mispredictions for the cost of execution of spurious path instructions, this cost is increased when branches are among those spurious instructions. Not only do they consume a processors functional resources, but they may cause additional penalties by mispredicting more frequently than they did in the original code.

In the process of predication, branches in basic blocks incorporated into predicated regions do not have to be transformed into predicate defines. If the branch is an indirect branch (potentially with more than two possible targets) or if one of the branch target blocks is excluded from the region for some reason (of region formation) then then branch will remain a branch but will be guarded by the appropriate predicate define. These region branches will need to be predicted during fetch more frequently than they were in the original, non-predicated code because they will be fetched and predicted even when their guarding predicate is false. *Spurious branches* are those dynamic instances when a branch is fetched, but its guarding predicate is false. This execution of spurious branches can cause what we call *misprediction migration*, where the poorly predictable pattern of a hard-to-predict branch chosen to be predicated is merely migrated to a region branch. Figure V.2 in Section V.A.1 shows a very basic example of misprediction migration stemming solely from spurious branch prediction.

This spurious branch information impacts a Meta Chooser branch predictor in two ways. First, region branches may mispredict more frequently that they did in the original code due to their "spurious" path executions. Since region branches are executed, and hence predicted, more frequently after if-conversion, they may mispredict more often. Second, this spurious branch execution increases utilization of the branch history and prediction tables by making entries to predict these spurious branches.

Next we will outline three prediction schemes that attempt to reduce the impact of spurious branch predictions by utilizing information about the value of a branch's guarding predicate. These schemes are similar in that they store the guarding predicate register number for each predicated branch and use that to read a value from the predicate register file. The branch's guarding predicate register value is used to try to identify and separate out predictions of spurious branches.

V.D.1 The PEP local predictor

Previous work by August et al. [12] investigated the poor execution of spurious branches in a local only prediction scheme. They propose a solution for use in a local branch prediction scheme as outlined in Figure V.8. This scheme attempts to reduce the number of mispredictions caused by spurious branch execution, but does not attempt to reduce the utilization of prediction resources caused by spurious branch execution. In fact, this method actually devotes more branch prediction history resources to spurious branches - in an attempt to recover true path branch history by separating out spurious branch history.

In Figure V.7(a) we see a control flow graph representation of the predicated region shown in (b). Above Block 1 in (a) we outline a possible trace of code execution before reaching the predicated region. Along this trace, the last three control flow changes that occur are: `branch1` is taken, `branch2` is not taken, and `branch3 is taken`. In part (c) of the example, we show how PEP stores two local histories for branch `(b>c)` as well as information about the predicate register number of the branch's guarding predicate. The PEP predictor reads the value of the guarding predicate from the predicate register file and uses that to indicate which per-branch history should be used to predict the branch. The intention is that by separating out the spurious predictions of a branch into their own local history that a) those predictions will converge to "predict not taken" since a spurious branch never causes a control flow change and b) that the original (non-predicated) local history pattern for a branch will be captured in the true local history. This will only produce valuable results when the definition of the branch's guarding predicate can complete execution before the branch is predicted in fetch. This is partially determined by their relative scheduling of predicate define and branch instructions as is described in section V.D.6.

The PEP scheme, shown in Figure V.8 as part of a Meta Chooser predictor, records a branch's guarding predicate register number with its entry in the BTB (which provides the branch target address for taken branches). Additionally, two local histories are stored in the local history table - one associated with the branch behavior when the guarding predicate is true, and one when it is false. The goal is that the "true history" should be used and updated with the same pattern that the original branch was accessed - since it will only be used and updated when the branch's guarding predicate is true. The "false history" should only be used and updated when the branch's guarding predicate is false - hopefully producing a prediction of "not taken".

A branch prediction with PEP takes two serial table lookups. The first lookup accesses

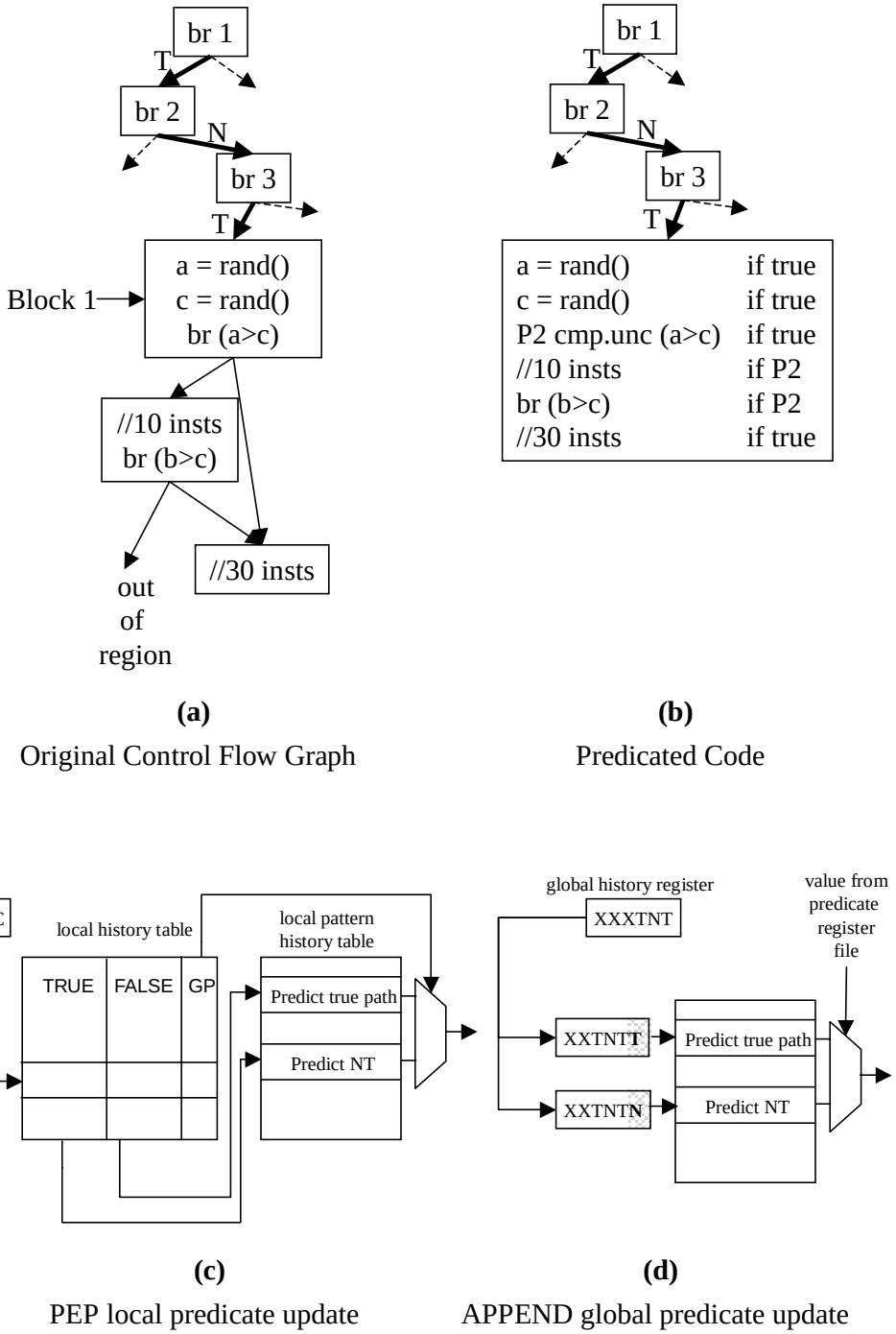


Figure V.7: Example of behavior of PEP and APPEND predictors for local predicate-aware and global predicate-aware predictions, respectively. Predictions are shown for the region branch `br(b>c)`. PEP maintains separate per-branch histories for the true path and false path instances of each branch. APPEND appends either a `1` or `0` to the current global history register to create an index for a true path prediction or a false path prediction, respectively. In (d) the grey section of each index represents the guarding predicate value appended onto the global history register.

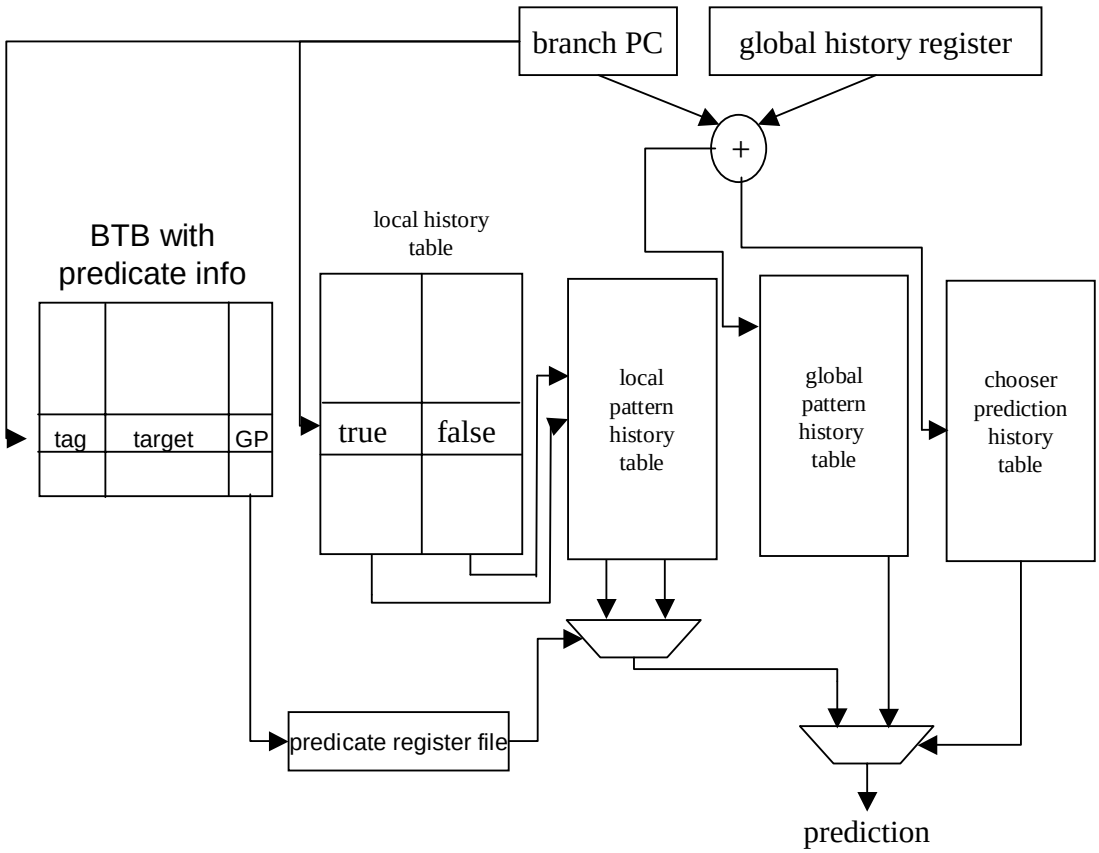


Figure V.8: A schematic of the PEP local history predicate-aware predictor as part of a Meta Chooser predictor. PEP separates out true and false path predictions by maintaining separate true and false path local histories in the local history table. PEP uses direct branch/guarding predicate information stored in the BTB to look up the current value of a branch's guarding predicate and to then select a prediction from the two produced by the true path history and false path history, respectively. PEP requires a serial two table lookup to retrieve the branch's guarding predicate value.

the BTB providing the guarding predicate associated with the branch. Then another lookup is made in the predicate register file to find the currently available predicate value. The predicate value is then used to choose between the predictions found by indexing a 2-bit pattern history table using the histories from the local history table. We assume that the local histories are speculatively updated at fetch and correctly recovered on a branch misprediction.

V.D.2 A new global predicate update predictor: APPEND

We propose a similar approach for a global prediction scheme where the global history register is *appended* with the guarding predicate information. This will have the same affect as PEP for the global prediction history - we attempt to divide predictions of the same global history into two categories and to two entries in the global prediction table - one for true path predictions and one for spurious predictions.

In Figure V.7(d) we we show how APPEND produces two distinct addresses into the global pattern history table for two occurrences of a branch: one when the branches guarding predicate is true and one when it is false. First we show the index into the global pattern history table produced when the guarding predicate of branch `(b>c)` is true, and how when the value of the guarding predicate is false, we index into a different location in the global pattern history table. This should allow a separate 2-bit predictor each for each of the two possible states (true path and false path) of branch `(b>c)`. Again, this scheme only benefits when the definition of `P2` is scheduled early enough that the predicate register file can be updated in time for branch `(b>c)` to use it in its prediction.

Figure V.9 shows the structure of the APPEND predicate update global branch predictor as incorporated as part of a Meta Chooser Predictor. A Meta Chooser branch prediction with APPEND also takes two serial table lookups. First we need to read the guarding predicate associated with the branch from the branch target buffer. Then another lookup is made in the predicate register file to find the currently available predicate value. In parallel, we access the global history register with two different values. One is the current value of the global history register appended with a `0`, the second is the current value of the global history register appended with a `1`. This will produce predictions for a "false path" occurrence and "true path" occurrence of the branch, respectively. After the current value of the guarding predicate has been read from the register file, it is used to select between the two predictions produced from the global pattern history table. Note that in this process the value of the global history register is *not* modified with the value appended onto it. This value is only used in creating the

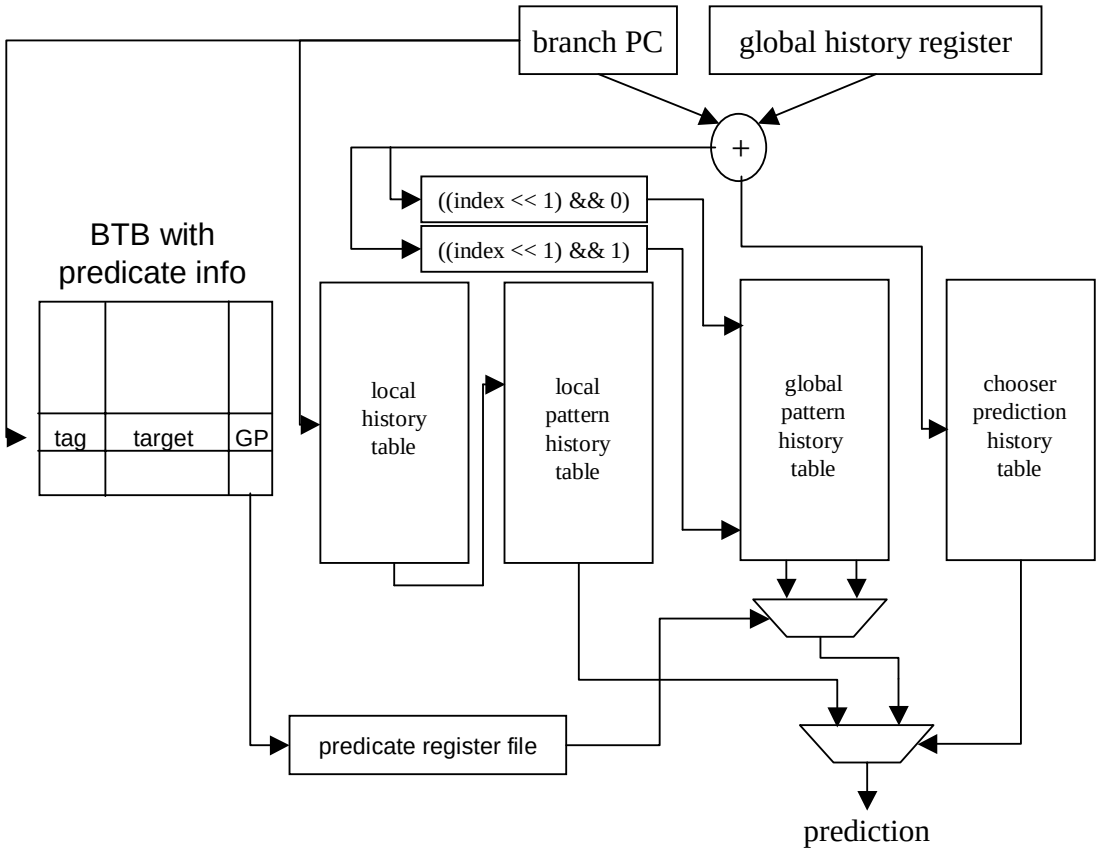


Figure V.9: A schematic of the APPEND global history predicate-aware predictor. APPEND separates out true and false path predictions by appending the global history register with the value of the branch's guarding predicate register. This scheme appends the global history register with both `0` and `1` and indexes into two places in the global pattern history table, producing two predictions. The value of the branch's guarding predicate as read from the predicate register file is used to select one of these predictions. Finally, as part of a Meta Chooser predictor, the chooser table entry indicates whether the local or global prediction should be used. APPEND requires a serial two table lookup to retrieve the branch's guarding predicate value.

access index into the global and chooser prediction tables. The global history is recovered on a misprediction using the traditional SHQ mechanism as no predicate define information is kept in the global history register.

V.D.3 The Impact of PEP and APPEND

In Figure V.10 we show the abilities of PEP and APPEND to utilize predicate define information to reduce branch mispredictions for predicated codes. It is important to note that PEP and APPEND are complementary techniques. PEP utilizes predicate information in a *local* prediction scheme. APPEND utilizes predicate information in a *global* prediction scheme. Modern branch prediction architectures use both local and global prediction schemes because each scheme is beneficial for certain classes of branch behavior.

To illustrate this point, the first three bars for each benchmark in Figure V.10 show the misprediction rates achieved on the original, non-predicated code for these benchmarks. The first bar shows the misprediction rate using only a local history prediction scheme, the second shows the rate with only a global history prediction scheme, and the third shows the rate with a Meta Chooser architecture utilizing both local and global history schemes. For some benchmarks a local history prediction scheme does better than a global history prediction scheme, and sometimes the opposite is true. All of the benchmarks benefit from a Meta Chooser branch prediction scheme where branches can dynamically select to predict using a global or a local scheme based on previous history of recent branches.

The fourth bar of each graph is the first to show results from execution of predicated code. These results are for our Baseline Meta Chooser predictor. It does not utilize predicate information in any way. Here (as for all misprediction rates shown for predicated codes) the miss rate is calculated as the number of prediction misses in the predicated code divided by the number of branch predictions in the original, non-predicated code. This provides a constant metric for evaluation as the number of predictions made in the predicated code differs from the number of predictions in the original code.

The fifth, sixth, and seventh bars show the benefits of utilizing predicate define information in predicting branches for predicated codes. The fifth bar shows the PEP method of predicate update predictions for a local scheme and the sixth bar shows the APPEND method of predicate update predictions for a global scheme. Both runs utilize the baseline Meta Chooser predictor with predicate define information available in either the local predictor (PEP) or the global predictor (APPEND).

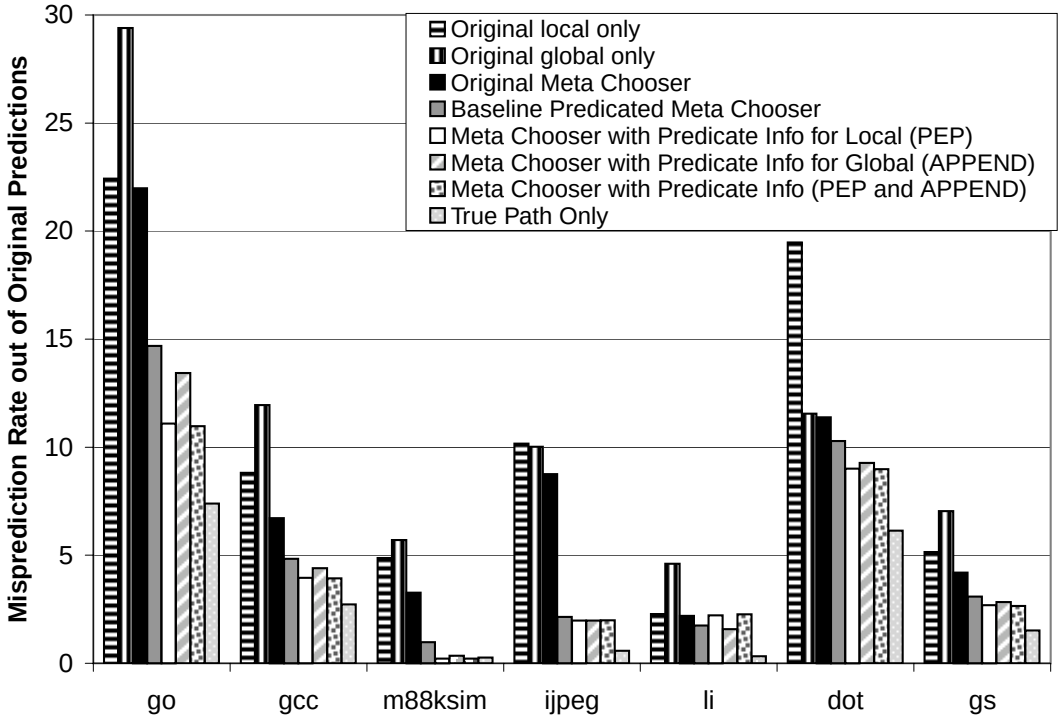


Figure V.10: The impact of predicate region formation on the misprediction rate over a set of benchmarks. The first three bars show the mispredict rate of the original, non-predicated code. The first bar shows the results of using a local prediction scheme, the second a global prediction scheme, and the third a Meta Chooser scheme using both local and global predictors. The remaining bars show the mispredict rate of the predicated code with varying branch prediction architectures - all based on the Meta Chooser. The first is a Meta Chooser predictor that uses no predicate information. The second incorporates predicate information in a local prediction scheme (PEP) and the third incorporates predicates in a global prediction scheme (APPEND). The fourth combines PEP and APPEND to utilize predicate information in both parts of the Meta Chooser. The last bar shows an idealized execution of predicated code where only true path branches have to be predicted (using a baseline Meta Chooser architecture). The mispredict rates of the predicated codes are calculated based on the number of predictions from the original, non-predicated code as that provides a constant metric for comparison.

The seventh bar shows a predicate update Meta Chooser with predicate information available in both the local and the global schemes (PEP and APPEND). Though this predicate update solution is a notable improvement over the baseline Meta Chooser architecture in predicting predicated codes, there are still excess mispredictions as a result of spurious branch execution. We show this is the case by providing the mispredict rate gathered from an idealized execution of predicated code in the last bar for each benchmark (True Path Only). In this execution, we know *in fetch* which branches are true path branches and which are spurious. For spurious branches, we don't predict. The "True Path Only" mispredict rates show that the cost of spurious branch prediction in predicated codes is significant, even after application of predicate update techniques. On average across our benchmarks, our PEP and APPEND predicate update techniques have a 1.7% higher mispredict rate that a predictor which does not have to predict spurious branches at all.

V.D.4 More Accurate Guarding Predicate Information for PEP and APPEND

The direct read of the predicate register file for predicate information as used in the previous sections' PEP and APPEND predicate-aware predictors can be potentially performance-hindering. In cases where the predicate define guarding a branch has been issued to the pipeline, but not yet resolved, one may read an invalid value for that predicate define value. The value in the register may be from a define in a different portion of the code, or, in the case where more than one instruction can define a guarding predicate (via cmp.or or cmp.and), not the final value that the guarding predicate will receive. This will only affect those predictions made for branches whose guarding predicate define has not yet resolved.

In either case, perfectly accurate information about the state of the guarding predicate register file and any outstanding (in the pipeline) defines of the predicate is available. We rely upon a modified register lookup to tell us if the latest definition has written to the register, or whether an instruction in the pipeline has yet to produce its value. This information is needed in traditional pipelined architectures to determine if a bypassed value from the pipeline should be used instead of the register file value when executing an instruction.

The effect of implementing a more accurate predicate value read in both the PEP and APPEND predicate update branch prediction schemes is shown in Figure V.11. On average this additional predicate accuracy only leads to 0.16% improvement for a PEP and APPEND Meta Chooser predictor. However, having completely accurate guarding predicate define information

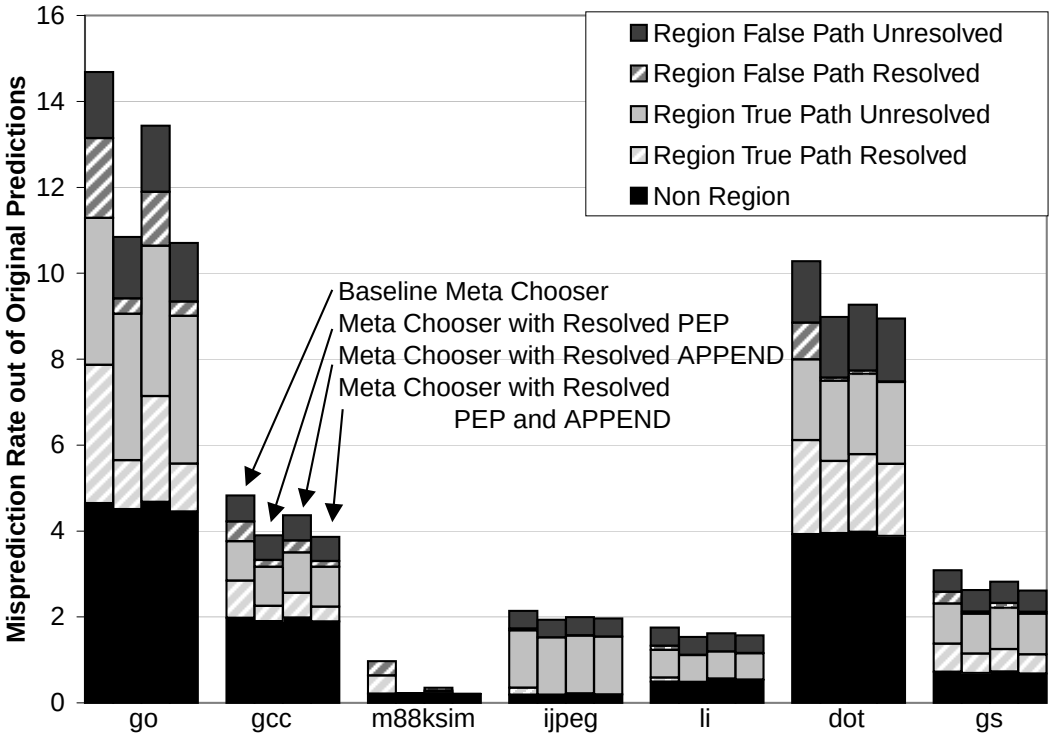


Figure V.11: The impact of accurate predicate information on the ability of PEP and APPEND to reduce mispredictions via spurious branch identification. Branch mispredictions are classified into five categories (from bottom to top): those from branches not in predicated regions, those from true path region branches whose guarding predicate is resolved in fetch, those from true path branches whose guarding predicate is *not* resolved in fetch, those from false path branches whose guarding predicate is resolved in fetch, and those from false path branches whose guarding predicate is *not* resolved in fetch. The mispredict rates of the predicated codes are calculated based on the number of predictions from the original, non-predicated code as that provides a constant metric for comparison.

allows us to categorize the source of branch mispredictions in predicated regions. In Figure V.11 we break down the branch prediction misses stemming from predicated regions into the following categories based on the state of the information available about the branch's guarding predicate:

- Resolved True Path: The definition of the branch's guarding predicate has finished execution and the value of the guarding predicate is true.
- Resolved False Path: The definition of the branch's guarding predicate has finished execution and the value of the guarding predicate is false. This is a *resolved spurious branch*.
- Unresolved True Path: The definition of the branch's guarding predicate has not finished execution at the time a branch guarded on it needs to be predicted. The value of the guarding predicate eventually (by the time the branch reaches the end of the pipeline) will be true.
- Resolved False Path: The definition of the branch's guarding predicate has not finished execution at the time a branch guarded on it needs to be predicted. The value of the guarding predicate eventually (by the time the branch reaches the end of the pipeline) will be false. This is an *unresolved spurious branch*.

We can see that both PEP and APPEND reduce the number of mispredictions from resolved spurious branches *and* from resolved true path branches. This benefit for true path branches accrues from reduced conflict with spurious branch predictions. However, we note that resolved spurious branches still contribute a significant number of misses. In the next section we examine a predicate-aware resolved spurious branch filter scheme that can be used in conjunction with any branch prediction scheme to eliminate the mispredictions from resolved spurious branches.

V.D.5 Squash-FP: Tackling Spurious Branches Directly

Although the PEP and APPEND predicate update predictors attempt to identify spurious branches, they still require a predictor table entry to predict them. But the prediction for a spurious branch can be made 100% accurate without the aid of dynamic branch prediction. A spurious branch should predict not taken. We propose the Squash-FP branch prediction filter technique which can be applied in conjunction with any branch prediction scheme. This scheme squashes predictions for resolved spurious branches and, instead, predicts that they are not taken. This has the double benefit of achieving 100% prediction rates for resolved

spurious branches and reducing utilization of traditional branch prediction resources. For the Meta Chooser branch prediction scheme we evaluate, this has the added benefit of reducing contention in the branch prediction tables.

The Squash-FP filter technique is designed to directly reduce the mispredictions caused by the spurious execution of branch instructions in predicated regions. It is titled a filter technique, since it does not replace traditional branch prediction architectures, but rather, removes some branches from the consideration of traditional branch predication architectures. Specifically, the Squash-FP filter attempts to determine is a branch is a *spurious branch*. This determination is made in branch fetch if the last predicate define of the branch's guarding predicate has completed execution *and* the value of the predicate register is `0`. If this is the case, we say the branch is *resolved spurious*. Resolved spurious branches have the characteristic that they should never affect control flow. Resolved spurious branches should always predict "not taken".

In Figure V.12, we show a schematic of the Squash-FP filter. In addition to feeding the traditional branch prediction architecture, the branch PC is fed to the BTB which contains the predicted target address of the branch instruction. The BTB is normally indexed for a branch to produce the branch target, but Squash-FP requires additional bits in each BTB entry to store the guarding predicate register number for each branch. That number is used to index into the predicate register file to read the current value of the branch's guarding predicate. Additionally, the register number is used to query pipeline structures which indicate whether any outstanding definitions of the predicate have not yet finished execution. If the branch's guarding predicate is `false` and the last definition of the predicate has completed, then we squash the traditional branch prediction architecture's prediction with a prediction of "not taken". Additionally, we don't update the traditional branch prediction tables for this branch. The Squash-FP filter requires a two table lookup to produce a predicate-aware prediction. If a two table lookup cannot be implemented in a single cycle, Squash-FP may be used to redirect a prediction made in the first cycle when it identifies a resolved spurious branch.

In Figure V.13 we show the results of applying the Squash-FP filter technique for resolved spurious branches to the following branch prediction architectures: Baseline Meta Chooser, PEP, APPEND, and PEP and APPEND. Using the breakdown of misprediction origins, we can clearly see the effect the Squash-FP filter has by removing any miss stemming from a resolved spurious branch.

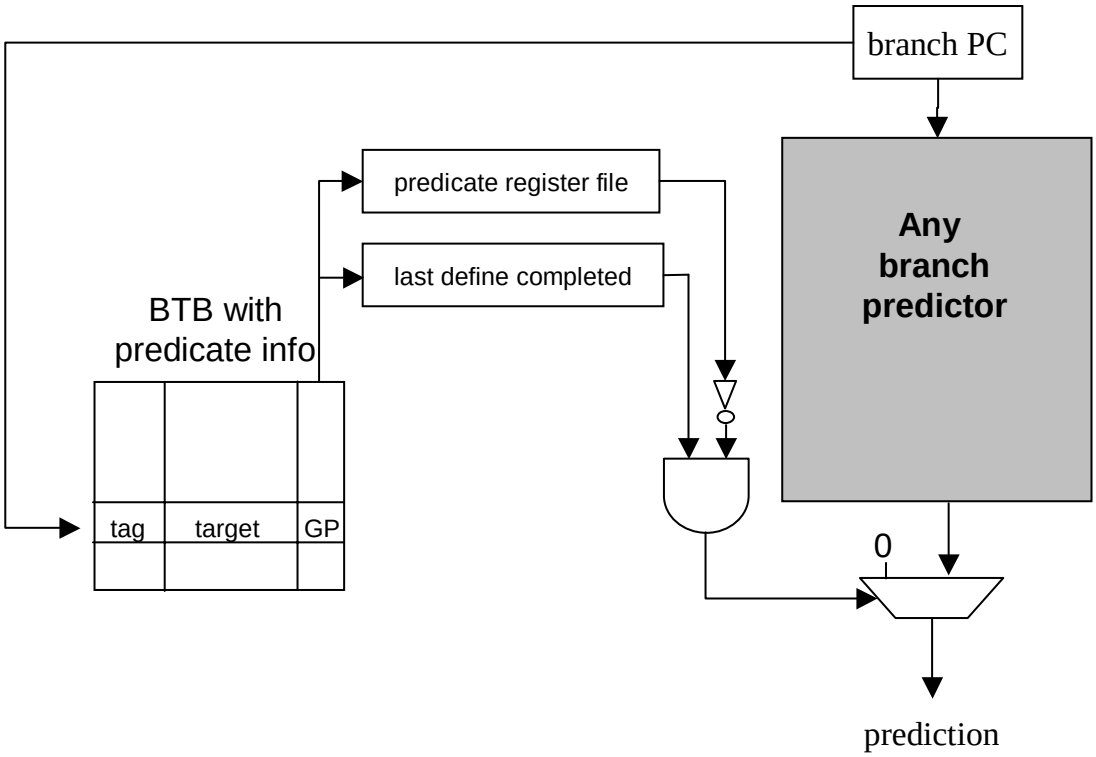


Figure V.12: A schematic of the Squash-FP predicate-aware branch prediction filter. Squash-FP requires additional information to be stored in the BTB regarding the guarding predicate for each branch. That predicate register number is used to query the predicate register file and the processor pipeline bypass information to determine if: a) the predicate register is false and b) the last predicate define of that register has completed. If both those conditions are met, then the branch is *resolved spurious* and we override the prediction from the branch predictor with a prediction of `0` or `not taken`. A serial two table lookup path is required for a Squash-FP predicate-aware prediction.

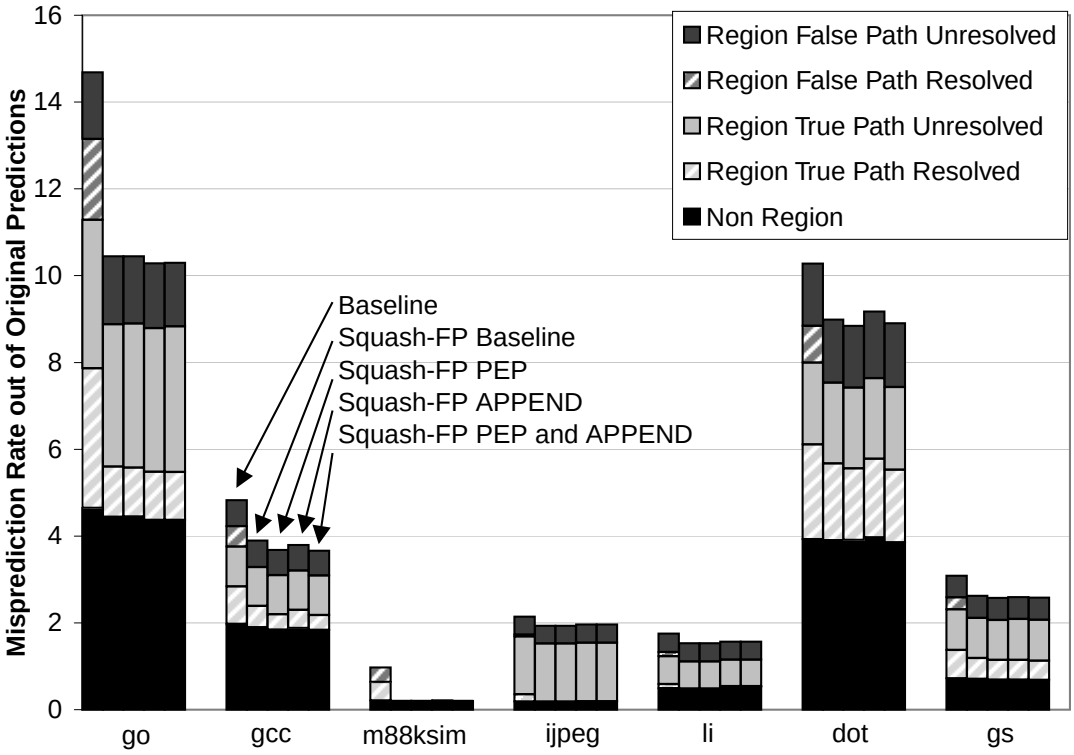


Figure V.13: The ability of the Squash-FP filter to reduce mispredictions by filtering out and predicting resolved false path branches as not taken. The Squash-FP filter can be applied on top of traditional branch prediction schemes. Here we show it filtering a baseline Meta Chooser predictor, a Meta Chooser using predicate information with PEP, a Meta Chooser using predicate information with APPEND, and a Meta Chooser using both PEP and APPEND. Note that all Squash-FP filter-enhanced schemes have no misses from resolved false path branches.

V.D.6 Better Scheduling for Additional Benefit from PEP, APPEND, and Squash-FP

For some benchmarks we examined, such as `ijpeg` and `li`, there is a relatively low percentage of mispredictions from *resolved* spurious branches. These are not able to be targeted by the Squash-FP filter because the value of the branch's guarding predicate register is not known in time for the branch to be predicted. The classification of whether a branch's guarding predicate value is resolved or not is primarily contingent on the scheduling of the code. While in our experiments we made an attempt to schedule branches as far from predicate defines as possible (see section V.C.2), our attempts to do so were limited by lack of compiler support. An aggressive optimizing compiler would both be able to schedule branches and predicate defines further apart in order to allow guarding predicate defines to resolve, but also be able to schedule branches "just" far enough away from the guarding predicate define on which is is contingent. Our current method is both occasionally ineffective in scheduling enough intervening work and often overzealous in scheduling too much intervening work. Both characteristics are detrimental to overall behavior and could be improved upon by an optimizing compiler. The compiler would need to know architectural pipeline characteristics in order to schedule the "delay" between predicate defines and branches appropriately. Such techniques would benefit not only Squash-FP, but also PEP and APPEND which rely on a branch's guarding predicate define completing execution by the time the branch is fetched.

V.D.7 A global history predictor filtered by Squash-FP

Although we do not elaborate on the interaction of the Squash-FP filter with any generalized branch prediction architecture, we make a specific suggestion for global prediction schemes which utilize a global history register - the global history register should be updated with the results of the predictions made via the Squash-FP filter.

While one of the benefits of the Squash-FP filter is that it reduces the dynamic working set of branches for a general branch prediction scheme, schemes which rely on reproducible strings of history to produce accurate predictions can be negatively affected by Squash-FP. This is because whether or not a given branch is *resolved* spurious or *unresolved* spurious can differ with dynamic execution characteristics. Hence, some spurious predictions are made via Squash-FP (when resolved) and some are made with the branch prediction architecture. Moreover, in the case of a global history register, further branches in the pipeline may utilize

information about the spuriousness of this branch to make correlative predictions. Just because the spuriousness was identified via the Squash-FP filter does not mean that we should remove knowledge of this branch's spuriousness from the global history register. To do so may remove valuable global correlation information for later branches. To avoid this problem the global history register should be updated with predictions made by the Squash-FP filter.

V.E Gathering Predicate Define Information Dynamically

One potential pitfall of the predicate update prediction architectures developed in section V.D is that they rely on specific branch/guarding predicate relationship information to take advantage of predicate define values. This has two effects. First, it means that whether a branch can be predicted in light of predicate information is completely reliant on whether its particular guarding predicate has finished execution by the time the branch is fetched. Second, it means that making a predicate-aware branch prediction requires a two-table lookup process: first one must look up the branch's guarding predicate register number; then the predicate register file must be queried for the current value of that register. Next we introduce the Predicate Global Correlation Update predictor (PGCU) which addresses both of these restrictions.

V.E.1 PGCU for Dynamic Predicate Correlation

While it may be that a given guarding predicate of a branch may not be resolved, a different predicate on the path of execution leading to that given predicate define may have finished execution. In the terminology presented in Chapter IV we would call this a partial path predicate. A partial path predicate is a predicate which contributes, in part, to the definition of the predicate in question. We would like to be able utilize partial path predicate information in determining whether a branch is spurious or not.

Consider Figure V.14 where if predicate `P5` is not defined by the time `branch if P5` needs to be predicted schemes using direct branch/guarding predicate define relationship information would not benefit. However, we can see that a false value for `P3` will eventually cause a false definition of `P5` - meaning that `branch if P5` will definitely be spurious. It would be preferable for cases like Figure V.14 for partial predicate define information to be used in the prediction of a branch in case the full guarding predicate has not been able to resolve before the branch guarded on it needs to be predicted.

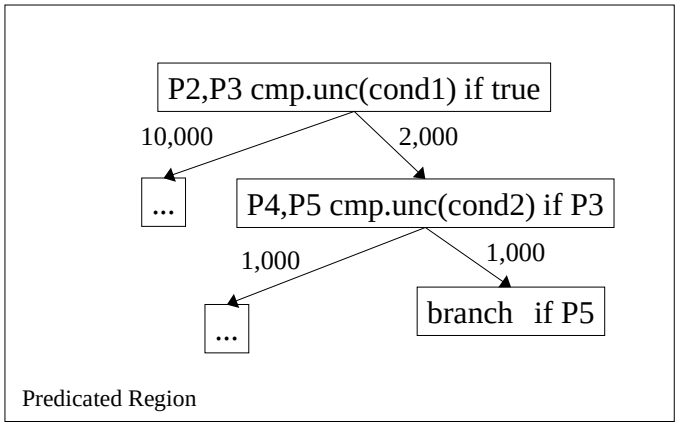


Figure V.14: An example where partial predicate relationship information can alleviate misprediction migration. In this example if `p4,p5 cmp.unc(cond2) if P3` cannot finish execution by the time `branch if P5` needs to be predicted, PEP cannot benefit. However, the information from the first predicate definition would be valuable in predicting `branch if P5`.

Our desire to take advantage of dynamic partial predicate relationship information mirrors closely the intention of a global history prediction scheme to utilize dynamic relationship information among branches to identify *global correlation*. While branch global correlation exists in architectures which don't support predicated execution, an additional form of global correlation is possible in an architecture which supports the predicate-guarded execution of branches. That is, there may be correlations between partial path predicate defines and the guarding predicate which controls the execution of a branch.

Besides exposing partial path predicate correlation opportunities, we also desire to expose global correlation opportunities along the actual path of execution. As shown in Figure V.4, the process of predication can remove important global correlation information from the branch prediction architecture. We would like all control flow information to update the branch prediction hardware regardless of the form in which is is expressed: branch or predicate define.

We propose the Predicate Global Correlation Update Predictor (PGCU) which allows predicate define instructions to update the global history register of a global branch prediction architecture to gather predicate-based global correlation information. This scheme will dynamically gather branch/guarding predicate relationship information without requiring either a) specific branch/guarding predicate register number information to be stored in the hardware or b) a predicate register file lookup of the value of a specific register. Guarding predicate information is stored directly in the global history register in much the same way as branch

information.

The PGCU predictor architecture stores the predicate result from predicate define statements in the speculative global history. When a predicate define instruction finishes its execution, it shifts the result of its condition evaluation into the lowest bit of the global history register in exactly the same way the result of a branch is shifted into the global history register. This update process will be outlined in detail in section V.E.3. This update methodology distinguishes the PGCU architecture from the others presented here in that it can produce a predicate-aware prediction in a single table lookup. This is because there is no need to obtain a predicate register number associated with a particular branch and then read the predicate register file. Predicate information is directly incorporated into the global history register, which is then used to directly index the global pattern history table for a prediction.

However, one important issue arises regarding the timing of the update to the global history register. In traditional architectures, the global history register is updated in fetch, when a prediction for the branch is made. However, predicate define instructions are not predicted. Therefore there is no value available to allow them to update in fetch. Instead, predicate define instructions will update the global history register after they complete execution in the pipeline. Since the predicate define instructions perform a *delayed update* of the speculative global history register, the ordering of the global history information will be different than in the original non-predicated program.

Additionally, for a branch to benefit from predicate information stored in the history register, the predicate update has to occur before that branch is fetched and predicted. This can be addressed by compiler scheduling of the predicate defines as early as possible in the predicated region combined with scheduling the region branches as late as possible (see section V.C.2. This comes at the cost of executing additional non-true path instructions, since we exit the region later when a branch is taken out of the region.

Region branches (both spurious and true-path) update the PGCU predictor in exactly the same fashion as the Baseline Predicated predictor. Both are predicted and speculatively update the global history register in fetch.

V.E.2 A Deterministic Predicate Update Architecture

A delayed update of the global history register results in very accurate predictions as long as branch/predicate define ordering is identical throughout execution. However, this is not guaranteed with traditional pipelined execution.

This problem stems from the fact that static architectural pipeline information does not always give accurate information regarding the time it will take for a predicate define to finish execution. Between the time a predicate define is fetched and has completed execution, the define may stall in the pipeline due to system-level effects such as a cache miss. For a modern system architecture, in these cases fetch (and branch prediction) continues, filling up buffers that will be emptied when the pipeline stall finishes. For PGCU, dynamic instances of stalls in predicate define execution while branch prediction continues (in fetch) means that the ordering of information in the global history register may change. In Figure V.16 we show a delayed update of the global history register by predicate defines. However, this example shows the predicate defines updating after a fixed-delay of 20 instructions. When no pipeline stalls occur, the `cmp1` and `cmp2` compares update the global history register after the branch `br3` but before branch `br4` - resulting in a global history register that holds results from `br3 cmp1 cmp2 br4`. In cases where a pipeline stall occurs after `cmp1`, the update by the predicate define `cmp2` will be delayed, possibly leading to a register that holds results from `br3 cmp1 br4 cmp2`. Since the global history register relies on a reproducible pattern of history information to make predictions, the potential variance in the delay of the update of predicate defines by PGCU could be a serious problem. While this effect might be relatively small in an in-order processor like the Intel Itanium, the issue will certainly be more noticeable in an out-of-order processor implementation.

We implement a deterministic predicate update scheme in the processor architecture to solve this problem. Figure V.15 shows a schematic of our structure. The goal of this modification is to fix a delay time for each static predicate define. In the deterministic update structure we store the following information for each predicate define as it is fetched: predicate define identifier (address), cycle count delay, and result value. The result value holds the result of the predicate define instruction. Before the instruction has completed execution, this is set to a default value of "false" or "not taken". This is the value we use to update the branch predictor's global history register. The cycle count delay determines when the value stored in result will update the global history register. This cycle count is initialized to a particular value for each predicate define when the instruction is fetched. Every cycle thereafter, the counter is decremented, until it reaches zero, when the value in result updates the global history register. This guarantees fixed update delay for predicate defines and avoids the problems that arise with a changing update order in the global history register.

In Figure V.15 we show how the deterministic update structure performs when pro-

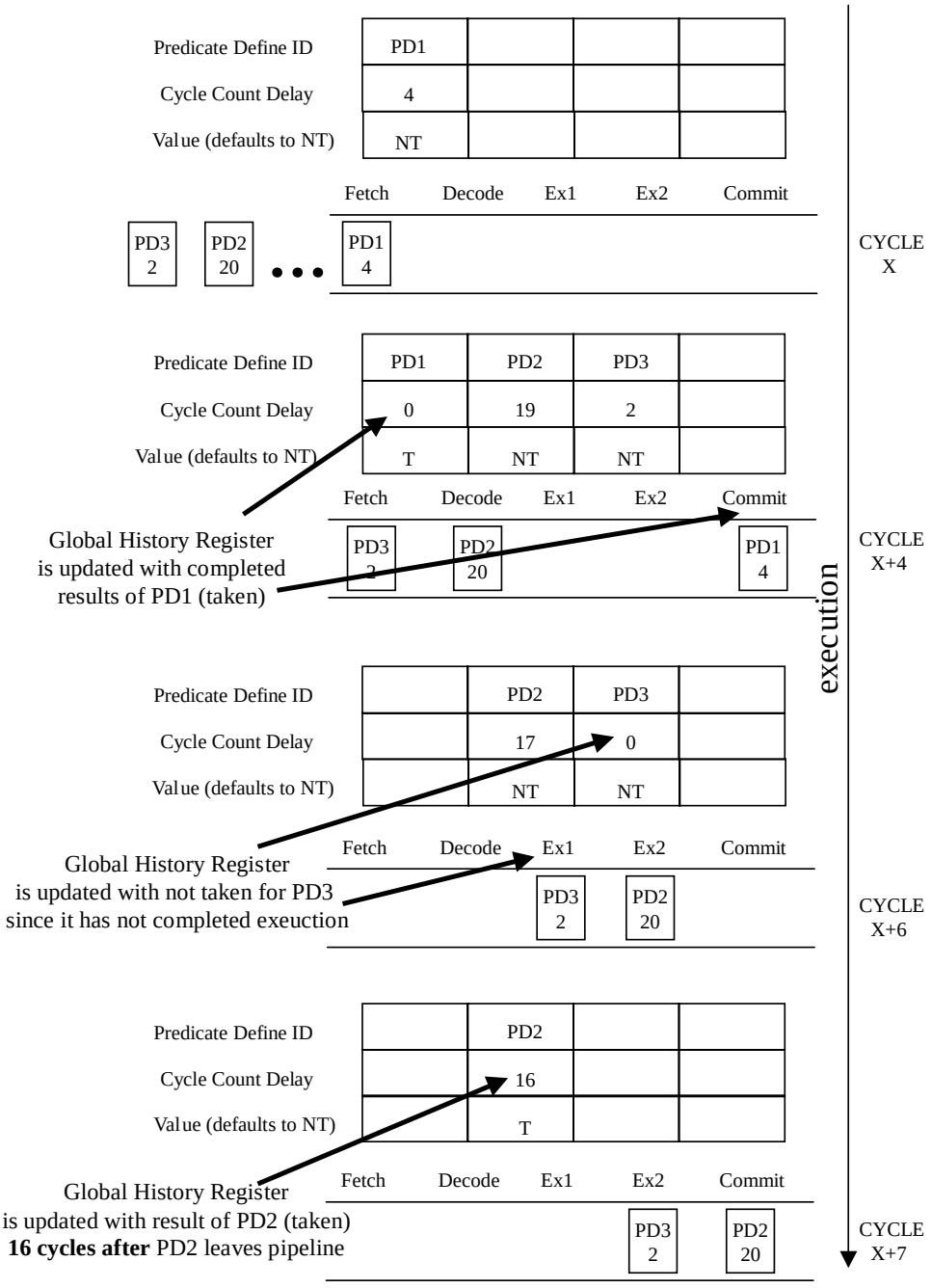


Figure V.15: An example of how a deterministic update structure functions to ensure a constant delay update of the global history register by predicate defines. Clearly, the delay selected for each predicate define is key. If the delay is too short, then the predicate define will not have enough time finish execution before it updates the global history register. In those cases not taken is entered into the global history register. In the case of `PD1` we show that sufficient time is allowed for the define finish execution. However, a longer delay (like that of `PD2`) will be more resilient to dynamic delays (such as cache misses) and will provide valid update information more consistently than `PD1`.

cessing three different predicate define instructions. In (a) the first predicate define, PD1, is fetched and inserted into the deterministic update structure with a delay of four cycles. Next PD2 is inserted with a delay of 20 cycles. Then, in cycle X+4 PD3 is inserted with a delay of 2 cycles *and* the cycle counter for PD1 has been decremented to zero. At this point, the global history register is updated with the value stored in PD1's value entry - which was updated after PD1 finished execution with the result of taken. After two more cycles have elapsed, the cycle count for PD3 reaches zero and it updates the global history register with the value in the deterministic update structure. As PD3 has not finished execution in the pipeline, the update to the global history register will be the default of not taken. There is no other update of the global history register made when PD3 finishes execution, the only update is made when the delay for the predicate define passes in the deterministic update structure. In this case, we may have entered invalid data for PD3. Finally, even though PD2 commits in the pipeline in cycle X+7, it will not update the global history register until cycle x+23 - 16 cycles after it leaves the pipeline. The result of its execution is stored in the deterministic update structure's value entry for PD2 and is used in updating the global history register.

Clearly, the setting of a predicate define's cycle count delay will be crucial to achieving benefit from predicate information. There are several options for setting the cycle count that delays the update of a given predicate define. The simplest, which requires no compiler intervention, is to set all predicate define delays to the minimum latency (in terms of pipeline stages) from fetch to the end of execution. This will not take into account any dynamic delays due to effects like cache misses that may stall a predicate define in execution. Another option is to set the cycle count according to the scheduled code. If the compiler was able to schedule 15 cycles between a predicate define and a branch guarded by it, then that predicate define will update after 15 cycles - just in time for the information it provides to be used. This scheme will break down in the cases where the compiler fails to provide a minimum latency between a predicate define and a branch guarded by it. This could be alleviated by selecting the maximum delay from the scheduling-based technique and the pipeline-stage-based technique. A third option is to use dynamic profiling information to create a cycle count for each branch - which will help take into account dynamic pipeline effects. This is the technique we use for the results presented in this chapter. We gather dynamic information regarding the actual delay between fetch and the end of execution for each predicate define in the code. This information is gathered in SimpleScalar, which provides detailed pipeline timing information. However, since SimpleScalar does not support the execution of our predicated code, our delays do not

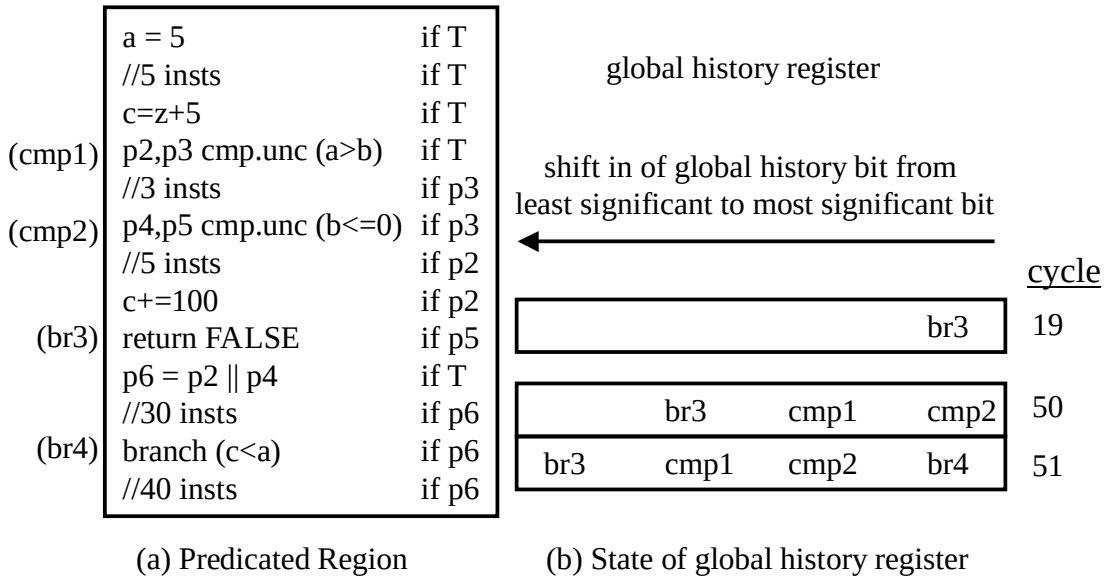


Figure V.16: Update of the Meta Chooser global history register when enabled with predicate update. Note that branches update the global history register in fetch, while predicate defines update in writeback. This causes a "re-ordering" of information in the global history register as compared with instruction fetch ordering. For the purposes of this example, we show a fixed 20 instruction delay from predicate define fetch to resolve. In our results updates occur based upon the resolution latency for individual predicate define instructions.

match the schedules we form for predicated regions. Instead, we gather the delays from the original branches in the original code, as simulated on an 8-issue out-of-order processor pipeline as described in section V.C.3. We keep an average of these delays per branch and use that data to set the cycle count delays for our predicate defines.

V.E.3 Speculative Global History Update

Even with a guaranteed fixed global history update ordering of branch and predicate defines as enabled by the deterministic update structure, the delay of predicate define information in updating has an impact on our ability to accurately predict branches. Figure V.16 shows an example of how the predicate update predictor would update the global history register. For this example, we assume a 20 instruction delay from the fetch of a predicate define until it resolves in the writeback stage where we update the speculative global history register. The actual update would occur at varying times for different predicate define instructions depending on their latency from fetch to writeback.

Using the PGCU predictor, the speculative global history register is updated with

the values of predicate registers p2 (cmp1) and p4 (cmp2). The predicate defines formed via our region formation process all define two complementary predicates, and we model the architecture such that it updates the speculative global history register with the value of the first predicate. The IA-64 architecture supports a variety of predicate define instructions that can define up to two predicates using many different boolean combinations [5]. Examining how to use other IA-64 predicate define instructions and their interaction with the predicate update predictor is an area for future work.

As can be seen in Figure V.16, the PGCU predictor achieves its goal of incorporating the important history of statement (a>b) into the speculative global history register. When (c<a) is fetched, the second most recent bit of history in the global history register (cmp1) helps determine the correct prediction for (c<a). This alleviates the problem of misprediction migration that would otherwise manifest for this branch without a predicate update branch predictor.

Using the schedule in Figure V.16, the region branch return FALSE is not able to benefit from the predicate information in the global history, since global history from neither of the cmps is updated in time. A possible solution to this issue is to make predicate region code scheduling aware of the correlative behaviors between predicate defines and branches. A conservative solution to the problem is to schedule predicate defines as early as possible in a region while also scheduling branches as late as possible in the region - which is the scheduling policy we follow in our simulations. This allows, as much as possible, the update of the global history register by the predicate define to complete before we fetch and predict the branches that are correlated with it. The optimal schedule would not move predicate defines as far away from branches as possible, but rather, just far enough to allow the predicate defines to updates as was discussed in section V.D.6.

In Figure V.17 we present results using the PGCU predictor to produce predicate-aware predictions for a global history prediction scheme. We see from comparing the second and third columns, that PGCU is beneficial over Squash-FP for li and ijpeg. This stems from the fact that very few misses in these two benchmarks come from resolved branches (either true or false path) in the Baseline Meta Chooser architecture (first column). For these cases, Squash-FP doesn't filter many predictions, but PGCU can take advantage of correlation based on predicate values, even when a branch's guarding predicate has not resolved by branch fetch. However, for any benchmark with more than a trace number of branches that can be known as spurious, Squash-FP is much more effective than PGCU in reducing mispredictions.

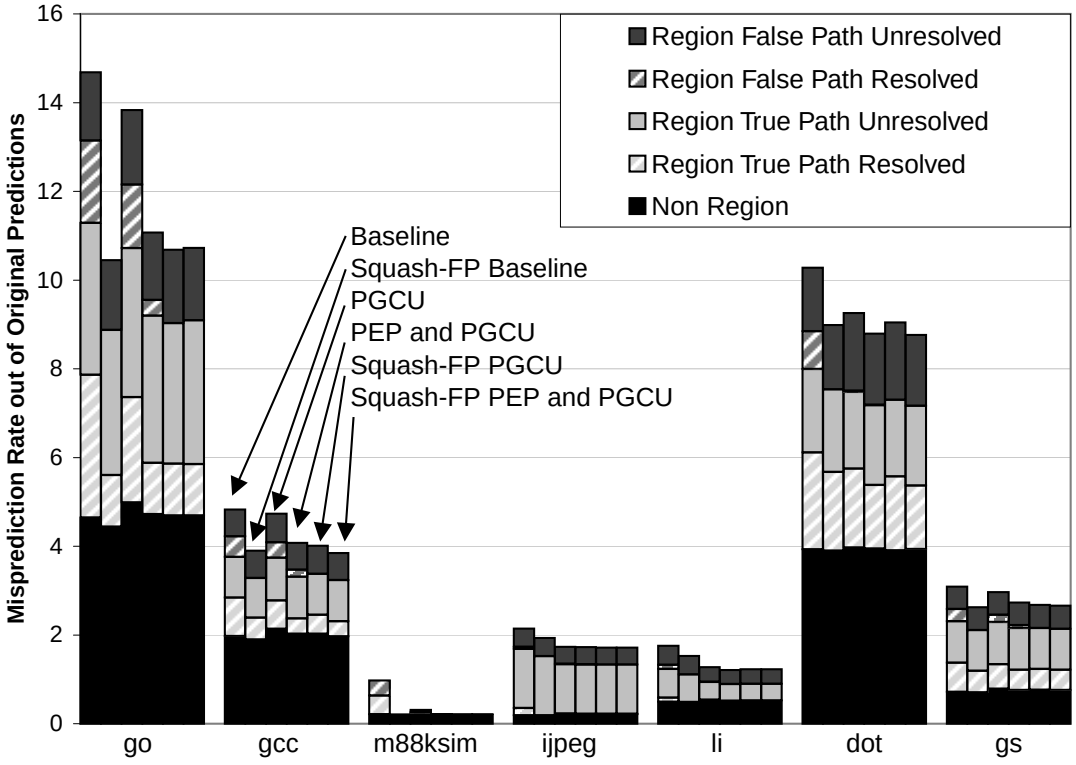


Figure V.17: The benefit of using a PGCU predicate-aware global prediction scheme. PGCU can also be augmented with a Squash-FP filter and combined with a PEP predicate-aware local predictor. While results here show misprediction rate only, it is important to note that PGCU can produce a predicate-aware prediction with a single table lookup, where both PEP and Squash-FP require a two table lookup.

Since PGCU can be complemented by a predicate-aware local history scheme, we also show the benefit of combining it with the PEP predictor. Finally, because any predictor can augmented with the Squash-FP filter, we show the impact of Squash-FP with PGCU and Squash-FP with PEP and PGCU as well.

V.E.4 PGCU and Global History Register Recovery on a Branch Misprediction

An important topic in branch prediction is recovering the prediction history state after a branch misprediction. Since we are updating the global history register with branches in the fetch stage and predicate define instructions in the commit stage, for this architecture to work it will be key that the updates to the speculative global history register occur in a consistent order for a given trace through the program's execution. In addition, we need to be able to correctly recover the speculative global history register in the case of a branch misprediction.

The traditional implementation of a branch prediction architecture maintains speculative global history information and non-speculative global history information. The non-speculative history is accessed when there is a misprediction and is used to restore the speculative history to the state that was valid for the branch that was just mispredicted.

Our architecture uses a Speculative History Queue (SHQ) [36, 41] to hold the state of the global history register, so it can be restored on a misprediction. The SHQ stores the full speculative global history register into a queue each time a branch is predicted. When a branch is mispredicted the global history register for that branch is restored from the SHQ. The last bit in the history register representing the mispredicted branch is inverted, and this becomes the new speculative global history register used for prediction for the next fetch. For example, if branch `br4` mispredicts in Figure V.16, the speculative global history register will be restored to {`br3, cmp1, cmp2, !br4`}, correctly keeping track of `cmp1` and `cmp2`.

In using the SHQ architecture as defined in [36, 41], there is a very small window in which a predicate define may *not* make its way into the SHQ, so that it can be restored after a misprediction. If the predicate definition resolves between the time a mispredicted branch is fetched and the time the branch triggers a misprediction, it will make its way into the speculative global history register, but not into the SHQ. This is because only a branch prediction will insert the speculative GHR into the SHQ. In this situation, the predicate define information will be lost if there is a misprediction before the next branch is fetched. We modeled this in our simulations.

V.E.5 Global Correlation Revisited

In Section V.A.2 we evaluated the potential for improved predictor performance via re-insertion of global correlation information when only true path branches were executed. However, because global correlation is identified via the global history register and because spurious execution has the pitfall of overloading the global history register by making entries for false path branches, it seems likely that spurious execution may have a serious impact on the ability to maintain beneficial global correlation information in the global history register. This is shown to be the case in the results from Figure V.17, even when Squash-FP is employed to reduce the impact of spurious execution as much as possible.

In comparing the results from Figure V.5 we expect to see the most benefit from PGCU on the benchmarks `go`, `gcc`, and `dot`. But, the impact of spurious branch execution so overshadows the benefit to be gained from original global correlation as to make it impossible to recapture. All three of these benchmarks do better with the Squash-FP filtered baseline predictor which efficiently removes their significant number of mispredicts from resolved spurious branches. Employing a Squash-FP filter on top of a PGCU predictor shows benefit, but there is still sufficient interference from non-filtered false path branches to inhibit benefit from global correlation.

However, PGCU clearly has the significant benefits for `li` and `ijpeg`. This is the direct result of the fact that both of these benchmarks suffer from very low rates of resolved branches. Figure V.18 shows a categorization of the *predictions* of region branches across the benchmarks. We show the number of region branches whose guarding predicates are resolved by the time the branch is fetched (bottom solid bars) relative to the number whose guarding predicates are unresolved (top dotted bars). Additionally, we show the breakdown between true and false path for each category.

In Figure V.18 we see that only 24% of region branch predictions in `ijpeg` and only 7% of region branches in `li` have resolved guarding predicates in fetch. This greatly reduces the ability of Squash-FP (or PEP or APPEND) to perform predicate-aware predictions for these benchmarks. PGCU, however, sees benefit in these cases by capturing a new form of global correlation exposed via multiple-path execution of predicated regions. For those benchmarks where few branch guarding predicates can be resolved in fetch, PGCU allows predicate-based correlative information to be exposed for use in predictions.

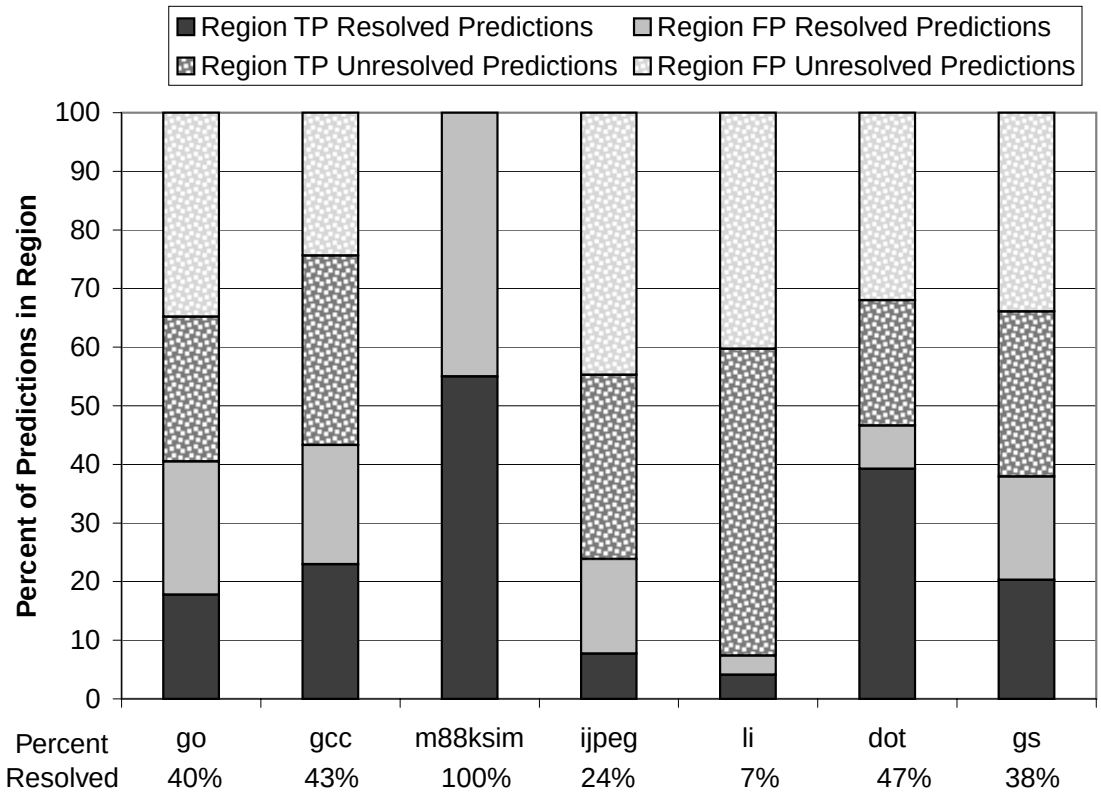


Figure V.18: The breakdown of branch predictions from predicated regions. We categorize each dynamic branch prediction based on whether it is on a true or false path of execution (i.e. its guarding predicate will be true or false) and by whether that information is known by the time the branch is fetched (resolved) or not (unresolved). Both `ijpeg` and `li` experience a very low percentage of branches with resolved guarding predicates in fetch. These benchmarks benefit from PGCU which doesn't require explicit resolved guarding predicate information to make a predicate-aware prediction.

V.F Design Space Comparison

The predicate update branch prediction schemes presented in this chapter have somewhat varying abilities to reduce the additional occurrences of mispredictions that stem from predicate region formation and execution. However, these techniques also have varying costs and abilities in terms of the details of their architectural implementations. In this section we discuss a subset of the predicate update prediction schemes developed in this chapter with respect to their design and implementation.

There is one important factor distinguishing PGCU from all the other predicate-aware predictions schemes presented here. It can make a predicate-aware prediction with a single table lookup. The other schemes require a two table lookup which, if cannot be accomplished within a single cycle, means that no predicate-aware prediction can be made in the first-cycle prediction. As discussed in section II.C.2, branch prediction schemes which require two cycles to make a prediction are traditionally paired with a single-cycle predictor. The single-cycle predictor makes a prediction in fetch, then after the two-cycle predictor finishes, it may correct the prediction and start fetch down a different path. If that occurs a penalty of a single cycle is paid before the correct prediction is made. This one cycle penalty is not modeled in our results since we only report branch prediction rates. However, the complications introduced by delayed update of the global history register by predicate defines must be addressed, possibly by adding a deterministic timed update structure for predicate defines.

The PEP and APPEND prediction schemes both utilize branch prediction resources by using additional table entries to hold predictions for false path branches. In comparison, the Squash-FP filter reduces usage of branch prediction resources. The beneficial effects of this reduction should manifest in reduced branch misprediction rates.

In Figure V.19 we highlight the results from a selection of the branch prediction schemes we propose that cover a range of complexity in terms of implementation. We show how the relatively simple addition of a Squash-FP filter onto a baseline Meta Chooser predictor has significant results in reducing the impact of misprediction migration in predicated regions. Adding a PGCU predicate-aware global predictor to the baseline Meta Chooser increases complexity but allows for significant benefit for those benchmarks where few branches have resolved guarding predicates when fetched. The addition of a local predicate-aware prediction scheme sees very little improvement and cost of implementation is higher in that multiple local histories must be maintained. Additionally, PEP makes use of the same predicate information as the Squash-FP filter, so it is not surprising that less benefit results after combining the two.

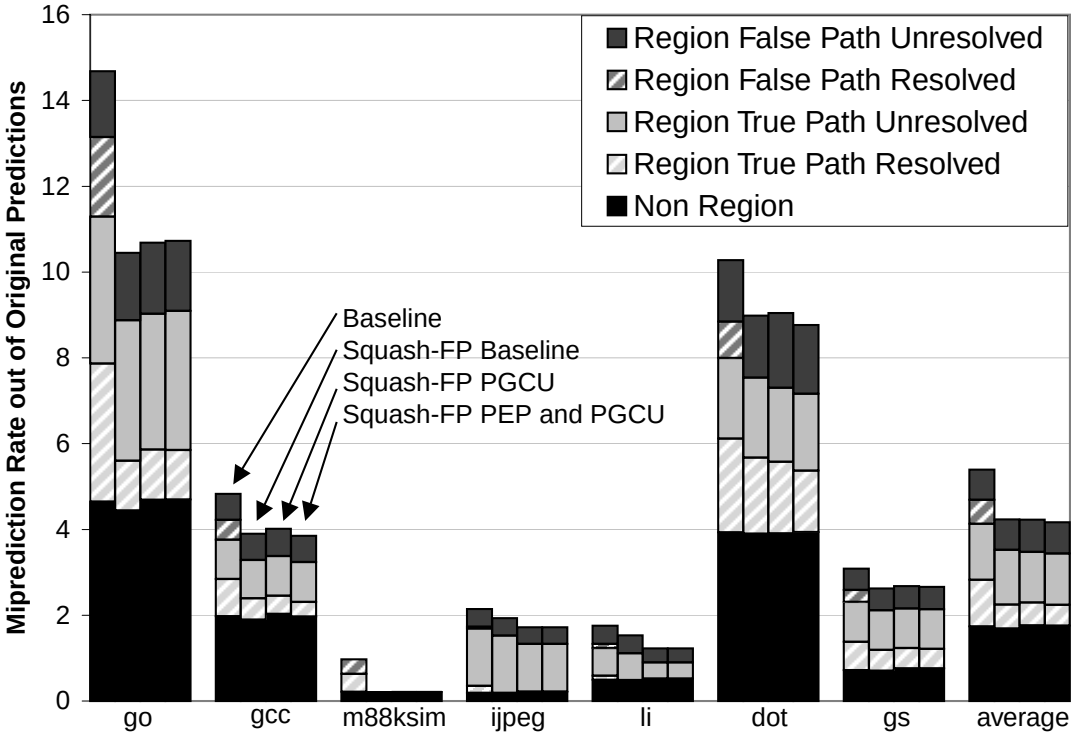


Figure V.19: A subset of the predicate update predictors examined in this chapter. From left to right, we highlight best performing designs in terms of increasing complexity of implementation. Both Squash-FP filter enabled schemes and PEP require storing branch/guarding predicate information and a two table lookup to produce a predicate-aware prediction. PGCU stores predicate information directly into the global history register and can make a predicate-aware prediction in one table lookup.

V.G Conclusion

Without any modification to branch prediction hardware, region branches become a major problem in achieving the reduced branch misprediction rates we expect from predicated codes. The ability to accurately predict these region branches is hindered by their increased dynamic occurrence from spurious predictions, their new prediction pattern based on their guarding predicate dependences, and the fact that information from predicate define statements is no longer available in the global history register. We show that, for our region formation technique across the benchmarks we studied, predicate region formation reduces the branch misprediction rate from 8% to 5.5% using a traditional branch predictor. However, an idealized execution of the codes where only true path branches have to be predicted yields an average misprediction rate of 2.7% and if an idealized (instant) update of global history from predicate define statements is added this drops to 2.4%. The impact of misprediction migration stemming from branches in predicate regions is a severe hindrance to branch prediction accuracy.

To address this problem, previous work has investigated augmenting local per-branch prediction schemes with guarding predicate information. The *PEP* prediction scheme uses direct branch/guarding predicate information to steer predictions from true path occurrences of a branch to one location in the local pattern history table, and false path occurrences to another. We propose the APPEND global predicate update scheme to complement the style of prediction produced by PEP. APPEND tacks the value of a branch's guarding predicate onto the current global history register value to form the index used to predict from the global pattern history table. Since PEP targets local predictions and APPEND targets global predictions, they can be used in concert in a Meta Chooser predictor. Our results show that, employed together, they can achieve an average misprediction rate of 4.4% across the benchmarks we studied. Using more accurate predicate define information for PEP and APPEND further reduces the mispredict rate an average .16%.

Recognizing that branches which are known at fetch to have false guarding predicates should be predicted not taken, we propose the *Squash-FP* filter. The Squash-FP filter can be used to augment any branch prediction scheme and achieves 100% prediction accuracy for spurious branches whose guarding predicate definitions have resolved by the time the branch is fetched. These falsely guarded branches, or resolved spurious branches, should always predict not taken. This technique also uses explicit branch/guarding predicate define information stored in the BTB similar to PEP and APPEND. It takes advantage of bypass information stored in the architecture pipeline to determine when branches are guaranteed to be spurious.

This technique also has the benefit of reducing contention in the branch prediction tables by no longer requiring predictor state that was previously used by false path branches. We show that, even augmenting a non-predicate-aware predictor, the Squash-FP filter achieves a sizable reduction in branch mispredictions (ranging from .2% to 4.2%). This method is arguably the simplest predicate update modification to current branch prediction architectures.

Squash-FP, PEP, and APPEND are all similar in that they rely on direct branch/guarding predicate information to produce predicate-aware predictions. This requires a serial two table lookup in the prediction hardware (one to identify the branch's guarding predicate register and one to read the value of that register). We propose the *Predicate Global Correlation Update Branch Predictor* (PGCU) architecture which can produce a predicate-aware prediction in a single table lookup by directly incorporating predicate define information into the global history register. The PGCU allows predicate define statements to provide correlative information to the branch predictor state by updating the speculative global history register at writeback. We model updating the speculative global history register out-of-order using predicate define statements, and recovering the state of the branch predictor on a branch misprediction. PGCU can be augmented with a Squash-FP filter and PEP to reduce misprediction rate to an average of 4.16%.

While the majority of predicate-aware prediction benefit comes from targeting spurious branch predictions via Squash-FP, Squash-FP may not be able to produce a predicate-aware prediction in a single cycle. In contrast, PGCU may provide an attractive option for providing single cycle predicate update predictions.

Chapter VI

Conclusions

Architectural support for predicated execution has been proposed as a manner of attacking performance bottlenecks resulting from modern processor pipeline design. In order to achieve desired high-frequency processors, processor pipelines have increased in length - requiring more stages for a given instruction to complete execution and allowing more instructions to be in-flight in the pipeline at one time. This causes higher penalties for mispredicted branches as more work will have been fetched from the wrong path of execution by the time a branch is resolved at the end of the pipeline.

Predicated Execution supports an alternate form of conditional execution that delays the need for the result of control flow decisions until the end of the processor pipeline. This mechanism allows one to trade the benefit of reduced branch misprediction penalties for the cost of the execution of multiple execution paths following a control flow branch. However, this new form of conditional execution expression breaks traditional compiler and hardware assumptions. Compilers traditionally utilize a control flow graph made up of basic block nodes and have built their entire data-flow analysis, optimization, and code-scheduling infrastructure around this framework. With predicate define data dependence relationships usurping the role of branches and control flow edges, compilers must be able to gather, analyze, and manage predicate relationship data in the same way they currently manage control flow graphs. Branch prediction hardware currently maintains and utilizes control flow information at run-time to make informed branch prediction decisions. Predicate defines replace branches - effectively removing dynamic control flow information from the branch prediction hardware. Additionally, predicated regions contain region branches - branches guarded by predicates that will sometimes

be false. These spurious branch executions have a significant impact on the ability of traditional branch prediction hardware to make good branch prediction decisions, as well as generally increasing the number of dynamic branches that must be predicted. Our experiments show that, after predication, 33% of all dynamic predictions are made for spurious branches.

In this dissertation we expose the importance of utilizing predicate information in both the compiler, for scheduling predicated code, and in the architecture for maintaining branch prediction accuracy in the face of predicated code.

We present Predicated Static Single Assignment (PSSA) as a method of exposing full-path relationships for predicated regions of code. PSSA is a predicate-sensitive implementation of SSA that implements renaming using full-path predicates and can be used to eliminate false dependences for predicated code. We showed the benefit of using PSSA to enable Predicated Speculation (PSpec) and Control Height Reduction (CHR) during scheduling. Predicated Speculation allows operations to be executed at their earliest schedulable cycle, even before their guarding predicates are determined. Control Height Reduction allows guarding predicates to be defined as soon as possible, reducing the amount of speculation needed.

By maintaining information about each of the original control paths in a hyperblock, PSSA can provide information that allows precise placement of renamed and speculated code, and allows the correct, renamed values to be propagated to subsequent operations. The renaming used by PSSA allows more aggressive speculation, as overwriting live values is no longer a concern. In addition, PSSA supports Control Height Reduction along every control path using full-path predicates, reducing control dependence depth throughout the hyperblock.

Our experiments show that PSSA is an effective tool for optimizing predicated code. We provide extended experiments that show using PSSA with PSpec and CHR results in a reduction in executed cycles ranging from 12% to 68% for a 16 issue machine.

Without any modification to branch prediction hardware, region branches become a major problem in achieving the reduced branch misprediction rates we expect from predicated codes. The ability to accurately predict these region branches is hindered by their increased dynamic occurrence from spurious predictions, their new prediction pattern based on their guarding predicate dependences, and the fact that information from predicate define statements is no longer available in the global history register. We show that, for our region formation technique across the benchmarks we studied, predicate region formation reduces the branch misprediction rate from 8% to 5.5% using a traditional branch predictor. However, an idealized execution of the codes where only true path branches have to be predicted yields an average

misprediction rate of 2.7% and if an idealized (instant) update of global history from predicate define statements is added this drops to 2.4%. The impact of misprediction migration stemming from branches in predicate regions is a severe hindrance to branch prediction accuracy.

To address this problem, previous work has investigated augmenting local per-branch prediction schemes with guarding predicate information. The *PEP* prediction scheme uses direct branch/guarding predicate information to steer predictions from true path occurrences of a branch to one location in the local pattern history table, and false path occurrences to another. We propose the APPEND global predicate update scheme to complement the style of prediction produced by PEP. APPEND tacks the value of a branch's guarding predicate onto the current global history register value to form the index used to predict from the global pattern history table. Since PEP targets local predictions and APPEND targets global predictions, they can be used in concert in a Meta Chooser predictor. Our results show that, employed together, they can achieve an average misprediction rate of 4.4% across the benchmarks we studied.

Recognizing that branches which are known at fetch to have false guarding predicates should be predicted not taken, we propose the *Squash-FP* filter. The Squash-FP filter can be used to augment any branch prediction scheme and achieves 100% prediction accuracy for spurious branches whose guarding predicate definitions have resolved by the time the branch is fetched. These falsely guarded branches, or resolved spurious branches, should always predict not taken. This technique also uses explicit branch/guarding predicate define information stored in the BTB similar to PEP and APPEND. It takes advantage of bypass information stored in the architecture pipeline to determine when branches are guaranteed to be spurious. This technique also has the benefit of reducing contention in the branch prediction tables by no longer requiring predictor state that was previously used by false path branches. We show that, even augmenting a non-predicate-aware predictor, the Squash-FP filter achieves a sizable reduction in branch mispredictions (ranging from .2% to 4.2%). This method is arguably the simplest predicate update modification to current branch prediction architectures.

Squash-FP, PEP, and APPEND are all similar in that they rely on direct branch/guarding predicate information to produce predicate-aware predictions. This requires a serial two table lookup in the prediction hardware (one to identify the branch's guarding predicate register and one to read the value of that register). We propose the *Predicate Global Correlation Update Branch Predictor* (PGCU) architecture which can produce a predicate-aware prediction in a single table lookup by directly incorporating predicate define information into the global his-

tory register. The PGCU allows predicate define statements to provide correlative information to the branch predictor state by updating the speculative global history register at writeback. We model updating the speculative global history register out-of-order using predicate define statements, and recovering the state of the branch predictor on a branch misprediction. PGCU can be augmented with a Squash-FP filter and PEP to reduce misprediction rate to an average of 4.16%.

While the majority of predicate-aware prediction benefit comes from targeting spurious branch predictions via Squash-FP, Squash-FP may not be able to produce a predicate-aware prediction in a single cycle. In contrast, PGCU may provide an attractive option for providing single cycle predicate update predictions.

Bibliography

[1] Next Generation Instruction Set Architecture. Transcript from Microprocessor Forum, Oct 14, 1997., 1997. http: //developer.intel.com/design/next/sld001.htm, 1998.

[2] Intel Press Release. Merced processor and IA-64 architecture., 1998. http: //developer.intel.com/ design/processor/future/iaa64.htm, 1998.

[3] Trimaran, An Infrastructure for Research in Instruction Level Parallelism, 1998. http: //www.trimaran.org.

[4] IA-64 Application Developer's Architecture Guide, Revision 1.0, 1999.

[5] *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*. January 2000.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, 1986.

[7] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, December 1994.

[8] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 33(5):72–84, May 1998.

[9] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[10] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu. The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual. Technical Report IMPACT-98-04, IMPACT, University of Illinois, Feb 1998.

[11] D. I. August, W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[12] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *The 3rd International Symposium on High-Performance Computer Architecture*, pages 84–93, 1997.

[13] David I. August, John W. Sias, Jean-Michel Puiatti, Scott A. Mahlke, Daniel A. Connors, Kevin M. Crozier, and Wen mei W. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 208–219, 1999.

[14] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, December 2–4, 1996.

[15] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[16] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[17] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[18] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[20] 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. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[21] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1-2):181–227, May 1993.

[22] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 114–125, December 1996.

[23] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimation using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, November 10–14, 1997.

[24] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[25] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 100–113, December 1996.

[26] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, HP Labs, Feb 1994.

[27] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprosessor architecture. In *International Conference on Computer Design*, December 1998.

[28] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.

[29] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[30] 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 Annual Intl. Symp. on Microarchitecture*, pages 217–227, December 1994.

[31] 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 *Proceedings of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[32] Scott A. Mahlke and Balas K. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 153–164, 1996.

[33] S. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[34] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[35] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, January 1989.

[36] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. In *IEEE Transactions on Computers*, April 2001.

[37] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, November 29–December 1, 1995.

[38] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 40–51, December 1994.

[39] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[40] Beth Simon, Brad Calder, and Jeanne Ferrante. Incorporating predicate information into branch predictors. In *Proceedings of the First Workshop on EPIC Architecture and Compiler Technology*, 2001.

[41] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction Level Parallelism*, January 2000.

[42] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. Association for Computing Machinery, 1994.

[43] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, November 30–December 2, 1994.

[44] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.