Control Flow Optimization Via Dynamic Reconvergence Prediction

Jamison D. Collins*†, Dean M. Tullsen*, Hong Wang†

*Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

†Microarchitecture Research Labs
Intel Corporation
Santa Clara, CA 95052

Abstract

This paper presents a novel microarchitecture technique for accurately predicting control flow reconvergence dynamically. A reconvergence point is the earliest dynamic instruction in the program where we can expect program paths to reconverge regardless of the outcome or target of the current branch. Thus, even if the immediate control flow after a branch is uncertain, execution following the reconvergence point is certain. This paper proposes a novel hardware reconvergence predictor which is both implementable and accurate, with a 4KB predictor achieving more than 95% accuracy for SPEC INT, and larger implementations achieving greater than 99% accuracy.

The information provided from reconvergence prediction can increase the effectiveness of a range of previously proposed performance optimizations, including speculative multithreading, control independence, and squash reuse. This paper also demonstrates a new technique that takes advantage of the dynamic reconvergence prediction information in order to predict a wrong path excursion ahead of branch resolution. On average, 34% of wrong path fetches are eliminated.

1. Introduction

Control flow instructions are those instructions in a program for which execution does not necessarily follow immediately to the next sequential instruction. Because of the increasing latency between the fetch of a branch and its computation, instructions following an unresolved branch are fetched, and often executed, speculatively, meaning they may or may not represent useful work. However, all control flow instructions have a *control reconvergence point*, a place in the program where all paths converge. Execution following this point is certain, and (depending on the architecture) instructions may be fetched, and possibly executed, non-speculatively. For example, the instruction statically following a call usually marks a reconvergence point for all the dynamic instructions within the function call. There have been many different proposals which rely on executing instructions expected to be independent of a hard to predict control instruction: speculative multithreading [18, 10, 19, 9], Dynamic Multithreading [3] (DMT), multipath execution [22, 8], and others.

Despite the plethora of techniques relying on the identification of control independent instructions, there has been no comprehensive study of techniques for dynamically identifying control reconvergence points with high accuracy. Control reconvergence is a concept similar to post-dominators computed by the compiler for certain optimizations. In fact, post-dominators can be identified offline by the compiler, and used to indicate conservative control reconvergence points [11], but that approach has a number of limitations. Such reconvergence points lack information on the dynamic behavior of the program, and will necessarily accommodate rarely or never taken control paths. Additionally, communicating the reconvergence information to the processor core requires non-trivial changes to the underlying instruction set architecture.

A second approach to identifying reconvergence points have been hardware-based approaches which use simple heuristics geared at specific program constructs. For example, in [3], the reconvergence points for loops can often (but not always) be identified by assuming that control reconverges at the instruction following the backwards branch which terminates the loop. Other proposed heuristics are geared at simple `if-then`, or `if-then-else` constructs in a program [5]. Though these schemes can be accurate for a subset of branches, they still fail to identify reconvergence points for a large number of control instructions when program behavior does not follow the expected pattern.

In this paper, we propose a new dynamic reconvergence prediction technique which requires no compiler support, but still achieves extremely high reconvergence prediction accuracy (> 99% accuracy on average for SPEC INT). Unlike previously proposed heuristic-based approaches, our technique remains robust in the face of aggressive compiler optimizations. In fact, we show that a different compiler and different ISA make no difference in the accuracy of our techniques.

We find that branches can be classified into one of four categories, based on a summary of the control flow observed following the branch on previous executions. For example, the dominant such category is *rec below max*, which indicates that, after executing the targeted branch, no instruction "below" (at a greater instruction address than) the reconvergence point executes prior to the reconvergence point. Note that this technique also provides fast identification of reconvergence mispredictions, which for this category is identified by the commit of an instruction "below" the predicted reconvergence PC. Prior techniques typically lacked this capability. For many dynamic control flow optimizations, this is a useful property; for example, with speculative multithreading it allows early release of execution resources.

Our technique adds hardware to the processor's back-end, which profiles the instructions committed by a program to

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

identify a candidate reconvergence point for each of the four categories for a targeted branch. Additional information is collected which allows accurate prediction of the reconvergence point by selecting among these four possibilities. We present data illustrating that this structure can be built both small and fast, making a physical implementation feasible.

This is the first study of the general problem of dynamic reconvergence prediction. Further, the novel technique we propose is the only technique to achieve accuracy comparable to a static compiler mechanism. At the same time, it exceeds the compiler-generated reconvergence points by identifying earlier points without loss of accuracy.

To demonstrate the effectiveness of the novel reconvergence predictor, we first illustrate how this predictor can be applied to previous techniques by achieving additional performance improvement on a processor implementing the Dynamic Multithreading (DMT) architecture. We also show that a compiler-driven control independence execution optimization (squash reuse [11]) can be performed in a compiler independent way without sacrificing any effectiveness.

Second, we demonstrate how the knowledge of control reconvergence points and the analysis performed to determine them conveys to the hardware qualitatively new information about program behavior, thus enabling more effective novel optimizations. In this paper we describe how our reconvergence predictor can be used to create a *wrong path predictor* that can identify speculative excursions onto the wrong control flow path before the mispredicted control instruction is executed. Through a simple modification of our control reconvergence predictor, an initial implementation of this predictor identifies on average more than 40% of excursions onto the wrong control path, allowing more than 34% of wrong path fetches to be eliminated for the studied benchmarks. This has application for power efficiency, multithreaded fetch policies, and accelerated branch correction.

The rest of this paper is organized as follows. Section 2 discusses relevant prior research. Section 3 describes our simulation and evaluation framework. Section 4 describes control reconvergence, and presents existing approaches to identify reconvergence points, and Section 5 presents our proposal for a new reconvergence predictor. We evaluate this predictor by applying it to a variety of potential applications in Section 6. Section 7 concludes.

2. Related Work

The research area of control speculation and control reconvergence has received considerable attention.

A number of techniques provide some form of control misspeculation tolerance. Multithreading [4, 21] is a technique allowing multiple programs to execute simultaneously on a single processor core, allowing threads to continue to make progress overlapped with one thread's misspeculation. Dynamic Multithreading (DMT) [3], at runtime, splits a single program into multiple execution streams at loop and function call boundaries. It is one example of Speculative Multithreading [18, 10, 19], a class of techniques to overlap the execution of predicted future control paths with current execution. Much of the work in this area relies on compiler analysis to identify future spawn points. Others tend to rely on simple strategies to identify likely reconvergence, like loop and procedure call continuations. Although many of these techniques rely on the compiler to identify future reconvergence, not all are constrained to be conservative. For example, [9] introduces the concept of a control quasi-independent point, which they define as a future instruction 95% likely to be on a future control-flow path. Other recent works have proposed the use of control quasi-independent points for instruction prefetching [1, 2].

Falcón et al. present Prophet/Critic Hybrid Branch Prediction [7]. This technique utilizes a traditional branch predictor to generate a sequence of future branch predictions, which are then interpreted to evaluate the accuracy of the current branch prediction. This approach has some similarities to the wrong path predictor we describe in this paper, though it targets a subset of the main thread hints targeted by our predictor.

Prior research has also attempted to mitigate control misspeculation by following multiple possible control paths [22, 8]. Skipper [5] is a technique for deferring the fetch of instructions likely to be on the wrong path, and instead fetching and executing instructions from a predicted independent instruction region.

Finally, techniques have been proposed to salvage some of the work performed on the incorrect control path via squash reuse [17], Control Independence [11, 12, 6], and Register Integration [13].

Both Skipper and DMT propose simple, heuristic-based dynamic reconvergence predictors which were used to enable their particular optimizations.

The research presented in this paper represents the first general, comprehensive study of dynamic reconvergence prediction. The mechanism we propose can be used to increase the effectiveness of many of the above techniques, or in other cases to remove the dependence on the compiler and/or ISA modification.

3. Simulation Methodology

This paper utilizes the SPEC CPU 2000 integer benchmarks as its target benchmark suite. This set of programs exhibits a wide range of control behaviors, including programs with a good mix of conditional branches, indirect calls and indirect jumps, both in terms of quantity as well as prediction rates.

For all simulations in this study, benchmarks are simulated 100M total instructions, with fast-forward distances determined by the SimPoint tool [16]. Simulation regions identified by this tool have been found to be highly representative of full program execution. Programs execute using reference inputs. The binaries simulated are SPEC peak binaries compiled for the Alpha 21264 by the DEC/Compaq cc compiler, provided by Chris Weaver from Michigan. These peak binaries have been very aggressively optimized, including aggressive predication.

Performance results given in Section 6 are derived using the SMTSIM [20] simulator. SMTSIM is an execution driven, cycle accurate processor simulator which models an SMT processor. We model an 8 instruction wide, 8 stage pipeline with 384 integer and 384 FP renaming registers (in addition to 32 of each for each thread context), a 384 entry re-order buffer, and 128 entry integer and FP instruction

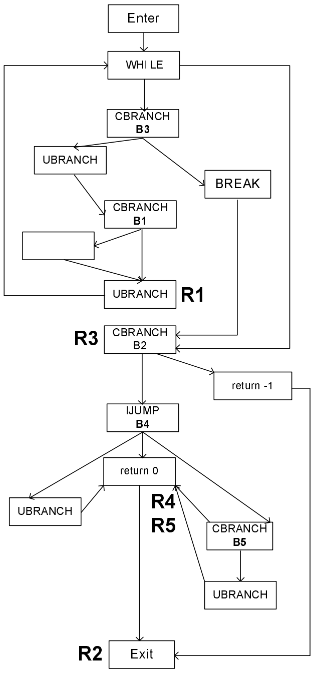


Figure 1. Control flow graph with branches B1-B5 and reconvergence points R1-R5.

queues. A 3 level memory hierarchy is modeled: 32KB 2-way I and D caches, a 512KB 4-way shared L2 cache, and a 4096 KB 8-way L3 cache. Round trip latency to the L2 cache is 10 cycles, to the L3 cache is 20 cycles, and to memory is 150 cycles. Unless otherwise specified, the branch predictor modeled is based on the EV8 branch predictor [15], but we do not model the EV8's banked design. To compensate for this, we model an instantaneous GHR update, and assume private hysteresis for all table entries. Total capacity of the branch predictor is 88 kbits. A 256 entry, 4-way BTB is modeled.

The results in this paper are insensitive to most of these simulation parameters, with the significant exception of the branch predictor, for which we have tried to model an aggressive implementation.

4. Reconvergence

This section describes control reconvergence in more detail. It also describes existing approaches that attempt to identify some kind of control reconvergence for use in some particular control flow optimization.

The control reconvergence point for a particular instruction is defined as a future dynamic instruction such that, regardless of the outcome of any non-exceptional intervening control flow, execution will eventually reach the specified PC.

We start out by providing examples of control reconvergence to give insight into why identifying reconvergence points at runtime on a compiled binary might be difficult, and some of the regularities in program behavior we rely upon to identify these points accurately. Figure 1 illustrates a sample control flow graph with some complex, but common, reconvergence behavior. Branches are indicated by the labels `B1-B5` and reconvergence points indicated by `R1-R5`. Assume basic blocks are laid out physically in the binary in the same vertical order in which they appear in the figure.

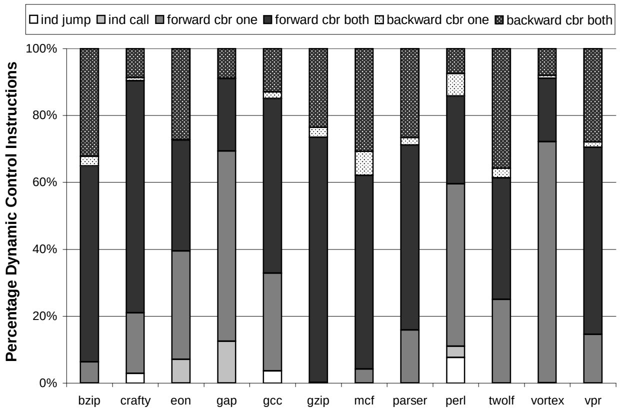


Figure 2. Breakdown of dynamically executed control instructions for the studied benchmarks.

Branch `B1` represents an example of a control construct which transferred to the binary intact; it follows the prototypical `if-then` construct form, in which a forward conditional branch leads to a rejoin point for control, regardless of whether that branch is taken or not. For such constructs, identifying reconvergence points can be achieved through simple pattern matching of the executed instructions.

It is important to note that in compiling and aggressively optimizing a program, the final assembly representation often differs significantly from the high level program code. Thus, conventional control flow constructions do not always translate to conventional control flow. For example, because the `if-then-else` branch at `B3` encapsulates a `break` statement, applying simple pattern matching to it would fail to identify its reconvergence point. Branch `B4` (a `case` construct) has been targeted by code layout optimizations which pack instructions along the "hot path" close together. When a "cold path" is executed, control leads to below the reconvergence point, which then branches back up to the reconvergence point.

Because programs are typically structured as a top-to-bottom flow of program constructs, we typically see reconvergence points appearing below the branch they correspond to. However, this need not be the case. For example, the simple `if-then` construct for branch `B5` has its reconvergence point above the branch itself. For branch `B2` the reconvergence point is given by whichever (static) instruction follows the most recent (dynamic) `call` instruction which invoked this function instance, as multiple function returns are reachable prior to any reconvergence point. While the structure of a compiled and optimized program might not exhibit the regularities of a high-level view, the next section demonstrates that some general regularities do exist at the binary level.

4.1. Survey of Program Behavior

This section presents a brief survey of the benchmarks studied in this work, and the reconvergence behavior they exhibit. Control reconvergence points are identified offline in

benchmark	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
rec below branch	74	492	69	61	1322	83	71	104	131	89	155	32
rec below max	59	487	68	61	1242	79	61	103	128	87	150	31
rec above branch	0	0	0	0	30	1	2	0	4	4	9	0
rec above max	0	0	0	0	30	1	1	0	4	4	8	0
rebound rec	15	5	1	0	80	4	10	1	3	2	5	1
return rec	0	52	11	7	69	8	10	3	20	27	14	0
forward cbranch one outcome	46	288	100	192	2296	139	115	38	610	79	1033	22
backward cbranch one outcome	6	52	2	11	121	12	12	1	25	19	38	2

Table 1. Information on where static reconvergence points are located relative to the branch instructions to which they correspond, for each static branch in the program.

this section, for the purpose of better understanding existing and proposed techniques.

We derive a branch's static reconvergence point using the following methodology. Because we are working with precompiled binaries, we generate a control flow graph from the binary. To include indirect jumps, we profile the code to track all taken targets of all indirect jumps, and add them to the control flow graph. This gives us a very close approximation of the control flow graph the compiler would see. We then use this data to calculate the static reconvergence point for each branch.

The static reconvergence point is guaranteed to be accurate, regardless of actually observed control flow, but is also restricted to obeying infrequently (or never e.g. `if(error)`) executed control paths. In the vast majority of cases, this is the immediate post-dominator of the basic block containing the control instruction.

Figure 2 gives a breakdown of dynamic control instruction instances in the analyzed programs. Conditional branches make up the majority of program control instructions, either exercising a single outcome (*cbr one*) or both outcomes (*cbr both*) during execution. However, all programs execute a significantly larger number of forward branches than backward branches.

Indirect jumps (such as found in `case` constructs) only make up a large fraction of control instructions in `crafty`, `gcc`, and `perl`. However, target prediction for indirect jumps is typically less accurate than conditional branch prediction, and may contribute a significant fraction of total control misspeculations. This heightens the importance of identifying reconvergence points for these instructions if it provides additional opportunity for techniques which mitigate misprediction costs [11, 12, 6, 5].

`Eon`, `gap` and `perl` contain a significant number of indirect calls. However, for indirect calls the reconvergence point is easily predicted conservatively as the sequential instruction following the `call` instruction.

Table 1 shows a breakdown of where the static scheme identifies reconvergence points for the conditional branches and indirect jumps in the studied benchmarks. Quantities are listed in terms of the number of static branches. In the remainder of the paper, "below" refers to an instruction with a higher PC than the reference point, and "above" refers to instructions with a lower PC. Figure 3 shows example control flow graphs for the dominant reconvergence categories.

In this table, *rec below branch* refers to the number of static branches for which the reconvergence point appears below the branch instruction. The category *rec below max* indicates the subset of these branches for which no instruction below the reconvergence point can appear (at the same call level) between the execution of the control instruction and the execution of the reconvergence point. This is the dominant branch behavior, and is exhibited for 93% of static branches. Branches `B1` and `B3` from Figure 1 illustrate this behavior.

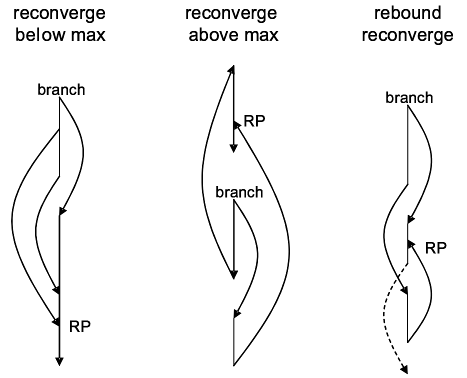


Figure 3. Example control flow graphs for the three dominant reconvergence categories.

Intuitively such control behavior makes sense, as it aligns with how many high level programming languages are structured — as a series of program constructs flowing "downward" to the next construct. In fact, this simple observation is sufficient to construct a highly accurate reconvergence predictor, as we will highlight in the next section. Though not stated explicitly, the heuristics used in both Skipper and DMT also rely on this assumption, in different ways.

Rec above branch refers to those branches and indirect jumps with their reconvergence point above the targeted branch. *Rec above max* is analogous to *rec below max* — no instruction below the reconvergence point (but above the branch) can execute between the branch instruction and its reconvergence PC. Note that while a significantly smaller number of branches have reconvergence points above the targeted branch, those that do are typically in the *rec above max* category. Branch `B5` in Figure 1 illustrates an example of this behavior.

Because some benchmarks contain a significant number of branches in the *rec below* category but not in the *rec below max* category, we assign them the special designation of *rebound rec*, of which `B4` in Figure 1 is an example. For such branches, some control flows (though not necessarily all) initially branch over the reconvergence point, and then later branch backwards to the reconvergence point. Some control paths might still lead directly from the branch to the reconvergence point (without first branching over it). We found such branches to appear commonly in instantiations of `case` constructs.

The final category of reconvergence points we consider is *return rec*, which are branches for which control can lead from the branch to multiple return instructions without first reaching a common reconvergence point. Branch `B2` in Figure 1 illustrates this behavior. For example, `crafty` contains a large dynamic number of such branches due to the frequent calls to `FirstOne()` and `LastOne()`, each of which identify the first or last byte in a word which has a bit set. Both of these functions contain five `return` instructions.

Finally, we note that all programs contain a significant number of static branches which only exhibit a single outcome (*forward cbranch one outcome*, and *backward cbranch one outcome*). Those are forward or backward branches, respectively, that are either always taken or always not taken. Such branches are omitted from the other categories in this table, and from all results depicting reconvergence prediction accuracy. Such branches don't truly "reconverge" as they never "diverge" in the first place, limiting their applicability to reconvergence-based optimizations, and the reconvergence point for these branches can be trivially predicted.

4.2. Existing Reconvergence Schemes

Briefly, we compare prior approaches to identifying control reconvergence points. The three previously proposed techniques we examine are:

Compiler Reconvergence Generation Some past research has relied on the compiler to identify and associate an appropriate reconvergence point with each control instruction [11, 12] by choosing the immediate post-dominator as the reconvergence point for a basic block involving a control instruction.

A basic block `B` is defined as a post-dominator for another block `A` if all simple paths from `A` to basic block `C` (where `C` is any basic block) pass through `B`. Therefore, when `A` is executed, we are guaranteed to reach `B`. The *immediate* post-dominator executes prior to all other post-dominators. This approach is roughly equivalent to the static control reconvergence scheme described above.

Dynamic Multithreading (DMT) [3] Dynamic Multithreading is a technique for dynamically breaking program execution at loop and function call boundaries. When a backwards branch (all of which are assumed to terminate loops) or function call is encountered, a speculative thread is spawned to execute the code following the conditional branch or past the function call. We extend this prediction scheme to implicitly predict the reconvergence points of all branches contained within the loop or within the function call. This technique predicts most reconvergence points accurately, but much more conservatively than even the static scheme, as we demonstrate in Section 5.4. The original form of this scheme presented in [3] does not predict reconvergence points for forward conditional branches.

Skipper [5] Skipper uses reconvergence prediction to identify instructions independent of a recently encountered unpredictable branch. Those instructions are speculatively executed while awaiting resolution of the branch. This technique predicts reconvergence points targeting three program constructs. When a forward conditional branch is encountered, it is assumed that either an `if-then`, or `if-then-else` construct has been encountered. They assume that all such constructs consist of a specific form. Backward branches are assumed to indicate a loop, and the sequentially next instruction (following the backward branch) is chosen as the reconvergence point, as is done in DMT.

Neither the DMT or Skipper heuristic target indirect jump instructions. As stated previously, the reconvergence for indirect calls is easy to predict, and not included in reconvergence prediction measurements for any scheme, for fairness.

5. Building a Better Reconvergence Predictor

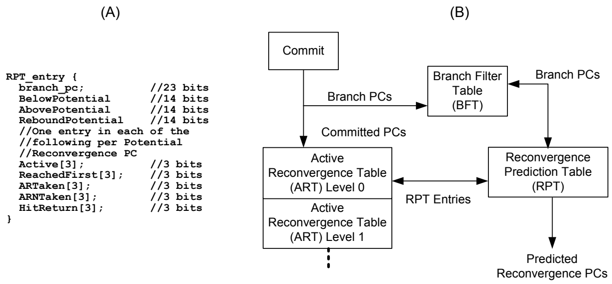
This section presents our scheme for dynamically learning and predicting the reconvergence PC for conditional branches and indirect jumps. Our reconvergence prediction scheme does not rely on heuristics geared towards specific control flow constructs, nor does it require compiler support. Instead, it is a general scheme based on constructing a concise summary of the expected control flow between the executed branch and its reconvergence PC. Because this scheme predicts based on path information (rather than simply predicting a reconvergence PC), it trains quickly and accurately by identifying not only that a prediction was wrong, but exactly where the instruction stream diverged from the expected path.

5.1. Reconvergence Predictor Hardware

To identify and predict the reconvergence points of branches, we add a back-end structure, the Reconvergence Prediction Table (RPT). This table is composed of individual RPT entries, each of which holds information on the reconvergence behavior for a single static branch.

An RPT entry is trained in multiple training phases. Until the branch is executed and committed, its corresponding RPT entry is inactive. After the branch is committed, it enters an active checking/training phase. An RPT entry is only updated in response to instructions committed at the same call level as the branch instruction which activated it. Therefore, at times when the main thread executes at a greater call level than the training branch, the RPT entry is temporarily inactivated (but another instance of the same branch may in fact be active and training the RPT entry). When control returns to the same call level as the training branch, the entry is again activated. When the function containing an active RPT entry returns, that entry is inactivated.

A simple (or idealized) reconvergence predictor can be implemented by broadcasting committed PC addresses to all entries (training only those entries which are currently active) of a potentially large table, by enabling each RPT entry to directly track the call levels at which it is currently active. Because this is the first proposal of this kind, we seek to understand the limits of this reconvergence prediction approach. Thus, we initially assume this type of implementation. The impact of more realistic assumptions is explored in Section 5.6.1.


(A)
(B)
Commit
Branch Filter Table (BFT)
Branch PCs
Branch PCs
Committed PCs
Active Reconvergence Table (ART) Level 0
Active Reconvergence Table (ART) Level 1
RPT Entries
Reconvergence Prediction Table (RPT)
Predicted Reconvergence PCs

```
RPT_entry {
  branch_pc;              //23 bits
  BelowPotential          //14 bits
  AbovePotential          //14 bits
  ReboundPotential        //14 bits
  //One entry in each of the
  //following per Potential
  //Reconvergence PC
  Active[3];              //3 bits
  ReachedFirst[3];        //3 bits
  ARTaken[3];             //3 bits
  ARNTaken[3];            //3 bits
  HitReturn[3];           //3 bits
}
```

Figure 4. Implementation of our reconvergence predictor.

5.2. Reconvergence Training

As Table 1 indicates, there are four primary reconvergence behaviors exhibited by programs (*rec below max*, *rec rebound*, *rec above max* and *rec return*). The approach taken by the reconvergence predictor we implement is as follows: (1) Assume that every reconvergence point fits one of the four dominant categories. (2) Observe program behavior to determine an appropriate candidate reconvergence PC for *each* of the targeted categories. (3) Select the reconvergence PC to be predicted from among the candidate points.

Of the four targeted categories, three of them specify a specific reconvergence PC. For *return reconvergence* branches, the actual reconvergence PC value is given by the top entry of the return address stack. The current best guess for each of the other three categories are referred to as *BelowPotential*, *AbovePotential*, and *ReboundPotential*. Figure 4A indicates the full contents of our most aggressive RPT entry configuration, and the individual fields which will be described below. Less aggressive implementations which target a subset of these reconvergence categories, and have correspondingly less area, are also possible.

Next, we describe the training algorithm for each of the potential reconvergence PCs explicitly tracked in an RPT entry. Each RPT entry contains one *active* bit per potential reconvergence PC (3 bits total). When the targeted branch is executed, each of these bits is set, and the corresponding potential reconvergence PC is a candidate for update. After matching or updating a particular potential Reconvergence PC, the corresponding *active* bit is cleared, and that potential reconvergence PC cannot be updated until after the next execution of the targeted branch.

Figure 3 shows example control flow graphs targeted by the three categories below.

Rec below max reconvergence The vast majority of branch instructions have reconvergence points in the *rec below max* category. We train the corresponding potential reconvergence PC (*BelowPotential*) as follows: (1) When a branch is executed for the first time, initialize the *BelowPotential* to the sequentially next PC following the branch. (2) On all future executions of the branch, the *BelowPotential* becomes active, and may be updated in response to committed PCs. Begin monitoring the PCs committed on behalf of this RPT entry. (3) If the *BelowPotential* PC is committed, the *BelowPotential* becomes inactive. (4) Else if a PC below the *BelowPotential* is committed, update the *BelowPotential* to the committed PC, and the *BelowPotential* becomes inactive.

One of the implications of the above training algorithm is that the *BelowPotential* can only be updated due to a taken forward branch. It also converges quickly, because every execution of the branch results in either a correct prediction (when the reconvergence PC was executed), or an unexpected control flow which leads to a forward branch over this point, causing an update with a better prediction. Thus, the total number of reconvergence mispredictions for a single branch is bounded by the actual control flow, typically to a small number.

Rec above max reconvergence The *AbovePotential* is trained similarly to the *BelowPotential*, except it is only updated in response to PCs committed which are above the targeted branch and below the current *AbovePotential* value. When a branch is first executed, the *AbovePotential* is initialized to an invalid value, and will be updated by the PC of the first instruction executed above the targeted branch. This potential reconvergence PC is guaranteed to ultimately yield the correct reconvergence PC for branches in the *rec above max* category.

Rebound Reconvergence For branches in this category, the reconvergence point is a PC below the branch, but some (though not necessarily all) control flows lead to below the reconvergence point, and then "rebound" back up to the reconvergence point. This potential reconvergence PC is trained as follows: (1) Initially (and each time after the *BelowPotential* is updated), the *ReboundPotential* is given the value of the static instruction following the branch. (2) After the *BelowPotential* has been executed, observe for the execution of any PC below the targeted branch, below the *ReboundPotential*, and above the *BelowPotential*. If such a PC is committed, update the *ReboundPotential*, and the *ReboundPotential* becomes inactive. (3) If the *ReboundPotential* is ever executed (before or after the *BelowPotential*), it becomes inactive.

5.3. Making Predictions

The above algorithms train the potential reconvergence PCs for a branch. However, we also collect additional information to enable a correct selection of the reconvergence PC from among these.

Hit Return For each potential reconvergence PC, a *Hit Return* bit is added. Initially (and whenever the corresponding reconvergence PC is updated) this bit is cleared, indicating that the potential reconvergence PC is always reached prior to a return instruction. If a return ever occurs before a potential reconvergence PC is executed, the corresponding *HitReturn* bit for that potential reconvergence PC is set.

Always reaches For each potential reconvergence PC, two "always reaches" bits are added (*ARTaken* and *ARNTaken*. Initially (and whenever the corresponding reconvergence PC value is updated) these bits are set, indicating that the potential reconvergence PC is always reached when the branch is taken or is not taken. If, for any execution of the targeted branch, a second dynamic instance is committed before reaching the corresponding potential reconvergence PC, the AR bit corresponding to the first outcome of this branch is cleared.

Reaches first More than one potential reconvergence PC may identify a valid reconvergence point for a particu-

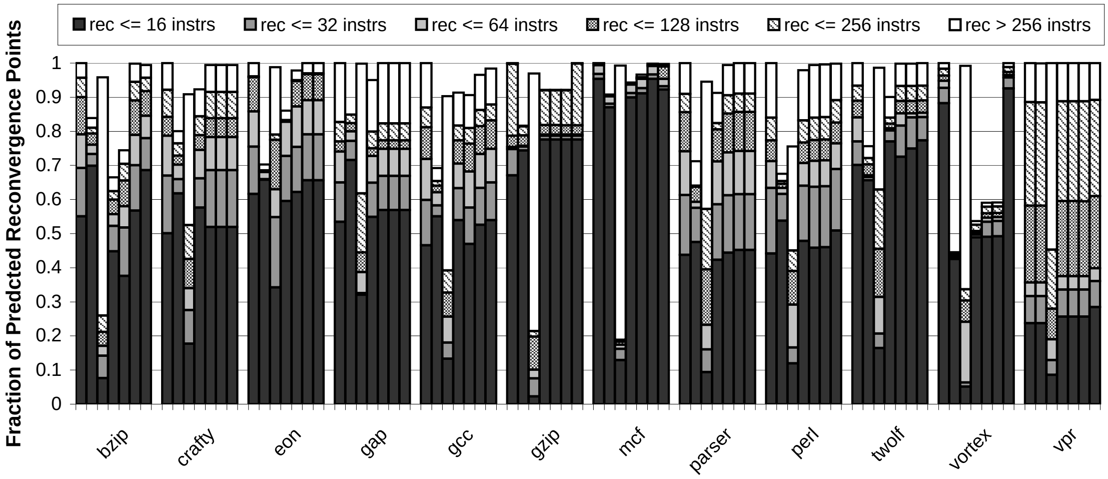


Figure 5. Comparison of reconvergence prediction rates for (left to right in each group of bars) static, Skipper, DMT, and four instantiations of our predictor.

lar branch. For example, both the *BelowPotential* and the *AbovePotential* might be executed following each execution of some branch. For such branches, the superior reconvergence point is the PC which is reached first between the two. For each potential reconvergence PC, a *ReachedFirst* bit is added. Initially, and whenever any potential reconvergence PC is updated, all *ReachedFirst* bits are set (indicating that that potential reconvergence PC is reached prior to all others). Following the execution of the targeted branch, whenever a potential reconvergence PC is executed, if it is the first reconvergence PC executed for this dynamic instance of the targeted branch, the *ReachedFirst* bits for the other two potential reconvergence PCs are cleared. In this way, if a particular potential reconvergence PC is always reached first, ultimately, only it will have a *ReachedFirst* bit set.

Using the above information, a reconvergence PC is predicted for a conditional branch or indirect jump using the following algorithm: (1) If *HitReturn* is set for all potential reconvergence PCs, predict the function return. (2) If some reconvergence PC has its reached first flag set, predict that reconvergence PC. (3) If some reconvergence PC is always reached whether the branch is taken or not taken, predict. that reconvergence PC. (4) If some reconvergence PC is always reached when the branch is either taken or else always reached when the branch is not taken, predict that reconvergence PC. (5) Predict the *BelowPotential*.

5.4. Comparison of Prediction Schemes

Figure 5 compares the accuracy of predicted reconvergence points made by the different reconvergence predictors described to this point. From left to right, the predictors compared are the static scheme, the Skipper scheme, the DMT scheme, and four instantiations of our scheme. Each instance of our scheme targets the same reconvergence categories as the previous scheme, as well as adding a new category. The four instances of our scheme, then, target *below max reconvergence*, adding *return reconvergence*, adding *rebound reconvergence*, and adding *above max reconvergence*.

The DMT scheme, as described in that research, only covers a fraction of branches. Therefore, we extend it, in the most natural way, to make a prediction for each branch, as follows. Backwards conditional branches and calls are still handled as described in Section 4.2; both predict the next static instruction as the reconvergence point. Forward branches and indirect jumps, which are normally ignored by DMT, are predicted to reconverge at the same PC as the innermost loop which contains them. If no loop contains a control instruction, then the predicted reconvergence PC for the containing function is chosen instead.

We classify reconvergence prediction as follows. For each scheme, the height of each bar indicates overall accuracy (i.e. number of correct predictions), and each bar is divided according to the distance in dynamic instructions between the targeted control instruction and each correctly predicted reconvergence point. For our scheme and the Skipper scheme, we consider a reconvergence prediction to be incorrect if the dynamically predicted reconvergence PC is not encountered prior to the reconvergence PC provided by the static scheme.

Because the DMT scheme makes very conservative reconvergence predictions, we define accuracy slightly differently. All predictions of reconvergence at function return are automatically considered correct (distance to reconvergence PC is still accurately tracked). Reconvergence predictions for backwards conditional branches are handled as described above.

Our simplest reconvergence predictor is quite effective in most cases, with very low hardware overhead. For this implementation (since the *BelowPotential* is always predicted as the reconvergence PC), each RPT entry only stores a tag, the *BelowPotential*, and one *active* bit.

While DMT achieves very high accuracy (given the relaxed definition of accuracy), its predictions are not very precise. For example, more than 80% of the reconvergence PCs it predicted for `mcf` were not reached for more than 256 dynamic instructions. In contrast, each instantiation of our predictor chooses an instruction which appears within the next 16 dynamic instructions 80% of the time.

Our most aggressive predictor sees an average mispredict rate of only 0.25%. Even our less aggressive predictors all but eliminate mispredicts, except for `vortex`, which is an outlier benchmark, having a significant number of reconvergence points above the targeted branch.

The relative portion of dark bars in the two graphs indicate that our predictor is consistently predicting earlier reconvergence points than the static scheme, and it is doing so without any loss of accuracy.

5.5. Generality of the Predictor

The preceding analysis demonstrates the accuracy of our predictor, but does not conclusively show that the properties it exploits are not specific to a compiler, or even an ISA. To validate that our technique is indeed general, we run experiments when varying the underlying ISA (IA32), compiler (gcc -O2), and simulation length (full reference execution). Due to an extremely long runtime (> 2 trillion instructions), only the first 200B instructions in `parser` are executed. Results are collected by analyzing programs using Valgrind [14], an ATOM-like tool for IA32.

Overall, we find that the trends identified previously when targeting the Alpha binaries continue to hold. On average, 94% of branches (which exhibit multiple outcomes) are in the *rec below max* category. Because of this, near perfect reconvergence prediction accuracy is achieved for nearly all of the benchmarks, with nine of the 12 demonstrating prediction accuracies of 99.9%, the exceptions being `bzip` (99.0%), `crafty` (99.7%) and `mcf` (98.1%). This indicates that the explored techniques will be transferable to different machine configurations.

5.6. Implementation Issues

Our primary emphasis in this paper is to explore the potential of these techniques. However, a reasonable implementation of this predictor is feasible, even under current technology assumptions. We find that, even with constrained hardware resources, reconvergence prediction accuracy is not significantly reduced.

5.6.1 Realistically Sizing Hardware

If the table is large, it may not be feasible to train the RPT by broadcasting the address of each committed instruction to all RPT entries. The capacity and frequency of access can be reduced by adding two processor structures which can absorb the majority of accesses. These structures are the Branch Filter Table (BFT) and the Active Reconvergence Table (ART), which interact with the processor pipeline as illustrated in Figure 4B.

The BFT serves as a filter to prevent branches which dynamically exhibit only one branch outcome from being entered into the RPT. A significant number of branches have this property, as shown in Table 1. When a branch commits, if it is not already present in the RPT, an entry in the BFT is allocated, and the outcome exhibited by the branch for this execution is recorded in the BFT entry. When that branch is committed in the future and it exhibits the opposite outcome, an entry in the RPT is allocated for the branch. Indirect jumps bypass the BFT and are entered directly into the RPT.

BFT size	Misp Rate	ART size / lvl	ART IP	Misp Rate	ART lvls / CP	Misp Rate
64	9.03	8	1	22.06	1 / 1	4.49
256	2.58	8	4	1.19	1 / 4	2.66
1024	1.53	32	1	2.46	4 / 1	0.63
4096	1.52	32	4	0.52	4 / 4	0.50

Table 2. Overall reconvergence prediction accuracy from varying design parameters individually. Parameters not specified are assumed to be ideal.

The ART is used to temporarily cache the subset of RPT entries actively being trained. Each entry in the ART holds RPT entries corresponding to the branches executed at a particular call level. The ART counter dictates the currently active ART entry. When a targeted branch is committed, the corresponding RPT entry (if it is not already present in the current ART entry) is copied to the current entry. When a call instruction is committed (and enters a new function), the ART advances to the next entry, and any further branches committed will cause their RPT entries to be copied into this new entry. When a return instruction is committed, all RPT entries corresponding to the current call level (within the "current" ART entry) are evicted, and the ART counter is decremented. Note that such an organization permits multiple copies of an RPT entry to be stored in the ART, each at different call levels.

Thus, the only structure that needs to observe the commit stream is a single frame of the ART. When an RPT entry held in the ART is updated by an unexpected main thread control outcome, that modified entry is copied back to the RPT. Utilizing an ART, then, the RPT is only probed when an RPT entry present in the ART is updated. Because the ART holds a (small) subset of entries from the RPT, it can be accessed in a significantly smaller time.

Feasibly sizing BFT and ART structures (the structures accessed with high frequency) does not cost significant reconvergence prediction accuracy compared to an ideal configuration. Note that we observe a reduction in prediction accuracy because of slower training of the RPT, but no entries are incorrectly trained. Table 2 compares the overall reconvergence mispredict rate as the size of both the BFT and ART are varied, changing each design parameter independently. Unspecified parameters are assumed to be ideal. These results assume our most aggressive RPT entry implementation.

The left of Table 2 explores the impact of varying the BFT size between 64 and 4096 entries (all configurations assume 4-way associativity). When a branch is committed, if no RPT entry exists for the branch (and it is not already present in the BFT), an entry is allocated for it using LRU replacement.

We observe the following trends. First, the Branch Filter Table need not be overly large. While having only 64 entries is too few, and reconvergence prediction accuracy suffers, a table of 256 entries achieves nearly the prediction accuracy of 4096 entries. A smaller BFT hurts prediction accuracy because it delays creating an RPT entry for a branch which exhibits one of its branch outcomes rarely. Note that even a large BFT table suffers some accuracy loss because of the need to observe both outcomes for a conditional branch before creating the RPT entry, during which time no reconvergence predictions are made for that branch.

In the middle of Table 2 we vary the number of RPT entries which can be stored at each ART call level. If no space for a new entry exists at the current level, a committing branch is ignored and no RPT entry is copied to the ART. In fact, we observed benefits from not always entering RPT entries into the ART, even when space was available, as always entering RPT entries permitted certain branches (for example, those at the beginning of a function) to habitually occupy spots that would be better given to other branches.

To avoid this problem, we vary the *installation probability (IP)*. When a branch commits, it has probability equal to one over the IP of having its RPT entry inserted into the ART, if space is available. If a branch commits with its RPT already in the ART, then it has a probability equal to one over the IP of retaining its entry; otherwise, the RPT entry is evicted from the ART. This makes it possible to free up room in the ART after all space is occupied.

We find that indiscriminately inserting RPT entries into a small ART can significantly impact prediction accuracy. However, when only entering (or retaining) an RPT entry into the ART with 25% probability, overall misprediction rate improves from 22.06% (when doing so with 100% probability) to 1.19%. In fact, an ART which stores only 8 entries per level, but with an IP of 4 achieves greater prediction accuracy than one with 32 entries and an IP of 1. If the ART can store 32 RPT entries per call level, prediction accuracy is nearly ideal.

The right of Table 2 limits the maximum number of call levels which can be simultaneously tracked within the ART. When a call instruction is committed, if not all call levels in the ART are occupied, a new entry is allocated for the function just entered by the main thread. If no entries are available, the entry corresponding to the lowest call level in the ART is preempted for the new call level. As with above, we found benefit from only handling new call levels selectively. When a call is executed, an ART entry is allocated only with probability equal to the inverse of the *call probability (CP)*. If a call level is ignored, then no RPT entries are entered into the ART for any branches at that call level. Note that further calls from within that function may still be handled. Limiting the number of call levels which can be tracked in the ART impacts prediction accuracy if the number of levels is less than four, but provided no significant impact otherwise. Varying the CP did not have much impact.

Considering configurations which model realistic implementations for all parameters, we find that a prediction rate over 95% is achieved when utilizing a 256 entry BFT and a 2-level, 32-entry ART, requiring approximately 1.5KB of storage. High prediction accuracy is achieved under realistic hardware constraints because RPT entries are both trained quickly, and because their ability to generate accurate predictions is tolerant of incomplete program behavior information – important program behavior will be repeated often enough that the training still converges quickly. Also, while training is still incomplete, many predictions using the intermediate knowledge will still be correct.

5.6.2 Additional Implementation Optimizations

Additional factors which weigh in favor of a feasible implementation are a limited necessary RPT capacity, and techniques to reduce the frequency of ART access.

The RPT itself need not be overly large. For most benchmarks only a small number of total table entries is needed — for all benchmarks other than `crafty` and `gcc`, the RPT needs a capacity of fewer than 256 entries to track all branches which dynamically exhibit multiple branch targets. The working set of important branches is smaller. For our most aggressive predictor, a reasonable implementation requires only 80 bits per RPT entry – a 23 bit tag (supporting a 32MB binary), the three potential reconvergence PCs represented as 14-bit differential offsets, and 12 total bits for the *ReachedFirst*, *ARTaken*, *ARNTaken*, and *HitReturn* flags. A 256 entry RPT with table entries of this size requires only 2.6KB. Because writes into this table are infrequent (and reconvergence prediction does not impact correctness), accesses to it can be pipelined without bypass paths.

Second, we need not probe the ART for every committed instruction. An RPT entry can only be updated based on the target of a control instruction, which is simply the PC of the next committed instruction. This reduces accesses to only 13% of dynamic instruction executions. On a processor achieving the incredible performance of 8 instructions committed per cycle, this amounts to only one ART access per cycle. In fact, the number of ART probes can be even further reduced by accessing it only in response to *taken* branch instances, with a small change in the address comparison mechanism. This optimization would reduce the number of ART accesses to 9% of dynamic instructions.

This section demonstrates techniques that provide highly accurate dynamic reconvergence prediction. But reconvergence prediction is not a goal in itself. The next section demonstrates that it is an effective tool to either enhance or enable a varied set of control flow optimizations.

6. Applications of Reconvergence Information

This section demonstrates a few of the potential applications of dynamic reconvergence prediction. We demonstrate its effectiveness in making two previously proposed techniques, DMT and squash reuse, more effective. For DMT, we demonstrate how greater performance is achieved through knowledge of more complete reconvergence information. We also show how squash reuse can be implemented relying only on dynamic reconvergence prediction (as opposed to compiler analysis), yet achieve equal performance gains.

Our reconvergence prediction also provides information of a qualitatively new nature. We illustrate this by presenting initial results of a new optimization geared at reducing the fetch of wrong path instructions — a technique we call wrong-path prediction.

6.1 DMT

Dynamic Multithreading [3] is a technique for breaking a program into multiple pieces, then executing these regions in parallel on a simultaneous multithreaded processor. Large backing structures known as Trace Buffers are employed to hold speculatively executed instruction results, and instructions are re-dispatched from these structures if they executed with invalid inputs. Programs are split across loop and function call boundaries. The processor is augmented with a back-end reconvergence stack, which tracks

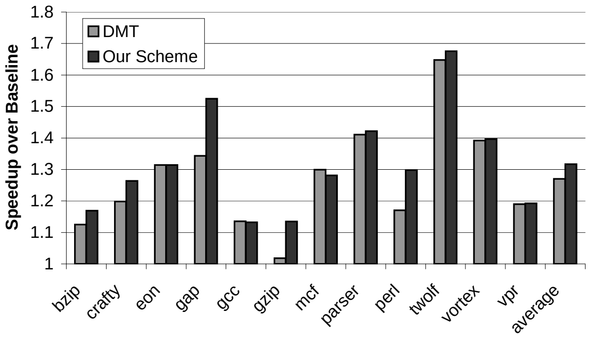


Figure 6. Speedup from basic DMT and DMT enhanced with our reconvergence predictor.

those function calls and loops for which spawning a speculative thread is expected to yield performance gains, avoiding spawns for threads expected to hurt. When the oldest (i.e., non-speculative) thread, which represents architected state, matches the PC of the next oldest thread, instruction results from the second-oldest thread are incorporated.

We next compare the performance of a baseline DMT implementation and one which utilizes our reconvergence predictor. We utilize the most aggressive predictor described in Section 5, which assumes an ideal BFT and ART. We extend the basic technique by permitting speculative thread spawn in response to indirect jumps, for which our technique accurately predicts the reconvergence point. The base processor we model is given in Section 3, and we implement a faithful implementation of the DMT threading technique given in [3].

Figure 6 shows the speedup over a baseline processor achieved by a basic form of DMT compared to that from a form of DMT enhanced with our reconvergence prediction. In many cases we achieve further performance gains over the original DMT. `Perl` and `gap` nearly double the DMT gains, and `gzip` gets significant speedup where DMT got none. These gains result primarily from more accurate thread spawning (the ability to target branches whose reconvergence PC was mispredicted in the baseline scheme). A thread spawned from a future PC that will never be reached prevents other, more speculative, threads from being matched in the main thread as well until it is squashed. This increases thread context occupancy, and results in unnecessary speculative thread squashes.

6.2 Squash Reuse

Prior research has identified that a significant number of instructions executed under incorrect control dependences actually perform useful work which can be committed to architected state. One such technique known as *squash reuse* has been proposed to exploit this fact by salvaging the work of control-incorrect (but data-correct) instructions, the results of which would otherwise be entirely discarded.

The approach described in [11] implements squash reuse through a heavy reliance upon reconvergence points. When a control misspeculation is detected, the processor's instruction window is searched for the reconvergence point of the mispredicted control instruction (determined by the compiler). Instructions prior to this reconvergence instruction are discarded and removed from the instruction window. Instructions following the reconvergent point are retained.

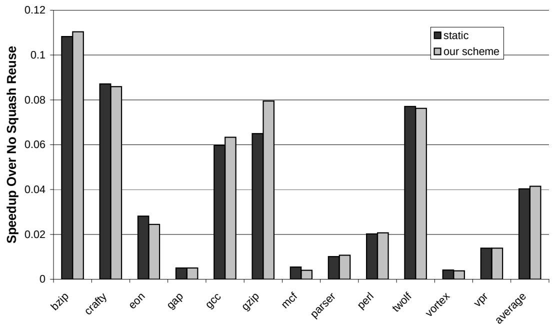


Figure 7. Squash reuse speedup when using static reconvergence and our reconvergence predictor.

As the processor restarts fetch from the correct control path, any change to the data input values consumed by the squashed (but retained) instructions is monitored. If their input data values are invalidated, they are forced to re-execute. As the processor re-fetches the squashed instruction PCs, those that remain in the instruction window without having been invalidated are directly incorporated into the processor state, bypassing their execution.

A complication arises from using our more aggressively derived reconvergence points as compared to static reconvergence points – they only represent the subset of possible execution paths observed when following correct path execution. If control follows a wrong path that is never exercised by the correct path execution, our reconvergence predictor will not be trained for it (because it is always trained along the correct path).

We address this problem by slightly modifying the window search for the reconvergence instruction. We look for three types of convergence instructions, and if necessary, choose the most appropriate. If we successfully match the predicted reconvergence PC from our predictor to an instruction within the instruction window, then that instruction is selected as the reconvergence instruction. If not, then we choose the first PC that is below the reconvergence PC (Section 4), or any return from the current call level. This amounts to allowing some simple just-in-time speculative training of the reconvergence predictor on the current wrong path.

Figure 7 illustrates the performance impact of squash reuse. Results are shown as speedup over a processor without squash reuse, and we show results for static reconvergence prediction, and our dynamic scheme. The static reconvergence bars represent the closest approximation to the original compiler-directed mechanism. We observe that our prediction scheme performs comparably to the static scheme without requiring compiler support or ISA modifications. It achieves a speedup of 4.1% over the baseline, which actually exceeds speedup from the static scheme by a tiny margin. The number of squashed instructions which are reclaimed is fairly consistent among the two modeled schemes.

6.3 Wrong path prediction

As noted in Section 6.2, the reconvergence points identified by the proposed predictor are only trained in response

1072-4451/04 $20.00 © 2004 **IEEE**

Wrong Predictions	8231	6587	6614	3841	20149	2707	3122	12070	7512	9653	6313	9017
Correct Predictions	372857	461436	88814	293119	707846	57735	27463	194961	833928	397459	66355	192381
CBR WP Detected	21.1	50.4	11.5	62.2	64.8	10	5.6	50.9	41.2	33.4	75.8	27.2
IJump WP Detected	-	100	43.2	47.5	59.5	40	25	-	82.5	-	50	-
ICall WP Detected	-	68.2	-	-	82.6	100	100	-	87.2	-	97.1	-

WP Fetch	176M	78M	36M	40M	87M	109M	72M	66M	87M	146M	9M	178M
WP Execute	82M	36M	20M	20M	46M	70M	26M	32M	51M	64M	6M	108M
WP Fetch (throttle)	155M	49M	30M	16M	39M	103M	69M	36M	33M	106M	3M	134M
WP Execute (throttle)	75M	23M	17M	12M	24M	67M	24M	16M	23M	44M	2M	80M
Benchmark	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr

Table 3. Accuracy and effectiveness of our wrong path predictor. Results marked (throttle) correspond to a processor which stops fetching when it predicts it is on the wrong path until all fetched control instructions are resolved (or misprediction detected).

to instructions committed along the correct path, and wrong path execution often fails to reach this predicted reconvergence point. For example, a return instruction might be encountered before a predicted reconvergence PC. If the reconvergence PC is correct (our predictor typically is) then the return instruction must be the result of a control misprediction.

This phenomenon creates the opportunity for *wrong path prediction*, in which the processor predicts that it is currently on a mispredicted control path (without necessarily identifying the responsible control instruction). Such predictions can enable several potential optimizations, such as (1) reducing power by ceasing fetch for predicted wrong path instructions, (2) improving fetch bandwidth utilization in a multithreaded processor by biasing against threads predicted to be on the wrong path, and (3) improving performance by reversing the prediction of an earlier, low confidence branch.

The basis for our wrong path predictor is the observation that updates to RPT entries are infrequent. Therefore, our predictor simply consists of a test at fetch time — if the following instructions were to be committed, would they result in a change to an RPT entry? If so, predict that we are on the wrong path. Note that this predictor is possible to implement precisely because reconvergence points identified by this scheme are more aggressive than those determined by a compiler; any reconvergence points determined by a compiler are necessarily conservative enough that they capture even wrong path behavior.

The predictor can be implemented as follows. In the front-end a small and fast secondary RPT table is added. When a control instruction is fetched, the corresponding RPT entry is copied from the larger, backing RPT into this small RPT. This smaller RPT monitors *fetched* (rather than committed) instructions and detects if a fetched instruction, were it committed, would cause an RPT entry update. This section assumes the aggressive EV8-like branch predictor described in Section 3.

In most cases, this simple implementation of the wrong path predictor yields good results. However, in some cases (notably `vpr`), only a small number of wrong path excursions are predicted. This is because, over the regions which incur a large number of branch mispredictions, correct path execution exhibits similar control flows to those along the incorrect path. Thus, there is little "unique" behavior to distinguish wrong path behavior from correct path behavior.

In order to improve the effectiveness of this technique, we slightly extend our RPT training algorithm in the following ways. Each RPT entry which corresponds to a conditional branch is extended to record whether both taken and not taken outcomes have been exhibited by the branch. These added flags are updated at commit time just like the other state in an RPT entry, and also play a similar role for wrong path prediction.

Additionally, because infrequent control paths will eventually be followed, we modify our algorithm to prevent RPT entries from being trained for infrequent paths. We accomplish this in two ways. First, an internal counter is added to each RPT entry. Each time the predicted reconvergence PC is reached, this counter is incremented. If committing (not fetching) a particular instruction would result in an RPT entry update, this counter is checked. If the counter is below 32, the RPT entry is updated. Otherwise, the counter is halved, and the update is ignored. Wrong path predictions are still generated in response to fetched instructions regardless of the counter value.

Second, a technique is introduced to "dislodge" RPT entries which have become trained to infrequent program behavior. We record the number of times each branch is mispredicted, and how often the wrong path predictor detected the misprediction ahead of time (in most programs a very small fraction of branches are responsible for the vast majority of control misspeculations). If, after a branch has been executed at least 100 times, only 5% or fewer of the control misspeculations are detected for some particular branch, we conclude that some RPT entries are overly trained. The next time that branch is mispredicted, we reset all RPT entries fetched between the fetch of the mispredicted branch and that branch's resolution. Because this technique can increase the number of incorrect wrong path predictions (predicting the processor is on the wrong control path when it was actually on the correct path), this optimization is only applied when the overall wrong path prediction accuracy is at least 90% (over a recent period of 128K cycles).

Table 3 indicates essential statistics for the implementation of our wrong path predictor.

In the table, *wrong predictions* indicates the number of times the processor erroneously predicted that control was on the wrong path, and *correct predictions* indicates the number of times the processor correctly predicted it was on the wrong path. *CBR WP Detected*, *ICall WP Detected*, and *IJump WP Detected* record the percentage of time that a mispredicted conditional branch, or misspeculated indirect call or indirect jump was the initial cause of the processor being on the wrong path, and the wrong path predictor detected that the processor was on the wrong path before that instruction was resolved.

WP Fetch and *WP Exec* indicate the total number of instructions fetched and executed while on the wrong control path. *WP Fetch (throttle)* and *WP Exec (throttle)* indicate the total number of instructions fetched and executed while on the wrong control path in a processor implementing the first of the proposed optimizations outlined above, ceasing fetch following a prediction that the processor is on the wrong path.

The results indicate that this simple form of wrong path prediction can be effective. In all cases, a wrong path prediction accuracy of more than 90% (and in most cases significantly higher) is achieved. Ceasing instruction fetch at such times can significantly reduces the number of wrong path instructions fetched and executed. On average, we observe a 34% reduction in wrong path fetched instructions, and four benchmarks (`gap`, `gcc`, `perl` and `vortex`) reduce wrong path fetching by more than 60%. In many cases, that comes with very little cost in false predictions (which will degrade performance).

7. Conclusion

This paper presents a novel approach to dynamic reconvergence prediction. Unlike previous approaches, which used simple heuristics to enable particular instances of control flow optimizations, our technique is highly accurate, and identifies aggressive reconvergence points, but operates entirely at runtime, requiring only a 1.5KB hardware table for most benchmarks. Our simplest reconvergence scheme achieves a reconvergence misprediction rate of 12.5%. It is more accurate than one previous dynamic scheme and significantly less conservative (more timely) than the other. Our most aggressive scheme reduces this rate to only 0.25%, making our predictor nearly as effective (and for some benchmarks, more effective) at providing useful reconvergence points as the compiler.

We apply the reconvergence predictions derived by our predictor to two previously proposed applications — augmenting a DMT processor architecture, and implementing squash reuse. For squash reuse, we are as effective as the compiler, eliminating the need for ISA changes to support the scheme. For dynamic multithreading, we show the potential for performance increases of up to 18% (average 4%).

Additionally, we present a novel optimization known as the wrong path predictor. We find that by observing for potential updates to our reconvergence predictor, we can often identify that the processor is on the wrong control path. We demonstrate a simple optimization geared at saving power by stopping fetch when a wrong path excursion is detected, resulting in a 34% reduction in wrong path fetches.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions. This work was supported in part by NSF grant CCR-0105743 and grants from Intel Corporation.

References

[1] T. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. Shen. Hardware support for prescient instruction prefetch. In *To Appear in Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 2004.

[2] T. Aamodt, P. Marcuello, P. Chow, A. Gonzalez, P. Hammarlund, H. Wang, and J. Shen. A framework for modeling and optimization of prescient instruction prefetch. In *SIGMETRICS*, June 2003.

[3] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 30 November–2 December 1998.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[5] C. Cher and T. Vijaykumar. Skipper: A microarchitecture for exploiting controlflow independence. In *34th International Symposium on Microarchitecture*, Nov. 2001.

[6] Y. Chou, J. Fung, and J. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *International Conference on SuperComputing*, June 1999.

[7] A. Falcón, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet/critic hybrid branch prediction. In *31st Annual International Symposium on Computer Architecture*, May 2004.

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

[9] P. Marcuello and A. Gonzalez. Thread-spawning schemes for multithreaded architectures. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb. 2002.

[10] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *International Conference on SuperComputing*, July 1998.

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

[12] E. Rotenberg and J. Smith. Control independence in trace processors. In *32nd International Symposium on Microarchitecture*, June 1999.

[13] A. Roth and G. Sohi. Register integration: a simple and efficient implementation of squash reuse. In *MICRO33*, Dec. 2000.

[14] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux. In *http://valgrind.kde.org/*, 2004.

[15] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[17] A. Sodani and G. Sohi. Dynamic instruction reuse. In *ISCA97*, June 1997.

[18] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[19] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[20] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[22] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.