Predicate Prediction for Efficient Out-of-order Execution

Weihaw Chuang Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{wchuang,calder}@cs.ucsd.edu

Abstract

Predicated execution is an important optimization even for an out-of-order processor, since it can eliminate hard to predict branches and help to enable software pipelining. Using predication with out-of-order execution creates a naming bottleneck, because there can be multiple definitions reaching a use, and not knowing which use is the correct one can stall the processor.

In this paper, we examine using predicate prediction to speculatively allow execution to proceed in the face of multiple definitions. We show that the penalty for mispredicting a predicate is not as severe as mispredicting a branch. Thus, making it advantageous to replace hard to predict branches with predicate predictions. We present a predicate misprediction recovery architecture that replays instructions through the renamer to link up the correct dependencies on a misprediction. This approach allows us to avoid putting the predicted false path instructions in the issue queue reducing the pressure on the dynamic out-of-order scheduler.

Categories and Subject Descriptors

C. - Computer Systems Organization, C.1 - Processor Architectures, C.1.0 -General

General Terms

Design, Performance

Keywords

Predicate Prediction, Predicated Execution

1. INTRODUCTION

In this work we concentrate on techniques to efficiently apply predication to out-of order processors. Predication is beneficial even in an out-of-order processor by eliminating hard to predict branches using if-conversion [15]. By eliminating branches, if-conversion also encourages greater instruction fetch throughput, since fewer branches need to be predicted per cycle.

In out-of-order execution, register renaming demands that every register-writing instruction have a new register name allocated. Work by Wang et. al. [20] examined the performance of an out-of-order processor using IA64. They identified that predication is a performance bottleneck due to resolving the ambiguity caused by having multiple register definitions defined on the same control flow graph. We call this the *multiple-definition* problem, and it arises when during rename, a register use needs to determine what definition to take its value from. If there are multiple definitions of the same register guarded on different predicates, and the predicates have not been resolved, then it is unknown which definition the use should be linked to. Figure 1 illustrates the problem. The predicate generating CMP.EQ instruction has yet to reach the execute pipeline stage, so when the ADD reaches the rename stage it doesn't know which MOV is the correct assignment to R33. One solution to the multiple-definition problem is to stall the ADD instruction until it is clear which is the true definition of R33, but this results in a loss in performance over not predicating the program at all.

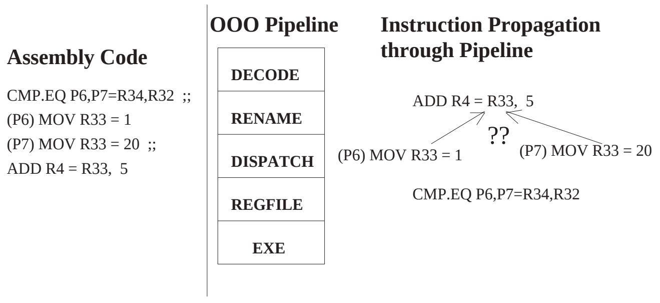


Figure 1: Multiple Definition Problem: Which value does R33 take for its use? This ambiguity causes performance issues for out-of-order machines with predication.

In this paper we examine using *predicate prediction* to efficiently resolve the multiple definition problem. By performing predicate prediction before register renaming occurs, we predict which predicate paths will be true and which will be false.

Predicate prediction is closer to value prediction than branch prediction. Predication is used in if-conversion to turn control-flow into data-flow. Lipasti et.al. [13] and others extend the idea of branch speculation to data-flow to break data dependencies through value prediction. They found that value prediction could be used to break data-flow dependences to help increase instruction-level-parallelism. With predicate prediction we speculatively predict predicate values to break the data-flow dependencies in the "multiple-definition" problem.

Predicting predicates may appear counter intuitive, as if-conversion is used to eliminate hard to predict branches. However, the predicate misprediction penalty is not as severe as a branch misprediction. A branch misprediction flushes the entire pipeline of instructions, when the wrong path of instructions is fetched. Then fetch is redirected down the correct path. On the other hand, a predicate misprediction does not need to refetch down the correct path, since both paths of the control flow were fetched for the if-converted region. Therefore, a predicate misprediction requires only the replay of instructions with the correct dependencies enforced. This costs fewer pipeline penalties than the full pipeline flush of a branch misprediction. In this paper, we examine two unique replay predicate recovery architectures, which we call "rename-replay" and "selective-replay". Both of these are similar to previously proposed squash and selective replay architectures examined in value prediction, but modified to hook up the correct dependencies on a misprediction replay. Predication used with predicate prediction and replay can be thought of as a form of multi-path instruction fetch directed by the compiler.

We also examine using predicate prediction to simplify register renaming and out-of-order execution in a process called *predicate early evaluation*. Predicate early evaluation eases the pressure on the issue queue and the out-of-order dynamic scheduler by not inserting into the issue queue falsely guarded instructions, where the value of the guarding predicate is either the real value or a predicted value.

The paper is organized as follows. In section 2, we discuss related work to out-of-order execution with predication. Section 3 describes the hardware structures to implement predicate prediction. To understand later results, we state in Section 4 the methodology used for simulation. Section 5 provides a comparison of alternative design choices, and demonstrate why predicate prediction is a good platform for out-of-order execution with predication. In addition, we compare predicate prediction to Wang et.al. [20] and the most recent work in this area. Section 6 summarizes our paper.

2. RELATED WORK

In this section we describe related work on dealing with the multiple-definition problem for out-of-order architectures with predication.

2.1 Prior Hardware Solution for Multiple Definition Problem

Wang et.al. [20] recognized that multiple definitions would be a problem in the renaming stage of an out-of-order implementation of IA64. The renaming stage is used to give each definition of an architectural register a unique physical name (removing WAW and WAR dependencies). With conditional-writer predication, it is possible to have multiple instructions guarded by different predicate registers write to the same architectural register. When a use of this architectural register is encountered in the rename stage, the values of the predicates may be required to determine which physical register to map to the architectural register. If the predicate values are not yet available, a stall must occur.

In an effort to remove as many unnecessary stalls as possible, Wang et. al. proposed the use of the select-μop instruction in hardware for an IA64 out-of-order execution model. The new select-μop instruction was conceptually based on the ϕ-function used by static-single-assignment (SSA). It allows the resolution of multiple definitions to be postponed to later stages of the pipeline, providing greater chance that a stall would not have to occur. To form the select-μop instructions, the possible definitions of a use are needed. To this end they presented an augmented *Register Alias Table* (RAT), and use this to create the select-μop instructions. Each set in the RAT represents all of the current potential definitions for each logical register. Each entry contains the renamed definition and the guarding predicate under which it was defined. The most recent definition guarded by a true predicate is the correct definition. Once this definition is determined, any further dependencies left to be reconciled could be eliminated from consideration. We simulated the hardware select-μop mechanism used by Wang et. al. [20], and compare its results to using predicate prediction.

Perhaps the earliest paper on executing predicated code on an out-of-order machine was done by Chang et.al. [3]. They noted the multiple-definition problem, and solve it by converting all predicated instructions into an internal select following the instruction. Depending on the predicate the select chooses the original destination value, or the computed result. This requires an extra register file read port for each instruction (to read the original value), and incurs a serialization on predicate instructions. The Wang et.al. technique subsumes this idea but reduces the serialization penalty through their wide fan-in select-μops.

In implementing an out-of-order processor [11], the Alpha 21264 dealt with the conditional move instruction used for light-weight if-conversion. They split the CMOV into two instructions that selected the source of the CMOV or the destination, depending on the predicate and behaving very much like a select operation.

2.2 Solutions for Different Instruction Sets

Klauser et.al. [12] examined dynamically if-converting if-then-else hammocks for an out-of-order processor to eliminate hard to predict branches. They depend on the compiler or binary back-annotation tool to provide hints that a control region forms a hammock that can be if-converted, but otherwise don't require a predicated instruction-set. The processor decoder uses confidence predictors to determine if-conversion profitability, and when selected uses internal "CMOV" (selects) to choose the results of the executed path (qualified true).

Our prior work [4] examined a light-weight ISA extension targeted at if-conversion, called Phi-predication. Phi-prediction is derived from select-predication first seen in [14] with features for qualifying memory and predicate assignments, to increase the applicable control-flow regions. Select-predication always assigns its register destination, and behaves like regular RISC operations, thereby avoiding the multiple definition problem for out-of-order execution. This paper makes extensive use of predicate promotion and move combining optimizations to further improve performance. Despite the utility of these technique, providing an ISA compatible solution for conditional-writer predicated ISA's like IA64 remains an important goal.

2.3 Predication Taxonomy

This section assists classifying predication by modes of operation, both at the instruction set level and implementation. It demonstrates how prior work relates to one another, and to this paper. First we consider the differences at the ISA. Conditional-writer predication qualifies the state update of an instruction under some predicate register. Select predication uses select instructions to choose between two different registers, and always updates its definition. In Figure 2 on the left, illustrates this ISA taxonomy.

In our taxonomy, the IA64 ISA [10] architecture falls under conditional-writer, as do ISA's using conditional move (CMOV) operations like Alpha. These ISA's suffer from the multiple-definition problem on Out-Of-Order implementations as described by [20] for IA64, by [11] CMOV on Alpha 21264. An example

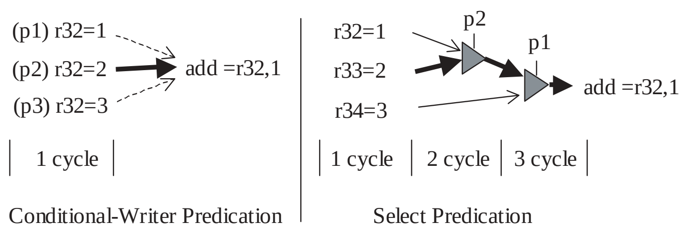


Figure 3: Conditional writer predication on the left evaluates in a single cycle, but must deal with the multiple-definition problem. Select predication inherently has already dealt with the multiple-definition problem, so it may need several cycles to evaluate the select chain. Both examples illustrate R22 taking the value 2 via the bold path. Dashed lines indicate no assignment is made. Triangles represent select operations.

of select predication is phi-predication extensions as noted in Section 2.2. Select predication does not suffer from the multiple definition problem, but suffers from a serialization problem called select (phi) chaining. This is shown in Figure 3. We can further classify ISA's by the generality of predication, where IA64's predication applies to nearly all instructions, while Alpha's predication is only present on a single CMOV instruction. Generality correlates with the complexity necessary to implement that form of predication, as denoted by the "heavy-weight" and "light-weight" criteria.

We now consider a taxonomy based on implementation, since it can be different than the original ISA. This observation is motivated by three papers that overcome the multiple definition problem for the conditional-writer predication by translating to a select predication internal representation [20, 3, 11]. We can see this taxonomy on the right of Figure 2. At the first level of the taxonomy, machines are classified by whether the approach is translated, or directly executes on hardware. Translated approaches have greater implementation cost but remain backwards compatible to their original ISA. For direct execution, conditional-writer (e.g. IA64 [17, 10]) executes efficiently only on in-order machines, while select can execute on both in-order and out-of-order [4]. Predicate prediction, as described in this paper, is a general technique that can be applied to assist a conditional writer model execute efficiently on out-of-order.

3. HARDWARE CONCEPTS

To address the multiple definition problem we examine using predicate prediction. We examine predicting predicates as they are decoded using a traditional branch prediction architecture. The most recent definition with a predicted true guarding predicate is assumed to be the current definition to speculatively avoid the multiple-definition problem. If the prediction is incorrect, then the instruction that used the prediction needs to (1) be hooked up to the correct definition, and (2) needs to be re-executed using the out-of-order processors replay mechanism. We describe these techniques in more detail in the rest of this section.

3.1 Predicate Predictor

Predicate prediction is the process of predicting predicates. These predicate definitions were once a branch prediction before if-conversion was performed. Therefore, we examined using well studied branch prediction architectures to provide the predicate predictions [16, 21].

The mapping from branch predictors is as follows: given the PC of the predicate defining instruction, a prediction is generated. The resulting taken and not-taken prediction directions provided by a branch predictor are mapped to True and False for the predicate prediction. One prediction is provided for each predicate generating operation. Since a typical predicate defining instruction (compare) generates two predicate definitions, one boolean prediction is used to predict the first definition, and the predicated value of the second predicate is derived from the boolean relationship it has with the first predicate and the qualifying predicate. As our code is limited to "unconditional" compares, we only have to worry about these and the value of the qualifying predicate. When the qualifying predicate is true, then one of the output definitions is predicted and the other is the inverse of the prediction. When it is false, the "unconditional" form assigns the predicates to false. Potentially one could predict both definitions instead of one to improve accuracy, but we did not find that necessary for our if-conversion algorithm. Other relationships such as "OR" and "AND" compare forms, complicate this simple relationship, and most likely could be solved by predicting both definitions.

In this paper, we examine having a separate predicate prediction architecture from the branch prediction hardware. The branch prediction hardware does its lookup prior to fetch, while the predicate predictor does its lookup during decode once we know the fetched instruction is a predicate definition. We separate them because design constraints are different for predicate predictors and branch predictors.

There might be benefit in merging the predicate and branch history to improve prediction accuracy, but the following four considerations compelled us to split them. Unlike modern branch predictors, predicate predictors only need the branch history table (or more aptly called boolean-value history table), and not the return-address stack or the branch target buffer. A second important consideration is that predicate predictions are independent of one another unlike branch predictions. Branch predictions have an implicit serial control dependency needed to form an execution trace. Independence for the predicate prediction makes it possible to build large multi-port predicate predictors, or replicate them. Therefore, we focus on only using local history prediction information for predicate prediction. Third, predicate values are not needed till register renaming allowing multi-cycle predictor lookups. Forth, many predicate predictions may be necessary. For a machine with Itanium resources [8], it needs at least six predictions for the four integer and two floating point compare instructions able to execute concurrently.

3.2 Predicate Early Evaluation

Predicate early evaluation is the application of the predicate state during the rename stage as shown in Figure 4. For an out-of-order processor, we use this to eliminate the multiple-definition problem at the register renamer. Qualified false instructions evaluate to NOP-like instructions, and don't affect register mapping, leaving only resolved true or predicted true instructions to change the register mapping. This allows subsequent pipeline stages to view instructions as normal RISC-like instructions without specialization for predication, simplifying many implementation details. For example, score-boarding and register bypassing, memory dependency resolution, and exception determination need not be predicate aware. These structures are complicated even in-order implementations as seen on Itanium [18].

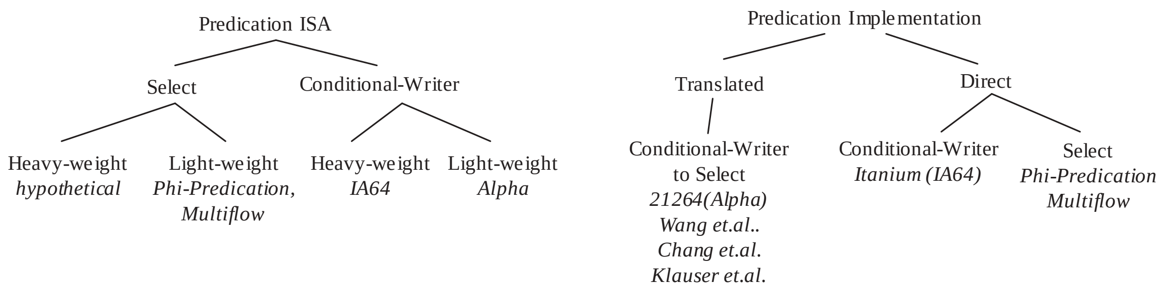

Predication ISA
Select
Conditional-Writer
Heavy-weight hypothetical
Light-weight Phi-Predication, Multiflow
Heavy-weight IA64
Light-weight Alpha
Predication Implementation
Translated
Direct
Conditional-Writer to Select 21264(Alpha) Wang et.al.. Chang et.al. Klauser et.al.
Conditional-Writer Itanium (IA64)
Select Phi-Predication Multiflow

Figure 2: Taxonomy of predication based on ISA (left) and implementation (right). We can avoid the multiple definition problem on conditional-writer by translating the ISA to select-predication in the implementation, or by using predicate prediction.

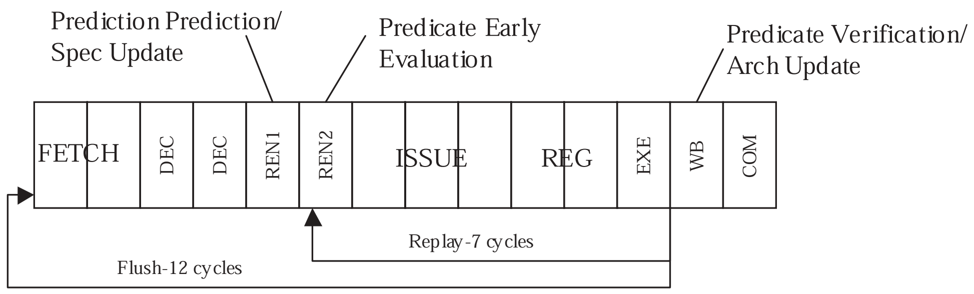

Prediction Prediction/ Spec Update
Predicate Early Evaluation
Predicate Verification/ Arch Update
FETCH
DEC
DEC
REN1
REN2
ISSUE
REG
EXE
WB
COM
Replay-7 cycles
Flush-12 cycles

Figure 4: Pipeline: Predicate prediction is completed by rename stage 1 (REN1). The predicted predicates and the true predicate values are early-evaluated at REN2. Architectural predictor state updates occurs at commit, though speculative update is at REN1.

For predicate early evaluation, a predicate value used during renaming can be a predicted value, or can be the actual computed state if the predicate is already resolved. Resolved predicates occur if there is sufficient execution distance, in cycles, between the predicate definition and its use for qualification.

3.3 Predicate Misprediction Recovery

With speculative early evaluation, the hardware verifies that the prediction was made correctly. If it was not, misspeculation recovery initiates. This process corrects the mispredicted predicate state, and fixes up the register data-flow caused by the incorrect register renaming. There are potentially at least three different recovery mechanisms possible. We first discuss using the branch misprediction hardware for predicate misprediction recovery. Second, we consider rename-replay, which replays instructions from a recovery instruction buffer through the register renamer to properly hook up dependencies. Third, we consider selective-replay, which replays the mispredicted instructions through the broadcast of the correct values using a selective-replay recovery tag in the dynamic scheduler. We discuss all three techniques in the following subsections.

3.3.1 Flush Predicate Misprediction Recovery

Branch misprediction recovery hardware does all the needed work to recover from a mispredicted predicate. Upon finding a misprediction for a predicate generating instruction, the recovery hardware flushes the instruction pipeline and refetches instructions from the I-Cache.

When the recovered instructions travel through the register renamer, it uses the corrected predicate state and remaps the registers to the correct data-flow. Using the branch misprediction recovery path would seem like a natural design choice. However the penalty of branch prediction loop includes refetching and decoding, which is an unnecessary task, since the instructions are already in the pipeline.

3.3.2 Rename-Replay for Predicate Mispredictions

The next idea is to reuse instructions already fetched and decoded to reduce the recovery loop. When a predicate misprediction is detected in write-back, this form of replay takes all instructions seen by the pipeline from the point of the 1st use of the mispredicted predicate to the last instruction that was renamed and sends them all back through the register renamer and back into the issue queue to hook up the correct dependencies. All instructions before register renaming are stalled until the recovery process is done. Because this recovers instructions through the renaming to correct instruction dependencies we call this “rename-replay”.

Just like branch speculation, the register rename mapping needs to be checkpointed. We checkpoint the register mapping at the *first use of a predicate prediction*, since this is the point at which the rename-replay would start if the predicate was mispredicted.

Since we are using early predicate resolution, instructions that are qualified on false predicates (predicted or resolved) are not put into the issue queue to be dynamically scheduled. This is to reduce the pressure on the issue/instruction queue. We instead store the instructions in a dedicated buffer called the recovery queue (RecQ). The RecQ can be thought of as separate from the Reorder/Commit buffer or a pointer into it. Instructions are replayed at the first use of a predicate misprediction from the RecQ back into the rename stage. All instructions are kept in-order in the RecQ and are replayed on a misprediction. During normal execution, only instructions with a predicted or true path guarded predicate enter the instruction queue and are dynamically scheduled.

These properties can be seen in Figure 5. In this example, six instructions (IN0 to IN5) are fetched and sent through the pipeline. We then show three points in their progress through the pipeline. The first step represents normal execution where the instruction at Rename are evaluated based on the predicted predicate where P6 is predicted to be TRUE and P7 is predicted to be FALSE. Instruction 3 is predicted true and enters the IQ, while instruction 2 is predicted false and does not enter. Instruction 4 is always true and enters both the IQ and RecQ. Instruction 1, which was previously predicted true, is (speculatively) issued, and its entry removed from the IQ. Note that all instructions are copied in-order to the recovery queue, just like a ROB/Commit buffer. In step 2, evaluation of the compare at instruction 0 finds a misprediction of the predicates. The first instruction to use the mispredicted value, instruction 1, marks the beginning of the replay in the RecQ. It and all subsequent instructions in the issue queue are sent through the register-renamer to fix the register dependencies. When going through early-evaluation in the rename stage a second time, instructions 2 and 4 evaluate to true. Instruction 0 is not touched by the replay event, and continues towards commit. In step three, normal execution resumes. By this point, instruction 0 has committed and was removed from the RecQ. Instruction 5 enters the IQ and RecQ with the corrected register rename tags.

3.3.3 *Selective-Replay for Predicate Mispredictions*

Selective replay can further improve the performance of the recovery mechanism, as it only recovers instructions that use incorrect state, allowing other instructions to execute. For predicate prediction recovery, the mispredicted instructions and any dependent instructions that use its results need to be re-executed using the updated predicate information. For this paper we explore one potential implementation of selective-replay for predicate prediction.

For the selective-replay model, all instructions- predicted qualified true and false- are placed in the issue queue (IQ) due to selective recovery. In this model, all of the possible dependencies are kept track of through serialization for replay in the issue queue, therefore they do not have to go through the renamer on a replay to fix up the dependencies.

To correctly find the dependencies during a replay on a predicate prediction, we keep track of two tags for each operand and an extra input tag to retain destination register state. For the operands, the two input tags represent (1) the predicted data flow when the instruction first enters the issue queue, and (2) the replay data flow that conservatively links def-use chains. This second recovery operand tag links to the nearest potential definition, no matter whether it is qualified true or false. If the instruction is qualified on false, a replay tag associated with the destination register, propagates the original definition state. This tag acts as an input tag, and passes the original value through the instruction. The overall effect is to create a *serialization* through the multiple definitions for a register guaranteeing that the correct value will eventually reach all uses on a replay. On a replay, starting with the first recovery instruction, it writes to the replay tag (2). Input operands on replay instructions can accept either the regular or replay data-flow tag. All dependent instructions are re-executed in data-flow fashion.

These register renaming and replay concepts are illustrated in a diagram in Figure 6. The top left of the figure shows the original predicated code example, where we are executing five instructions. The top right shows the instruction queue state after these instructions have gone through predicate prediction and early register evaluation and illustrates the recovery tags. In this example, p6 is predicted to be TRUE, and P7 and P8 to be FALSE. The instruction queue state shows two tag values for each operand as described above. For example, the `shl` instruction has t5 as the predicted definition and `t11` as the recovery definition shown as `(t5,t11)`. If `shl` ends up being TRUE and no prior instructions before it are replayed, it will get its value from `t5`. If a prior instruction before it ends up being replayed, then it would get its value instead through `t11`. The instruction queue state is the destination recovery tag `< tag >` at the end of each instruction line. This tag points to the last instruction that defined the same register. This tag is used to propagate the correct value serially if the instruction is qualified false. When a replay occurs the multiple-definition problem is handled by serializing all potential definitions through each other, which we will show in the rest of this example.

The bottom part of Figure 6 shows two different misprediction examples. The first example on the bottom left shows what happens when the `shl` ends up being TRUE when it was originally predicated as FALSE. The state in the example shows the dependencies used during the replay with the solid arrow. In this case, since `shl` is TRUE it will get its input dependency from `t5`, and then selectively propagate its value.

The second example on the bottom right, shows the case when the `sub` instruction is TRUE when it was predicted to be FALSE. In this example, it will gets its operand value from `t5`, and propagate its result value using its destination tag of `t11`. Note that the correct destination tag is propagated through the `shl` instruction using its destination recovery tag. The `shl` ends up acting like a move and it knows to take the tag `t1l`, since it is guarded on FALSE, and then it propagates its value using its destination tag of `t12`.

Traditional selective-replay implementations are complicated, with predication making them more so. Recent literature from industry seems to suggest that the complexity level necessary for general selective-replay is considered tractable. Borsch et.al. [1] state that the Alpha 21464 would have used selective replay, as compared to the window-replay mechanism used in 21264. They describe the realistic implementation detail for both. The Pentium 4 paper [7] simply states it uses selective-replay. Recent academic papers for example by Ernst et.al. [6] use selective-replay exclusively as the recovery mechanism in their high performance speculative trace execution processor. Taking the next step to include predicates, may not be that far fetched.

3.3.4 *Broadside Predicate Register Assignment*

IA64 contains several instructions for the maintenance of predicate state that causes difficulties in an out-of-order machine. Broadside read and write instructions are used to preserve predicate register file state across procedure calls (and software pipelined loops). Simultaneous “broadside” writes of the predicate register file, assigns all 64 predicates through special MOV instructions, and imposes a synchronization point for all predicate registers in the out-of-order pipeline. For an out-of-order processor, all 64 predicate writes need new physical definitions and need to be renamed. This general problem is faced by all out-of-order IA64 implementations that support a predicate broadside write.

We model using a physical predicate register file that supports broadside writes to a sequential series (vector) of 64 predicate registers. This allows a broadside write to write a whole vector of 64 physical predicate registers at a time. Each vector is represented as a single tag in the register map, and an instruction would find its predicate value by indexing into the predicate register file with the `tag + virtual predicate number`. If a vector of 64 predicate registers is not available for the broad-

```
        pred cor                               pred cor
IN0              cmp p6,p7=r15,2       IN3    T   F  (p6) shl r17=r16,4;;
IN1     T   F  (p6) add r16=1,r8;;   IN4            st4  [r18]=r17
IN2     F   T  (p7) sub r17=13,r8    IN5            shl  r19=r17,3
```

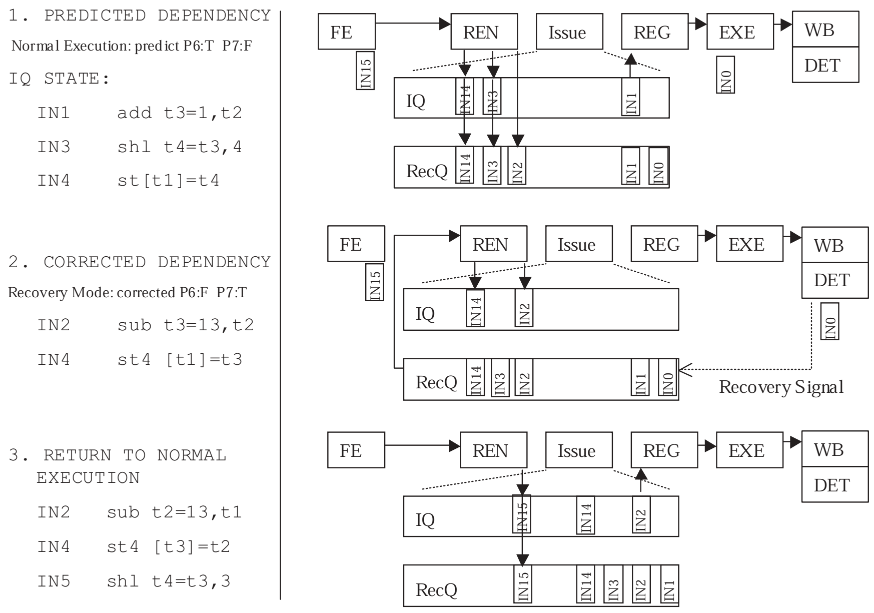


Figure 5: Conceptual view of Rename-Replay for predicate misprediction recovery. Top figure indicates normal operation. Middle figure receives a recovery signal when the compare (IN0) writes back its predicates. This wakes up instructions from the Recovery Queue (RecQ), and forwards them to the register renamer. These instructions are early-evaluated with the correct predicate, and register dependencies are repaired. The bottom figure shows execution starting from the corrected Instruction Queue (IQ) state.

side write during renaming, the renamer would stall until they became available. In our simulations, this rarely ever occurred, and this was modeled with having two 64 predicate vectors (for renaming broadside predicate writes) on top of the default register file.

4. METHODOLOGY

Our simulation methodology was derived from an earlier paper [4], but is described in greater detail in this paper. Most of our infrastructure work for this paper was to modify SimpleScalar-3.0 [2] to support predicate prediction and gather supporting results.

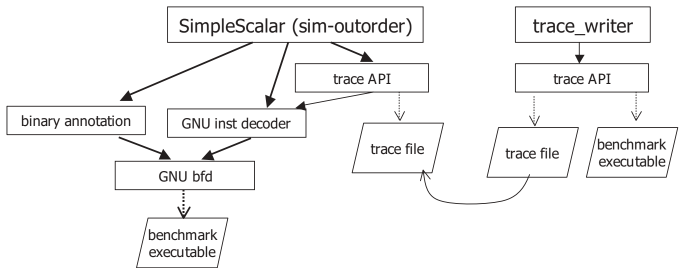


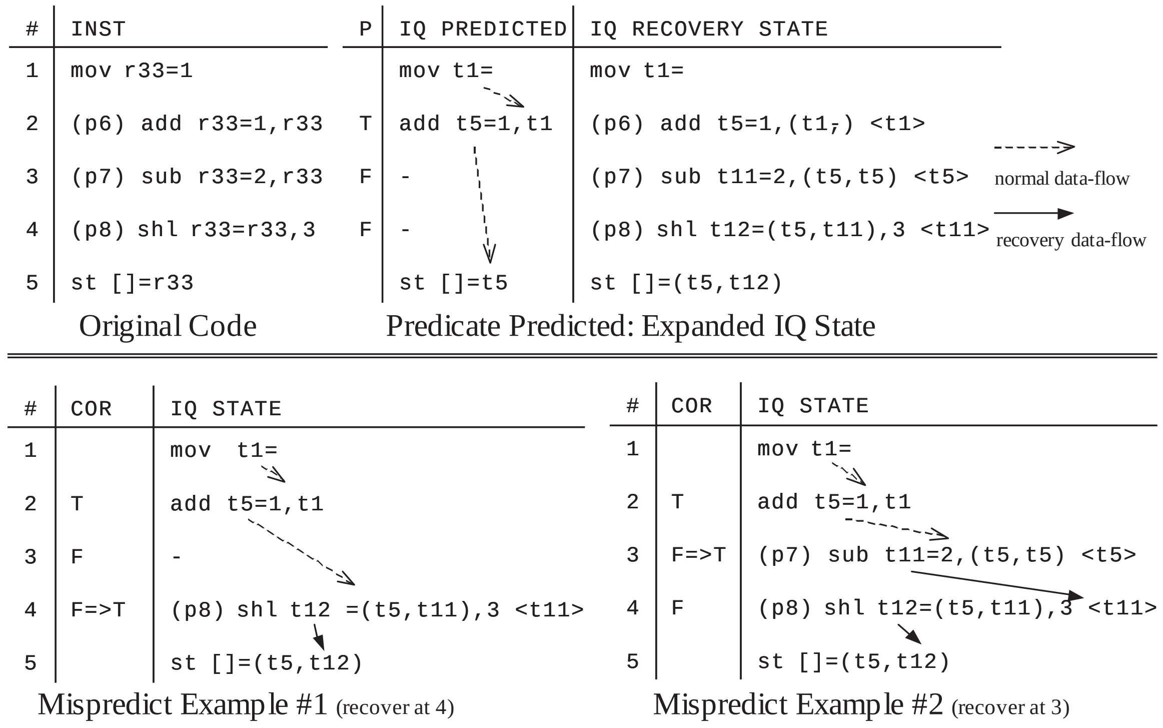
Figure 7: Software Components for Simulator and Trace-Writer.

4.1 Trace Generation

We built a trace driven IA64 simulation environment with SimpleScalar-3.0. The key components of the tool chain are shown in Figure 7. We started with a set of tracing and decoding libraries. GNU bin utilities provide a basic opcode library and instruction decoder that we enhanced with unique opcode identifiers, functional unit classification, and several other features. We also use GNU “libbfd” to access the “.text” section. Over this we built a custom tracing infrastructure using concepts learned from David Mosberger’s ”utrace.c” utility. Like his tool, our tracing mechanism uses *ptrace*, an API frequently used by debuggers, to access low-level, child-process context information [5]. Each trace record contains state for the execution of an IA64 instruction. It contains an IP (instruction pointer), PR (predicate register file), and either memory address or the register stack engine (RSE) state from AR.PFS. This information is sufficient to simulate the most important performance features of IA64. These are predication, RSE, software pipelining (not used in this paper), caches, branch prediction, and execution core resources. This information is written out into a compressed trace, which is then consumed by our version of SimpleScalar. Access to this trace file is through a common API and library to simplify software development. We built a customized text section binary annotation that passes instruction attributes and primitive debug information from the IA64 compiler to the simulator. This is patterned after Intel’s Flexible Annotation library [9], although much more simpler.

4.2 Simulator

Our IA64 simulator is derived from SimpleScalar 3.0 [2], but has been radically modified to handle tracing the peculiarities of IA64 [18, 8]. We model the register stack engine, including injecting RSE spills and reloads with the necessary pipelining stalls, dependencies across software pipeline rotation, and data



#	INST	P	IQ PREDICTED	IQ RECOVERY STATE
1	mov r33=1		mov t1=	mov t1=
2	(p6) add r33=1,r33	T	add t5=1,t1	(p6) add t5=1,(t1,) <t1>
3	(p7) sub r33=2,r33	F	-	(p7) sub t11=2,(t5,t5) <t5>
4	(p8) shl r33=r33,3	F	-	(p8) shl t12=(t5,t11),3 <t11>
5	st []=r33		st []=t5	st []=(t5,t12)

normal data-flow
recovery data-flow
Original Code
Predicate Predicted: Expanded IQ State

#	COR	IQ STATE
1		mov t1=
2	T	add t5=1,t1
3	F	-
4	F=>T	(p8) shl t12 =(t5,t11),3 <t11>
5		st []=(t5,t12)

Mispredict Example #1 (recover at 4)

#	COR	IQ STATE
1		mov t1=
2	T	add t5=1,t1
3	F=>T	(p7) sub t11=2,(t5,t5) <t5>
4	F	(p8) shl t12=(t5,t11),3 <t11>
5		st []=(t5,t12)

Mispredict Example #2 (recover at 3)

Figure 6: Example of register renaming for selective replay predicate misprediction recovery. The top left is the original code, and the top right shows the instruction queue state for the original code. On the bottom shows two different misprediction recovery examples. Dashed-lines represent normal data-flow, while solid lines represent recovery data-flow. Note in misprediction examples, the normal data-flow reaches into the recovery region. Also note for example #2, at instruction #5, how recovery qualified false instructions effectively turn into a move to propagate the prior definition of `r33` so that it can reach its real use.

prefetching. SimpleScalar's generic branch predictor was extended to support counted loops, our predicate predictors and the Itanium Branch-Address-Predictor. We pay special attention to functional unit latencies and resource modeling to match the compiler scheduling model. The register dependency machinery was also substantially enhanced to support predication and various predicate prediction models. A summary of the architectural parameters modeled during simulation for our predicated configurations is given in Table 1. The baseline and Wang results are for an out-of-order architecture identical to this, except that the branch predictor tables for these baseline architectures are twice the size as those used for the predicate prediction architectures, which are shown in Table 1. This is to normalize the amount of prediction state between the models using a predicate predictor and the baseline. We chose a large meta chooser predictor with local (PAp) [21] and bimodal history as the baseline configuration. This does better than an earlier study with global/local meta-chooser, and about as well as global/bimodal meta-chooser. Using only local information allows us to perform multiple branch predictions in parallel without having any history dependence between prior predictions.

Our architecture models the Itanium pipeline depth with the addition of three additional pipeline stages between rename and execute to represent the added complexities of out-of-order execution. The pipeline in Figure 4 clarifies this.

4.3 Benchmarks

We use nine benchmark programs from Spec2000 Integer and Floating-Point. These programs represent all Spec2000 programs that compile and trace correctly in our modified environment. They were all compiled with a version of Intel's IA64 Compiler (Electron) modified for our environment. Some enhancements we have made are binary annotations, and a improved if-conversion region picker. All benchmarks use the reference input set, and in the case of `vpr` the "place" input was used.

For each benchmark, we compiled three sets of binaries. The first binary has no if-conversion to be used as a comparison baseline. The second binary contains conditional-writer (IA64) predication used with the predicate prediction and Wang's select-μops [20]. The third binary contains phi-predication [4] and is used to compare against the conditional-writer predication.

Because we are dealing with three binaries, we need to make sure they functionality execute for the same amount during simulation. To provide this, we performed fast forwarding using SimPoints [19] and determined when to stop simulation based upon the Nth and Mth procedure call invocation. We limit our simulations to at most 300 million instructions, but we make sure that all binaries, for the same program, start at the same function call invocation and end at the same functional call invocation. The number of instructions ended up being simulated for each binary is shown in Figure 2.

Figure 8 measures the precent of the executed code that was predicated due to if-conversion. On average 7.6% of a program's execution was inside if-converted code. `bzip`, `twolf` and `vpr` contain the most if-conversion. This measurement does not include predicated code generated outside the compiler's if-conversion routines, which occurs primarily from IA64 floating-point divides (and square roots) and library code.

Trace	Start Point	End Point	If-Conversion	Trace Length	Length change
164.gzip	function: longest_match	function: longest_match	None	299999974	0.00%
			Conditional-Writer	308150184	2.72%
			Select	280294174	-6.57%
175.vpr	function: read_arch	function: get_non_updateable_bb	None	299999926	0.00%
			Conditional-Writer	280030525	-6.66%
			Select	281682898	-6.11%
177.mesa	function: general_textured_triangle	function: sample_1d_linear	None	305123237	0.00%
			Conditional-Writer	296756476	-2.74%
			Select	297920347	-2.36%
179.art	function: match	function: reset_nodes	None	267860087	0.00%
			Conditional-Writer	285006398	6.40%
			Select	285010952	6.40%
183.equake	function: phi0	function: smvp	None	258547696	0.00%
			Conditional-Writer	261602566	1.18%
			Select	261421846	1.11%
186.crafty	function: SwapXray	function: FirstOne	None	299999966	0.00%
			Conditional-Writer	296844214	-1.05%
			Select	297980972	-0.67%
197.parser	function: region_valid	function: ppmatch	None	297341672	0.00%
			Conditional-Writer	299925970	0.87%
			Select	299999902	0.89%
256.bzip2	function: spec_putc	function: fullGtU	None	299207422	0.00%
			Conditional-Writer	298802221	-0.14%
			Select	299999983	0.26%
300.twolf	function: ucxx1	function: new_dbox_a	None	299999954	0.00%
			Conditional-Writer	286907180	-4.36%
			Select	289647451	-3.45%

Table 2: Trace Characteristics: Because predication may introduce additional instructions in the trace, we synchronized execution between known start and end points.

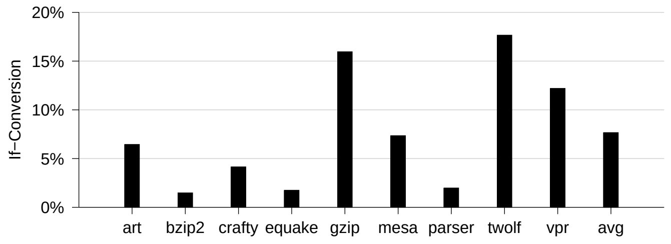


Figure 8: Percent of execution from if-converted regions.

5. COMPARISON

In this section, we first compare different design alternatives to predicate prediction. Subsequently we examine the different predicate prediction approaches. All speedup results are normalized to the same baseline out-of-order architecture without if-conversion with a branch predictor double the size of the architecture supporting predication. Equalizing the overall predictor area is meant to provide a fair comparison between the area for the predictors in the baseline and the predicate prediction architectures.

Three binaries are generated and used for all of the results. The first uses no if-conversion, the second uses standard conditional-writer if-conversion targeting mispredicted branches, and the third uses Phi-predication described in [4].

5.1 Alternative Strategies

In this section we compare different design alternatives to predicate prediction. Figure 9 provides the speedup comparisons between the initial execution models. First we contrast a less efficient, but straight forward design, for dealing with the multiple definition problem. This approach stalls the pipeline in the rename stage until the multiple definition problem is resolved. More specifically a predicated instruction waits before the register renamer until its guarding predicate value has been defined via a writeback. The performance penalty with this approach (stall-only) is caused by stalling the fetch pipeline shows an overall slowdown of -11.8%.

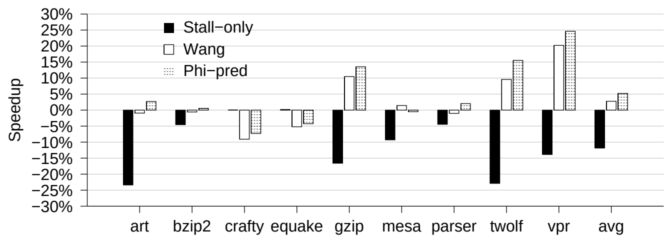


Figure 9: Alternatives to Predicate Prediction: Speedup normalized against No-predication.

Next we consider the prior technique by Wang et.al. [20] described in related work in Section 2.1 . This is a better solution that avoids the predicate early-evaluation stall and serialization of qualified simultaneous writers, by using a special select-μop to choose between different qualified writers. The performance is seen in Figure 9 under “Wang” is clearly better than “stall-only” with an average speedup of 3%. We can see there is some performance penalty due to inserting the select-μop in the evaluation path for a few benchmarks.

Predicated Machine Model	
Fetch Width	Up to 2 Bundles
Issue Width	6 instructions
Function Units	Itanium latencies, resource units
Reorder buffer	RUU: 256, LSQ: 128
L1D	64KB, 4 way, 32B Blk, 2 cycle latency
L1I	64KB, 4 way, 32B Blk, 1 cycle latency
L2 Unified	4MB, 4 way, 64B Blk, 12 cycle latency
DTLB	128 entry, 30 cycle miss penalty
ITLB	64, 30 cycle miss penalty
Memory	300 cycle latency
Branch Pred	16K meta chooser between bimodal (16K entry) and local 2-level table (16k entry); 32 Return Address Stack; 512 BTB; 12 cycle miss penalty Br mispred delay: 12 cycle
RSE	Lazy spill and reloads, model register stack
select-μop/phi Wang/Phi only)	1 cycle latency
Stall-only	Predicate resolution: min 7 cycle
Flush	Recovery: 12 cycle
Rename-Replay	Recovery: 7 cycle
Selective-Replay	Recovery: 7 cycle
Predicate-Pred	16K meta chooser between bimodal (16K entry) and local 2-level table (16k entry)

Table 1: IA64 SimpleScalar: This table describes the general characteristics of our trace-driven simulator.

One design issue for "Wang "is building select-μop for the floating-point operations. For the Itanium integer operations, the minimum latency move is a single cycle, however floating point is five due to exception reporting and greater logic complexity. For results in Figure 9, we use a single cycle select-op implementation for both integer and floating point, since the select should be unencumbered by floating point instruction requirements. We did simulate using a five cycle latency. With this change the speedup turns into a slowdown of -0.77%.

The Phi-predicated code as described by proposed by Chuang et.al. [4] can conceptually be viewed as taking Wang et.al.'s select-μops, and having the compiler automatically insert them. This requires an ISA change from IA64. We find a speedup of 5.2% for phi-predication as seen in Figure 10 under "phi-pred".

5.2 Predicate Prediction

Predicate prediction provides the possibility to provide parallel execution of qualified writers, and to eliminate the data-flow dependency and stalls of prior approaches. Further, its fully compatible with IA64 ISA predication. In this section, we describe the performance of using predicate prediction with three recovery mechanisms. These techniques are: flush, rename-replay, and selective-replay. We compare the results of predicate prediction with Wang et.al. and perfect predicate prediction. These results are seen in Figure 10.

Figure 11 shows the number of mispredicted branches for the original (no-predicated) binary and the predicated binary with predicate prediction. The bottom part of the predicate prediction bar shows the mispredicts due to branches, and the top bar shows the mispredicts due to predicate predictions that were actually used.

When using rename-replay the average speedup is 3.0% over no-predication, with the largest speedup being 12.7% for `vpr`. This model tends to track "perfect" especially on `gzip`, `twolf`, and `vpr`. These benchmarks have a larger percentage of execution from if-converted code as seen in Figure 8. In addition, Figure 11 shows that we are trading off reducing expensive branch mispredictions for increased predicate mispredictions.

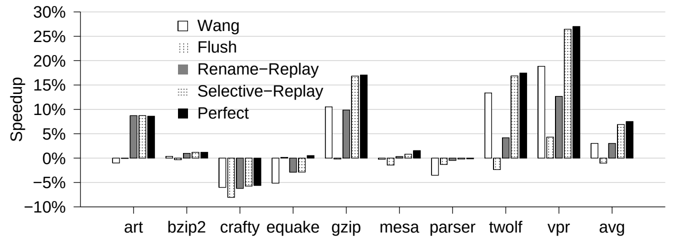


Figure 10: Predicate Prediction Speedups: Normalized against No-predication with double branch predictor capacity.

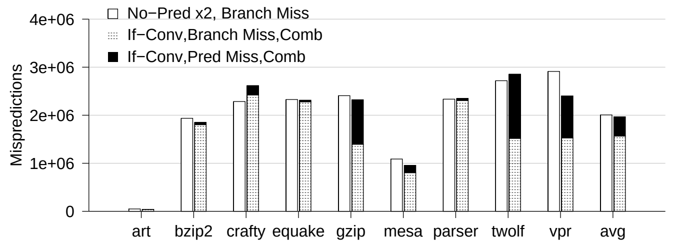


Figure 11: Distribution of predictor mispredictions. First bar is the number of branch mispredictions w/o if-conversion. Second bar shows the number of branch mispredictions and predicate mispredictions stacked.

Results for the second predicate prediction technique, "selective-replay", are shown in Figure 10. Selective-replay recovers mispredicted instructions and any instructions that depends on it that have been put in the Instruction Queue. We find this to provide a 6.9% speedup on average over no-predication. The results for Perfect predicate prediction suggest that selective-replay, even though it requires keeping the recovery operand and destination tags, provides close to ideal benefits.

5.3 Design Space Exploration

We consider one additional hardware change from our earlier results: modeling broadside predicate register file updates. Broadside predicate register-file saves and restores impose a synchronization penalty, and cause the predicate predictor state to become potentially stale. We now consider the impact of synchronization using the hardware described in Section 3.3.4. When we allow hardware to keep track of two broadside writes, we see a slow down of just 0.1%. Several programs `art`, `equake`, and `gzip` were essentially unaffected due to shallow call depth. `Parser` was most affected with a slow down of 0.4%, most likely caused by its deep recursive call pattern.

6. CONCLUDING REMARKS

In this paper we described the multiple definition problem for an out-of-order architecture with predication, and examined using predicate prediction to solve this. Unlike traditional recovery techniques for branches or load-speculation, predication introduces new problems and benefits that we examine.

We considered two techniques to recover from predicate mispredictions: "rename-replay" and "selective-replay". Rename-replay recovers instructions through the register renamer to repair register dependencies due to predicate misspeculation. It represents the more realistic implementation of the two techniques. This technique obtains a speedup of 3.0% over no-predication with double the branch predictor size, but has speedups of 10% for `gzip`, and 13% for `vpr`. Selective-replay only re-executes the instruction that uses the mispredicted predicate, and any instructions along those dependency paths. Its performance result represents an upper-bound with a speedup of 6.9%, also normalized over no-predication. Overall, rename replay is potentially the more straight forward recovery technique to implement, although selective replay achieves twice as much speedup for some programs.

7. ACKNOWLEDGMENT

We thank the anonymous reviewers for their thorough and insightful comments. This work was funded by National Science Foundation grant No. CCR-0073551 and a grant from Intel Corporation. We especially would like to thank Intel for providing the Electron compiler sources, and their assistance in using it.

8. REFERENCES

[1] E. Borsch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb 2002.

[2] D.C. Burger and T.M. Austin. The Simplescalar Tool Set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, Jun 1997.

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

[4] W. Chuang, B. Calder, and J. Ferrante. Phi-prediction for light-weight if-conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2003.

[5] S. Eranian and D. Mosberger. The Linux/ia64 Project: Kernel Design and Status Update. Technical Report HPL-2000-85, HP Labs, 2000.

[6] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[8] Intel Itanium Processor Reference Manual for Software Optimization, November 2001. http://developer.intel.com/design/itanium/downloads/245474.htm.

[9] Intel Flexible Annotations. http://www.intel.com/software/products/opensource/tools1/perftools.htm.

[10] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.

[11] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, Mar–Apr 1991.

[12] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, October 1998.

[13] M. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, Dec 1996.

[14] 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.

[15] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 138–150, 1995.

[16] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Compaq WRL, June 1993.

[17] M. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Procesors. Technical Report HPL-1999-111, HP Labs, 2000.

[18] H. Sharangpani and K. Aurora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept–Oct 2000.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Languages and Operating Systems*, October 2002. http://www.cse.ucsd.edu/users/calder/simpoint/.

[20] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, February 2001.

[21] T. Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th International Symposium on Computer Architecture*. ACM and IEEE Computer Society, 1993.