Dynamic Predication of Indirect Jumps

José A. Joao*, Onur Mutlu‡, Hyesoon Kim*, and Yale N. Patt*

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

‡Microsoft Research
onur@microsoft.com

***Abstract*—Indirect jumps are used to implement increasingly-common programming language constructs such as virtual function calls, switch-case statements, jump tables, and interface calls. Unfortunately, the prediction accuracy of indirect jumps has remained low because many indirect jumps have multiple targets that are difficult to predict even with specialized hardware.**

This paper proposes a new way of handling hard-to-predict indirect jumps: dynamically predicating them. The compiler identifies indirect jumps that are suitable for predication along with their control-flow merge (CFM) points. The microarchitecture predicates the instructions between different targets of the jump and its CFM point if the jump turns out to be hard-to-predict at run time. We describe the new indirect jump predication architecture, provide code examples showing why it could reduce the performance impact of jumps, derive an analytical cost-benefit model for deciding which jumps and targets to predicate, and present preliminary evaluation results.

I. Introduction

Indirect branches are becoming more common as an increasing number of programs is written in object-oriented languages such as Java, C#, and C++. To support polymorphism [4], which significantly eases the development of large software projects, these languages include virtual function calls that are implemented using indirect jump instructions in the instruction set architecture (ISA). Previous research has shown that modern object-oriented languages result in significantly more indirect branches than traditional languages [3]. In addition to virtual function calls, indirect branches are commonly used in the implementation of programming language constructs such as switch-case statements, jump tables, and interface calls [2].

Unfortunately, current pipelined processors are not good at predicting the target address of an indirect jump if multiple different targets are exercised at run-time. Such hard-to-predict indirect jumps not only limit processor performance and cause wasted energy consumption but also contribute significantly to the performance difference between traditional and object-oriented languages [14]. The goal of this paper is to develop a new technique to reduce the performance impact of indirect jump mispredictions.

II. Basic Idea

We propose a new way of handling hard-to-predict indirect jumps: dynamically predicating them. Our technique stems from the observation that program control-flow paths starting from different targets of some indirect jump instructions usually merge at some point in the program, which we call the control-flow merge (CFM) point. The compiler identifies such indirect jump instructions (called *hammock indirect jumps*) along with their CFM points and conveys them to the microarchitecture through modifications in the ISA. When the hardware fetches such a jump, it estimates whether or not the jump is hard to predict using a confidence estimator [7]. If the jump is hard-to-predict, the microarchitecture predicates the instructions between N targets of the indirect branch and the CFM point.[1] When the processor reaches the CFM points on all N different target paths, it inserts select-μops to reconcile the data values produced on each path and continues execution on the control-independent path. When the indirect jump is resolved, the instructions -if any- that correspond to the correct target address do not need to be flushed from the pipeline. If the jump would have actually been mispredicted, its dynamic predication saves a pipeline flush, improving performance.

III. Conditional vs. Indirect Jumps

Our approach is inspired by the dynamic predication of conditional branches [11], [10], which was proposed to reduce the performance impact of branch mispredictions. However, there are two fundamental differences between the dynamic predication of conditional branches and indirect jumps:[2]

1. There are exactly two possible paths after a conditional branch. In contrast, the number of possible paths after an indirect branch is dependent on the number of possible targets, which can be very large. For example, an indirect jump used to implement a switch statement in the SPEC CPU2006 *perlbench* benchmark has 57 static targets. Predicating a larger number of target paths increases the likelihood that the correct path will be in the pipeline when the branch is resolved, but it also requires more complex hardware and increases the amount of wasted work due to predication since only one path is correct. Therefore, one important question in the dynamic predication of indirect jumps is *how to identify how many and which targets of a jump should be predicated.*

2. The target of a conditional branch is always available at compile time. On the other hand, all targets of an indirect jump may not be available at compile-time due to techniques like dynamic linking and dynamic class loading. Hence, a static compiler might not be able to convey to hardware which targets of an indirect jump can profit from dynamic predication. Another important question, therefore, is *who (the compiler or the hardware) should determine the targets that should be dynamically predicated.*

IV. Why Could It Work?

We first examine code examples to provide insights into why dynamic predication of indirect jumps can improve performance.

A. Virtual Function Call Example

Figure 1 shows a virtual function call (`Inside`) that is responsible for 24% of all indirect jump mispredictions (using a commonly-implemented branch target buffer-based indirect jump predictor) in the SPEC 2006 FP benchmark `povray` (a C++ ray tracing application). This function call has two

Manuscript submitted: 29 April 2007. Manuscript accepted: 17 May 2007. Final manuscript received: 25 May 2007.

[1]N is determined per jump using compile- or run-time cost-benefit analysis.

[2]For the purposes of this paper we assume a conditional jump is always a direct jump. A conditional indirect jump can be treated as an indirect jump with one extra target, i.e. the next sequential instruction.

Posted to IEEE & CSDL on 6/4/2007
DOI 10.1109/L-CA.2007.7

predominant targets and is implemented using an indirect call instruction that is mispredicted 17.1% of the time. When target `Inside_Plane` is taken (72% of all instances) 31 or 32 dynamic instructions are executed until the return instruction, whereas when target `Inside_Quadric` is taken (23% of all instances) 48 or 49 dynamic instructions are executed. Both target paths merge at the return point of the virtual function call. These two targets are interleaved in a difficult-to-predict manner at run time. Dynamically predicating the two target paths when the target is difficult-to-predict could eliminate most target mispredictions at the cost of executing useless instructions on one path. Note that the number of wasted instructions would still be smaller than the number of wasted instructions on a pipeline/window flush resulting from a misprediction (which is at least equal to the instruction window size of the processor in the steady state), assuming the processor parameters shown in Table I.

```
#define Inside(x,y) ((*((y)->Methods->Inside_Meth)) (x,y))

bool pov::Inside_Object(double*, pov::Object_Struct*) {
      // ...
  i = Inside(IPoint,Object)); // indirect call
      return i; // CFM point of the target paths
}

static int Inside_Plane   (VECTOR IPoint, OBJECT *Obj) {
    // 31 or 32 dynamic instructions
}

static int Inside_Quadric (VECTOR IPoint, OBJECT *Obj) {
   // 48 or 49 dynamic instructions
}
```

Fig. 1. A suitable indirect jump example from povray

B. Switch-Case Statement Example

Figure 2 shows a switch-case statement that is responsible for 3% of all indirect jump mispredictions in the SPEC 2006 benchmark `gcc`. This is one of the many switch-case statements used to perform common subexpression elimination on the input program. This indirect jump has three static targets that are respectively taken in 42%, 40%, and 18% of the execution instances. Because this statement is dependent on irregular input data, a BTB-based predictor mispredicts the target of the indirect jump 80% of the time! Since the three target paths merge at the end of the switch statement, this indirect jump is amenable to dynamic predication. In fact dynamically predicating all three target paths when the indirect jump is seen would eliminate all mispredictions at the cost of executing useless instructions. Note, however, that the number of useless instructions is relatively small in each target path (especially for targets 1 and 3) so the amount of wasted work would be small compared to the amount of wasted work on a full pipeline/window flush due to a misprediction.

We found many similar switch-case statements with few dynamically-exercised targets in `gcc`, `perlbench`, `perlbmk`, and `sjeng`. Also, virtual functions similar to the one shown in Figure 1 exist in SPEC 2000/2006 program portions written in C++ and object-oriented-style C (e.g. in `gcc, gap, eon, sjeng, povray`). Building on the insights we develop from code structures suitable for dynamic predication, we next develop an analytical cost-benefit model that can be used to decide when it is profitable to dynamically predicate an indirect jump compared to predicting it.

V. Cost-Benefit model

Figure 3 shows a control flow graph for an indirect jump with 3 targets, where targets A and B are dynamically predicated. Assuming target A is the correct target, instructions on the path from target B to the CFM point are the overhead

```
switch (code) {    // indirect jump
   case PC: case CC0: case CONST: // 10 cases - target 1
      // 2 instructions
      break;
   case REG: // target 2
      // 15-23 dynamic instructions
      break:
   case default: // target 3
      break;
}
// CFM point of the indirect jump
// ...
```

Fig. 2. A suitable indirect jump example from gcc

of predication. On the other hand, if target C is the correct target, all the dynamically predicated instructions are useless and have to be flushed when the indirect jump is resolved.

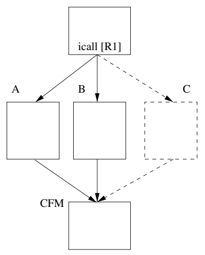

Fig. 3. Control Flow Graph for an indirect jump with 3 targets.

In general, every time a jump with M targets is dynamically predicated with N targets ($N \leq M$), the processor incurs the overhead of fetching and executing the instructions on the incorrect targets. The average fetch overhead of predicating one particular indirect jump in terms of instructions is the sum of the overhead when one of the predicated paths is correct and the overhead when all of the predicated paths are incorrect. When a particular target i is correct - which happens with probability f_i-, the instructions on all the other predicated paths are useless ($Overhead_i$). When none of the predicated targets is correct, we assume the processor continues fetching the maximum number of instructions ($fetch\_width$) every cycle, until the jump is resolved and the pipeline is flushed. Equation 1 gives the overhead of indirect jump predication.

$$dpred\_overhead = \sum_{i=1}^{N}(f_i * Overhead_i) + \sum_{i=N+1}^{M}(f_i * Max\_fetch) \quad (1)$$

$$Overhead_i = \sum_{j=1, j\neq i}^{N} n_j \quad (2)$$

$$Max\_fetch = jump\_resolution\_cycles * fetch\_width \quad (3)$$

$jump\_resolution\_cycles$: Machine-specific branch resolution latency in cycles

We use profiling results for each indirect jump that include the number of targets (M), the probability of each target (f_i), and the average number of dynamic instructions between the indirect jump and the CFM point for each target (n_i).

An indirect jump candidate is selected if the overhead of predication is less than the overhead of indirect jump prediction (Equation 4). The overhead of indirect jump prediction is the expected number of wrong-path instructions fetched.

$$Select\ if\ dpred\_overhead < Max\_fetch * misprediction\_rate \quad (4)$$

$misprediction\_rate$: Jump-specific or average branch misprediction rate

VI. Support for Indirect Jump Predication

A. Compiler and ISA Support

The candidates for indirect jump predication are selected using control-flow analysis and profiling. Control-flow analysis finds the CFM points for each indirect jump. The CFM point for an indirect call is the instruction after the call. The CFM point for a switch statement is the first instruction after the statement ends. Then, we profile the benchmarks to characterize the indirect jumps and obtain the data to use the analytical model from Section V to choose the indirect jumps that satisfy Equation 4. In this paper we restrict the implementation to 2-target predication, and we choose the two most frequently executed targets for each indirect jump.

The indirect jumps selected for dynamic predication are marked in the executable binary with a different opcode, and include the CFM point encoded relative to the indirect jump and the statically selected targets. Even though these special instructions increase the code size, the number of static jumps selected for dynamic predication in the current set of benchmarks is not significant (see Table II).

B. Hardware Support

The hardware required to dynamically predicate indirect jumps is similar to that of the diverge-merge processor (DMP) [10]. The main difference is in the definition of predicates: the predicate for a predicated path is `TRUE` if the actual target of the indirect jump is the starting address of the corresponding path. If more than two targets are predicated, additional contexts are required, including PC (program counter), GHR (global history register), RAS (return address stack), and RAT (register alias table - see [10] for context definition). Additionally, the select-μop generation has to use the predicates for all the paths that merge at the CFM point.

VII. Experimental Methodology

A. Simulation Methodology

We use a Pin-based [12] cycle-accurate x86 simulator to evaluate indirect jump predication. Our baseline processor parameters are shown in Table I. The baseline uses a 4K-entry BTB to predict indirect jumps [13]. The experiments are run using 5 SPEC CPU2000 INT benchmarks, 3 SPEC CPU2006 INT/C benchmarks, 1 SPEC CPU2006 FP/C++ benchmark and 1 other C++ benchmark. We chose those benchmarks in SPEC 2000 INT and 2006 INT/C++ suites that gain at least 5% performance with a perfect indirect jump predictor. Each benchmark is run for 200 million x86 instructions with the reference input set. All binaries are compiled with Intel's production compiler (ICC) using -O3 optimizations.

TABLE I
Baseline processor configuration.

Front End	64KB, 4-way I-cache; 8-wide fetch, decode, rename; 64KB perceptron predictor [8]; min. 30-cycle mispred. penalty
Execution Core	512-entry reorder buffer; 8 all-purpose functional units
Caches	64KB D-L1; 1MB, 32-way, 10-cycle unified L2; 64B lines
Memory	300-cycle min. latency; bus at 4:1 freq. ratio; stream prefetcher
Dyn. Predication Support	2KB enhanced JRS confidence estimator [7] (12-bit history, threshold 14); 32 predicate registers; 1 CFM register [10]

VIII. Experimental Evaluation

A. Indirect Jump Selection

Figure 4 shows the percentage of indirect jump mispredictions that can be avoided by dynamic predication of indirect jumps selected according to the analytical model in Section V, using up to the indicated number of statically-selected targets. 38% of mispredictions can be avoided by predicating only 2 targets. On average, predicating even only 2 targets gives 66% of the misprediction coverage of predicating an unlimited number of targets. Therefore, we use 2-target predication for our preliminary performance evaluations.

B. Preliminary Evaluation

The *Dynamic-pred* bars on Figure 5 show the performance improvement of indirect jump dynamic predication for the 2 most frequently-executed targets, selected based on profiling. Even though this implementation is the simplest realistic implementation of indirect jump predication, it still improves IPC by 6.7% on average, without increasing the hardware complexity beyond that of DMP [10].

Table II shows more details for each benchmark. Note that, even with 2-target predication, 45.1% of the dynamically predicated indirect jumps are potentially useful because they

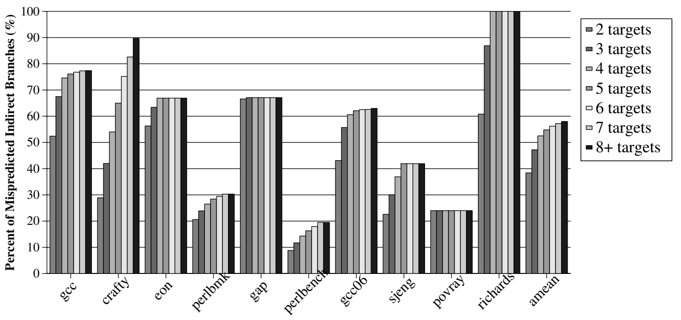

Fig. 4. Ideal coverage of indirect jump mispredictions vs. number of targets.

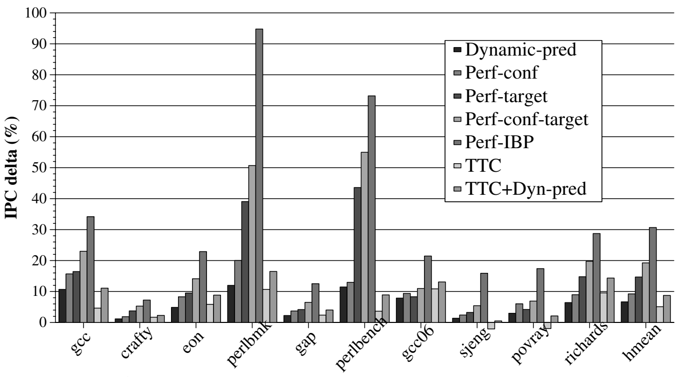

Fig. 5. Performance of indirect jump dynamic predication.

are mispredicted and one of the predicated targets is the correct target. The reduction in the number of pipeline flushes is significant, 18.8% on average, and is the main reason for the observed performance improvement. Even though dynamic predication requires fetching more than one path, the total number of fetched instructions is reduced by 9.5% on average (due to eliminated pipeline flushes), which would result in energy savings in the front end of the processor. Although the number of executed instructions includes the select-μops inserted at the CFM points, the processor executes 2.8% fewer instructions with dynamic predication of indirect jumps.

C. Potential of Improved Dynamic Predication

We performed four idealized experiments to explore the potential of indirect jump dynamic predication. The results on Figure 5 show increasing potential performance benefit.

- *Perf-conf* uses a perfect confidence estimator: each instance of an indirect jump is dynamically predicated if and only if the jump was mispredicted.
- *Perf-target* always includes the correct target in the pair of predicated targets, which improves performance by a significant 14.7%. Therefore, we plan to explore at least two ways of improving dynamic predication: (a) better methods of selecting targets to predicate using dynamic information instead of solely using statically-selected targets; (b) predicating more than two targets to increase the probability of having the correct target (which also increases the overhead of dynamic predication). Figure 4 shows that there is potential to further reduce the number of indirect jump mispredictions by dynamically predicating up to 5 targets. Our future work will focus on hardware and software heuristics to select how many and which targets to dynamically predicate.
- *Perf-conf-target* combines perfect confidence and perfect target, i.e. it perfectly predicates the selected indirect jumps (while preserving the overhead of the extra predicated path). The 4.6% performance improvement provided by this model over *Perf-target* shows that having

TABLE II
CHARACTERISTICS OF THE EVALUATED BENCHMARKS

	gcc	crafty	eon	perlbmk	gap	perlbench	gcc06	sjeng	povray	richards	AVG
baseline IPC	1.01	1.41	1.89	1.08	1.02	0.73	0.58	1.16	1.71	1.04	1.04
indirect jump mispredictions per kilo instructions (MPKI)	6.30	1.10	2.60	11.30	3.30	15.40	6.10	2.90	2.30	6.10	5.74
indirect jumps selected for 2-target dynamic predication	21	2	4	6	19	4	14	1	1	1	-
% of useful dyn. pred. instances, i.e. mispr. IJ predicated w/ correct target	55.4	31.3	44.7	32.8	44.7	30.4	75.9	28.2	40.9	67.0	45.1
% potentially useful dyn. pred. instances, i.e. mispr. IJ predic. w/o corr. target	23.1	60.3	27.2	52.1	29.8	66.7	8.4	32.8	13.1	19.9	33.3
average select-μops per dyn. pred. instance	3.2	5.9	9.3	6.9	5.8	6.2	7.9	8.0	10.0	6.0	6.9
Δ pipeline flushes (%) due to indirect jump dynamic predication	-27.6	-2.6	-28.4	-21.8	-8.3	-21.4	-28.1	-3.6	-15.1	-37.8	-18.8
Δ fetched instructions (%) due to indirect jump dynamic predication	-16.61	-1.27	-9.78	-10.88	-4.23	-13.36	-12.80	-1.75	-5.19	-14.08	-9.53
Δ executed instructions (%) due to indirect jump dynamic predication	-6.47	-0.44	-4.43	1.72	-1.55	-2.85	-15.20	0.97	1.50	2.47	-2.79
Δ energy (%) due to indirect jump dynamic predication	-10.20	-1.05	-6.13	-6.59	-2.82	-8.92	-12.40	-1.03	-2.29	-8.42	-6.63

a better confidence estimator becomes more important when the correct target is selected for predication.

- *Perf-IBP* perfectly predicts all indirect jumps. The 30.7% performance improvement shows the importance of indirect jump mispredictions as a performance limiter in these set of benchmarks. We expect indirect jumps to hinder performance even more significantly in large commercial code bases written using object-oriented programming.

Overall, these ideal experiments suggest that the performance provided by dynamic predication of indirect jumps can further be improved significantly.

The last two bars in Figure 5 show the IPC improvement of (1) a 32-entry tagged target cache (TTC) [5] and (2) both TTC and dynamic predication implemented together. TTC improves IPC by 5.1% and dynamic predication of indirect jumps provides a 3.6% performance benefit over TTC.[3] A specialized indirect jump predictor like TTC requires significant additional hardware. In contrast, if DMP [10] is already implemented for conditional branches, adding dynamic predication of indirect jumps requires very small hardware modifications. In fact, we believe that it is not cost-effective to implement dynamic predication *only for indirect jumps*. On the contrary, dynamic predication hardware is a substrate that can be (and perhaps should be) used for both conditional and indirect jumps. Therefore, our proposal could eliminate or reduce the need to implement a specialized indirect jump predictor.

IX. Related Work

Compiler-based predication [1] has been used to reduce the branch misprediction penalty due to conditional branches by converting control dependencies to data dependencies. Dynamic predication was first proposed to eliminate the misprediction penalty due to simple hammock branches [11] and later extended to handle a large set of complex control-flow graphs [10]. These previous approaches were not applicable to indirect branches. We build on the diverge-merge processor [10] to reduce the misprediction penalty of indirect jumps.

Most current processors use the BTB [13] to predict the target addresses of indirect jumps. A BTB predicts the last taken target of the indirect jump as the current target and is therefore inaccurate at predicting "polymorphic" indirect jumps that frequently switch between different targets. Specialized indirect jump predictors [5], [6] were proposed to predict the target addresses of indirect jumps. Recently, VPC prediction [9] was proposed to use the existing conditional branch prediction hardware to predict indirect jump targets. These previous approaches work well if the target is predictable based on past history. In contrast, dynamic predication of indirect jumps can reduce the performance impact of an indirect jump even if it is hard to predict.

[3]We found confidence estimation accuracy drops significantly when TTC is used. The benefit provided by dynamic predication can be increased by customizing the confidence estimator for TTC. We leave this for future work.

X. Conclusion

We introduced the concept of dynamic predication of indirect jumps. Code examples from existing applications suggest why dynamically predicating hard-to-predict indirect jumps can work. Our preliminary results show that even with untuned heuristics, dynamic predication of indirect jumps can provide significant performance improvements. If dynamic predication hardware is already implemented for conditional jumps, adding indirect jump predication requires very small modifications. We believe the importance of indirect jump predication will increase in the future as more programs will be written in object-oriented styles to improve ease of programming and to reduce software development costs. Our future work will focus on improving the heuristics and cost-benefit models used to select indirect jumps and target addresses to predicate.

XI. Acknowledgments

We thank Aater Suleman, members of the HPS research group, and the reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Intel, AMD and the Advanced Technology Program of the Texas Higher Education Coordinating Board.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *POPL-10*, 1983.
[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber, "Efficient implementation of Java interfaces: Invokeinterface considered harmless," in *OOPSLA*, 2001.
[3] B. Calder, D. Grunwald, and B. Zorn, "Quantifying behavioral differences between C and C++ programs," *Journal of Programming Languages*, vol. 2, no. 4, pp. 323–351, 1995.
[4] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, Dec. 1985.
[5] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *ISCA-24*, 1997.
[6] K. Driesen and U. Hölzle, "Accurate indirect branch prediction," in *ISCA-25*, 1998.
[7] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *MICRO-29*, 1996.
[8] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *HPCA-7*, 2001.
[9] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. S. Cohn, "VPC Prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," in *ISCA-34*, 2007.
[10] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *MICRO-39*, 2006.
[11] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *PACT*, 1998.
[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[13] E. H. Sussenguth, "Instruction control sequence," U.S. Patent 3 559 183, Jan. 26, 1971.
[14] M. Wolczko, *Benchmarking Java with the Richards benchmark*, http://research.sun.com/people/mario/java_benchmarking/richards/richards.html.