Software and Hardware Techniques for Efficient Polymorphic Calls

UNIVERSITY OF CALIFORNIA
Santa Barbara

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy in Computer Science

by Karel Driesen

Committee in charge:

Professor Urs Hölzle, chairperson
Professor Klaus Schauser
Professor Anurag Acharya
Professor Bradley Calder
Professor Trevor Mudge
Professor Yale Patt

Technical Report TRCS99-24
June 1999

Acknowledgments

"A Ph.D. is a prolonged identity crisis"
Theo D'Hondt [37]

I am deeply indebted to a number of people who supported me in the accomplishment of this work. It is impossible to mention all of them within the limits of this page, so I will apologize beforehand to those I forget (you know who you are).

Firstly, my advisor, Urs Hölzle, who not only invited me over to this jewel of the West Coast, Santa Barbara, but also provided me in ample quantities with the three things a graduate student needs: neat toys, time to play with them, and sound advice on how to produce something useful in the process. Whether it was an ergonomic keyboard tray, a summer RA-ship, or a literature study on conditional branch prediction, Urs anticipated my every need before I was even aware of it. He knew when to push me for better performance and when to give me some slack to sort things out. Many times, his constantly positive demeanor kept me going when I felt discouraged. And when I got firmly stuck, he invariably came up with a way out. I feel privileged to have worked so closely with a great mind, who also happens to be a really nice guy. Some debts one cannot hope to pay back.

Secondly, my former advisor, Theo D'Hondt, who supported me in the seven years preceding this work, and whose quote graces this page. In the entirely different circumstances of an underfunded, understaffed and overcrowded department, which he almost single-handedly lifted up to its current state (in my humble opinion the foremost academic center of object-oriented expertise in Belgium), he exhibited vision, boldness, and a infectious love of experimental computer science that influences me to this day. Thanks Theo.

Next, I have to lump together my friends, my family, my tribe. They taught me too many things to sum up, so I will just try to give a sample: Raimondas Lencevicius, for the Dharma and perseverance, Ioana Roxana Stanoi, for style, Lara Hassler, for health, Katherine Ford, for building a home, Denis Khotimsky, for scuba diving, Max Ibel, for hacking, Ruppert Koch, for mechanics, Shirley Geok-lin Lim, for poetry, Katinka Balthazar, for inspiration, Jennifer West, for physics, Catherine Dibble, for vision, Anne-Lise Vandenborre, for understanding, Hiroko Takanashi, for eating out, Smitha Vishveshwara, for beach walking, Sylvia A Liseling, for keeping in touch, Bettina Kemme, for shopping, Kevin Murphy, for running naked, Gerald Aigner, for down-to-earthness, Michel Tilman, Kris De Volder, Jeff Bogda, Andrew Duncan, Ralph Keller, Silvie Dieckman, Holger Kienle, for being fun to work with, Karlo Berket, for poker, Michael Schmitt, for winning at poker, Alicia Watt, for giving hugs, Crina A Vasiliu, for helping out, Jane Carlisle, for saving my life, my parents Jeanne Bleus-Driesen and Frans Driesen, for living, my brothers Willem, Guido and Frank Driesen and my sister, Annie Driesen, for living together, Marlies Bex-Driesen, for caring, Wim Lybaert, for hard-headedness, Wilfried Verachtert, for busyness, Michael Vermeersch, for sarcasm, Luc Hendrix, for fun, Jan Delaet, for rock climbing, Patrick Steyaert, for daring, Stefan Samyn, for love, peace and respect to the max, Stefan Kuypers, for hospitality, Johan Vanderspeeten, for making stuff, Georges Meekers, for guts, Rudi Claes, for bluntness, Peter Muller, for excellence.

And finally, Irene Zafrullah, for love.

Vita

Personalia

Surname:	Driesen
Given names:	Karel Johannes Lodewijk
Place of Birth:	Sint-Truiden, Limburg, Belgium
Date of Birth:	April 15, 1965
Nationality:	Belgian
Address:	Department of Computer Science RSL-lab 2122 Engineering I University of California Santa Barbara CA 93106 USA
Telephone:	(1-805) 893-4394
Fax:	(1-805) 893-8553
Email:	karel@cs.ucsb.edu
URL:	http://www.cs.ucsb.edu/~karel

Education

Master in Computer Science (VUB Brussels, 1993)
Grade: Great Distinction
Thesis: "Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages"

Licentiate Computer Science (VUB Brussels, 1987)
Grade: Great Distinction
Thesis: "Typesystemen in Smalltalk-80" (Type Systems in Smalltalk-80)

Experience

Research assistant (10/1995 - present, Department of Computer Science UCSB Santa-Barbara):
Quantitative analysis of polymorphic calls.
Indirect branch prediction.

Visiting researcher (10/1994 - 9/1995, Department of Computer Science UCSB Santa-Barbara):
Qualitative analysis of message dispatch in object-oriented languages.
Optimization of row displacement dispatch tables.

Researcher (12/1993 - 9/1994, Department of Computer Science VUB Brussels):
Research project: "Design and Use of Programmable Servers in the Modern Office"
Object placing and migration strategies in a distributed OO-system.

Teaching assistant (1/1988 - 11/1993, Department of Computer Science VUB Brussels):
Research:
- finalization of the Minipas programming environment in Lightspeed Pascal.
- analysis of the influence of network topology on generalization performance of multi-layered perceptrons (artificial neural networks).

- comparative analysis of message dispatch techniques.
- implementation of row displacement compression for dispatch tables.

Teaching:
- Structure of Computer Programs I
- Algorithms and Data Structures I
- Algorithms and Data Structures II
- Programming Project
- Parallel Systems
- Analysis and Design
- Assistant Thesis Advisor

Elected representative of teaching assistants (1991, Board of directors VUB Brussels)

Elected representative of teaching assistants (1990-1991, Faculty of Sciences board)

Research assistant (10/1987 - 12/1987, Department of Mathematics VUB Brussels):
Finalization of the two-year demographic project:
"Prognosis of the number of Belgian students until 1995-96"

Programmer (7/1985 - 8/1985, 7/1986 - 9/1986, U-Soft software company based in Hasselt):
Machine parts stock management software in UCSD Pascal.
Documentation of BancContact server software in COBOL.
EBNF to syntax diagrams translation in UCSD Pascal.

Publications

Multi-stage Cascaded Prediction.
with Urs Hölzle
To appear in the EuroPar'99 Conference Proceedings

The Cascaded Predictor: Economic and Adaptive Branch Target Prediction.
with Urs Hölzle
In Micro'98 Conference Proceedings, pp. 249-258, Dallas, Texas, December 1998

Accurate Indirect Branch Prediction.
with Urs Hölzle
In ISCA '98 Conference Proceedings, pp. 167-178, Barcelona, July 1998

The Direct Cost of Virtual Function Calls in C++.
with Urs Hölzle
In OOPSLA '96 Conference Proceedings, p. 306-323, San Jose, California,1996. Published as SIGPLAN Notices 31(10), October 1996.

Minimizing Row Displacement Dispatch Tables.
with Urs Hölzle
OOPSLA'95 Conference Proceedings: p. 141-155, Austin, Texas, October 1995, Published as SIGPLAN Notices 30(10), October 1995.

Message Dispatch on Modern Computer Architectures.
with Urs Hölzle and Jan Vitek
ECOOP '95 Conference Proceedings, p. 253-282, Århus, Denmark, August 1995. Published as Springer Lecture Notes in Computer Science Vol. 952, Springer-Verlag, Berlin Heidelberg 1995

Compressing Sparse Tables using a Genetic Algorithm.
Proceedings of the GRONICS'94 Student Conference, Groningen, February 1994.

Selector Table Indexing and Sparse Arrays.
OOPSLA '93 Conference Proceedings, p. 259-270, Washington, D.C., 1993. Published as SIGPLAN Notices 28(10), September 1993.

Awards

Third place in the 1996 Student Writing Contest of the Society for Technical Communication, Santa-Barbara Chapter, with the white paper "Multiple Dispatch Techniques: a survey.".

Graduate Division's Spring 1998 Dissertation Fellowship, University of California, Santa-Barbara

Inventions

"Cascaded Prediction", Karel Driesen and Hölzle, Patent Pending #09/268483

Reports

Improving Indirect Branch Prediction With Source- and Arity-based Classification and Cascaded Prediction.
with Urs Hölzle
Technical Report TRCS98-07, Department of Computer Science, University of California Santa-Barbara, March 15, 1998

Limits of Indirect Branch Prediction.
with Urs Hölzle
Technical Report TRCS97-10, Department of Computer Science, University of California Santa-Barbara, June 25, 1997

Multiple Dispatch Techniques: a survey.
White paper, Santa-Barbara, 1996.

Method Lookup Strategies in Dynamically-Typed Object-Oriented Programming Languages.
Master's Thesis, Free Brussels University (VUB) 1993.

Prognosis of the number of Belgian students until 1995-96
with Franz Bingen and Carlos Siau
Internal publication Free Brussels University (VUB) 1988

Typesystemen in Smalltalk-80.
(Type Systems in Smalltalk-80)
Licentiate Thesis, Free Brussels University (VUB) 1987

Abstract

Object-oriented code looks different from procedural code. The main difference is the increased frequency of *polymorphic calls*. A polymorphic call looks like a procedural call, but where a procedural call has only one possible target subroutine, a polymorphic call can result in the execution of one of several different subroutines. The choice is made at run time, and depends on the type of the receiving object (the first argument). Polymorphic calls enable the construction of clean, modular code design. They allow the programmer to invoke operations on an object without knowing its exact type in advance.

This flexibility incurs an overhead: in general, polymorphic calls must be resolved at run time. The overhead of this run time polymorphic call resolution can lead a programmer to sacrifice clarity of design for more efficient code, by replacing instances of polymorphic calls by several single-target procedural calls, removing run time polymorphism. This practice typically leads to a more rigid program structure and code duplication, increasing the short term effort required to build a functional prototype, and the long term effort of maintaining and adapting a program to changing needs.

We study techniques to minimize the run-time cost of polymorphic calls. In the software domain, we minimize the memory overhead of table based implementations (message dispatch tables), which are most efficient in terms of number of instructions executed. In the hardware domain, we reduce the cycle cost of these instructions through indirect branch prediction. For reasonable transistor budgets, hit rates of more than 95% can be achieved. As a result, only one out of twenty polymorphic calls incurs significant cost at run time.

Design of clear, maintainable and reusable code, as enabled by object-oriented technology, can thereby become less restrained by efficiency considerations. Only in very time-critical program segments should the programmer avoid the use of polymorphism. In other words, object-oriented code can become the norm rather than the exception. From our own experience in building software architectures, we consider this a Good Thing.

Table of Contents

List of figures

List of tables

1 Introduction

"All understanding begins with our not accepting the world as it appears",
Alan C. Kay [115]

The object-oriented programming style, and the languages that enable it, have acquired an aura of respectability. Almost everyone agrees it is a Good Thing. From the adoption of object-oriented software architectures in high-performance computing, with the expected support of polymorphic calls in Fortran2000 [1], to the emergence of a binary compatible component market, enabled by the Java Virtual Machine [61], objects are set to become a pervasive software organization paradigm.We therefore expect processors to have to deal with more object oriented code in the near future.

Object-oriented code looks different from procedural code. The main difference is the increased frequency of *polymorphic calls*. A polymorphic call looks like a procedural call, but where a procedural call has only one possible target subroutine, a polymorphic call can result in the execution of one of several different subroutines. The choice is made at run time, and depends on the type of the receiving object (the first argument). Polymorphic calls enable the construction of clean, modular code design. They allow the programmer to invoke operations on an object without knowing its exact type in advance.

This flexibility incurs an overhead: since the object type is typically unknown at compile time, polymorphic calls must be resolved at run time. This requires extra instructions, compared to single-target, or early-bound calls, and therefore implies a cost to the use of the object-oriented programming style. This overhead can lead a programmer to sacrifice clarity of design for more efficient code, by replacing instances of polymorphic calls by several single-target procedural calls, removing run time polymorphism. This typically leads to a more rigid program structure and code duplication, increasing the short term effort required to build a functional prototype, and the long term effort of maintaining and adapting a program to changing needs.

We study techniques to minimize the run-time cost of polymorphic calls in order to reduce the overhead of the object-oriented programming style. In the software domain, we minimize the memory overhead of table based implementations, which are most efficient in terms of number of instructions executed. In the hardware domain, we reduce the cycle cost of these instructions through indirect branch prediction. For reasonable transistor budgets, hit rates of more than 95% can be achieved. As a result, only one out of twenty polymorphic calls incurs some cost at run time.

Design of clear, maintainable and reusable code, as enabled by object-oriented technology, can thereby become less restrained by efficiency considerations. Only in very time-critical program segments should the programmer avoid the use of polymorphism. In other words, object-oriented code becomes the norm.

1.1 Problem statement

The goal of this work is to bring the run-time cost of lately-bound, multiple-target polymorphic calls as close as possible to the cost of early-bound, single-target procedural calls. In other words, we minimize the cost of polymorphic call resolution, i.e. the cycles spent on the mapping of a polymorphic call to a single target at run time. The main focus will lie on single argument polymorphism in statically and dynamically typed object-oriented languages with single and multiple inheritance.

We also address related problems, such as the run-time memory cost and responsiveness of polymorphic call resolution techniques. The hardware techniques we present optimize polymorphic calls in object-oriented languages as well as switch statements, dynamically linked calls, and hand-built polymorphism in procedural languages.

1.2 Background and motivation

Polymorphic calls are programming language constructs that enhance a programmer's ability to organize behavior. They allow bundling and encapsulation of groups of operations that work on a specific type of data, also called an *object*. For example, the UCSB web page, shown in

Figure 1. Fragment of the UCSB webpage *www.ucsb.edu (Author:Joseph Boisse)*

Figure 1, contains three different data types: a picture, a text fragment, and four buttons. These screen objects respond to common operations. Every object can *draw* itself on screen at a certain location. Although the intended effect of the *draw* operation (its *functionality*) is always same, the actions required to bring about that effect (its *implementation*) can differ between object types. For example, drawing a picture on screen requires that a given two-dimensional set of pixels is placed in the screen buffer. Drawing text requires that a string is translated into pixels by looking up each character in a font map and then writing those pixels to the screen buffer. Operations that require the execution of different subroutines depending on the type of object they are applied to, are called *polymorphic*.

Since each object-oriented language provides its own terminology for polymorphic operations, Table 1 shows the terms used in popular languages for easy translation. We will use the general terms provided as headings, unless we discuss language-specific techniques.

Language	Polymorphic call	Implementation
C++, Java	Virtual function call	Function definition
Smalltalk, SELF	Message send	Method
CLOS, Dylan	Generic function invocation	Method

Table 1. Language-specific terms for polymorphic calls

Polymorphic calls make life easier for the user of an object library, who only has to know what operations mean in terms of functionality. It becomes easy to manipulate collections of objects that are different but share functionality such as the *draw* operation.To draw a page consisting of many screen objects, a programmer merely has to call *draw* on every object. It is not necessary to know the implementation details of each object's type (see Figure 2).The run-time system intercepts the polymorphic call and redirects it to the appropriate implementation.

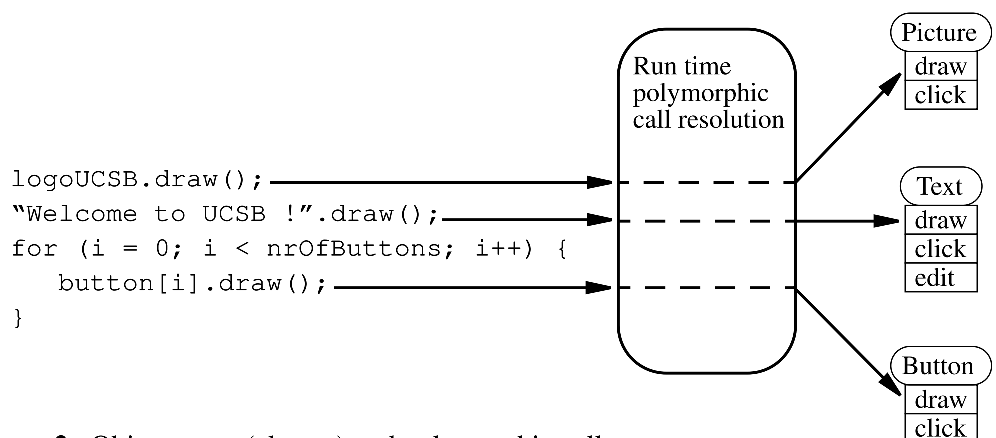


Figure 2. Object types (classes) and polymorphic calls

1.2.1 Inheritance

1.2.1.1 Subtyping

For the purpose of showing objects on screen, it doesn't matter what type of object is being manipulated; they are *substitutable*. Substitutability is a useful concept when you are handling large numbers of diverse objects, and therefore object-oriented languages provide a language mechanism to structure its use. For example, to indicate that a *Button* and *Text* are just like a *Picture*, we call *Button* and *Text subtypes* of *Picture*, as indicated by the hierarchy graph in Figure 3.

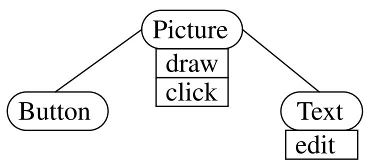


Figure 3. Subtype hierarchy

The user can consult the diagram in Figure 3 to check which objects respond to which operations. A given object type responds to all operations defined for it and all its super types. In this manner subtyping enables incremental interface definition. The similarity of subtypes in terms of functionality is thus structured by the subtype hierarchy.

1.2.1.2 Subclassing

This similarity often carries over into the implementation[1], since similar operations on similar objects are often implemented the same way. For instance, drawing a *Button* is just like drawing a *Picture* (placing the appropriate bitmap on screen). Object oriented languages provide support for code sharing between similar object types in the form of implementation inheritance or *subclassing*. When an object type or *class* is a subclass of another class, it not only responds to the same operations (it is a subtype), but it also *uses the same program code* to execute them. A subclass can change this default behavior by *overriding* (a new implementation is defined in the subclass). For example, in Figure 4, a *Button* shares code with *Picture* for drawing purposes, but overrides (indicated in ***italics***) the implementation provided for *click* with code that changes the

[1] Subclassing and subtyping run along similar enough lines that many object-oriented languages pretend the two are the same. Smalltalk only provides subclassing. Deviations for the sake of subtyping must be manually coded on top of the subclass hierarchy, by giving some operations the default implementation ShouldNotImplementError, to "remove" it from an interface (in the Collection class hierarchy, for example, where Set inherits from Dictionary). In C++, multiple inheritance combined with the use of constrained visibility, using the keyword protected, allows a user to inherit separately for implementation and interface. However, even simple cases of multiple inheritance in C++ inflate the run-time size of objects. In Java, single inheritance subclassing provides implementation inheritance as in Smalltalk. Interfaces enable the construction of subtype hierarchies, in which multiple inheritance is allowed. The keyword *implements* provides the connection between the two, declaring that a particular class is a subtype of (multiple) interfaces.

user's perspective to a different web page. *Text*, on the other hand, inherits *click* but provides its own drawing implementation and adds the *edit* operation to its interface and implementation. Subclassing therefore enables incremental implementation. This is useful for the *builder* of an object library. Instead of implementing the complete interface of a class, only the difference between it and an appropriately chosen super class needs to be coded from scratch.

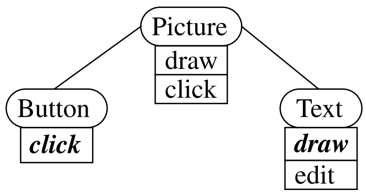


Figure 4. Subclass hierarchy

Inheritance requires polymorphic calls. Since objects are substitutable, the programmer is often unaware of the actual type of the objects that are manipulated. He/she relies on the system to find and call the correct implementation for a given operation. This enhances maintainability, since objects may be designed or constructed by a third party long after the code has been delivered. Only substitutability is required for the existing code to work correctly.

It is the responsibility of the language system builder to make polymorphic calls efficient. A compiler may be able to replace some polymorphic calls by early-bound single-target calls, if it can be proven that the call can result in only one target subroutine. However, real polymorphism must be resolved at run time. It is our aim to reduce the run time cost of polymorphic call resolution as much as possible.

1.3 Contributions

We studied techniques, first in software, then in hardware, to reduce the cost, both in time and space, of run-time polymorphic call resolution. In doing so, we made a number of contributions to the field.

In software, we made the following contributions:

- *Minimization of the memory overhead of dispatch tables.* Table-based dispatch, a fast polymorphic call resolution technique, was formerly restricted to statically typed programming languages. We increased its application domain by reducing its memory overhead from 95% to 36% in previous work [40]. Similar efforts by Andre and Royer [9], resulted in 43% overhead and took hours to compute. As part of the dissertation work we reduced the overhead to less than 1%, optimized the table construction algorithm to run in seconds instead of hours for large class libraries (see Section 4), and designed a variant to handle multiple inheritance hierarchies. This makes table-based dispatch practical for two additional classes of programming languages: dynamically typed languages and languages that allow multiple inheritance.

- *A qualitative and quantitative analysis of dispatch techniques on modern processor architectures.* We analyzed the assembly instruction call sequences of most polymorphic call resolution techniques (see Section 3 and Section 5). This study shows that delays associated with pipelining and superscalar execution can deteriorate the performance of table-based techniques compared to techniques based on inline caching. Previously it was assumed that table-based dispatch always performed better than caching methods. We then measured the performance of virtual function tables, the most widely used table-based technique, on a variety of processor architectures (see Section 6). This experiment shows the impact of load latency, superscalar execution, and branch penalty on virtual function call performance. The results indicate that co-scheduling of surrounding instructions is insufficient to hide branch penalties.

The main reason for the better performance of inline caches compared to table-based techniques is that they implement a form of software based branch prediction, similar to a Branch Target Buffer (BTB) of infinite size and unlimited associativity. Such ideal BTBs reach 75% prediction accuracy on the OOCSB98 benchmark suite. In other words, one out of four polymorphic calls takes a branch misprediction penalty.

Since the cost of polymorphic calls in the most efficient software scheme is dominated by branch misprediction penalties, further optimization must focus on more accurate indirect branch prediction. This benefits all indirect branches, not only those generated by object-oriented message dispatch, and therefore its relevance is higher than that of a polymorphic call resolution technique in software. In particular, switch statements, dynamically linked calls, and calls through function pointers, often used to hand-build polymorphism in procedural languages, become more efficient. We therefore proceeded to study indirect branch prediction in hardware.

In the hardware domain, we made the following contributions:

- *An extensive study of two-level prediction for indirect branches.*
  Two-level predictors, which use the targets of the n most recently observed branches (history path length n) to predict the next branch, were first tested on indirect branches by Chang, Hao and Patt [25]. They measured a linear relationship between branch misprediction rate (the reciprocal of prediction accuracy) and cycle cost. We measured prediction accuracy with unlimited hardware resources, varying the amount of sharing of history buffers and history tables [46]. After determining the best parameters on the OOCSB98 suite, a large benchmark set of procedural and object-oriented programs, we studied the effect of hardware constraints such as limited table size and limited associativity (see Section 8). With appropriate history pattern encoding, the resulting two-level predictor achieves a prediction accuracy of 90% for a history table with 1K entries.

Next we combined two-level predictors of different path length into a hybrid predictor to increase the prediction accuracy further. Hybrid predictors were first proposed by McFarling [98] for conditional branch prediction. To our knowledge, this study is the first to evaluate hybrid prediction for indirect branches. We studied three classes of hybrid predictors:

- *Classifying predictors*, first proposed for conditional branches by Chang, Hao and Patt [23], assign a class to each branch according to compile-time or profile-based criteria. We dedicate a separate two-level predictor to each class, with a separately tuned path length (see Section 9.2). The best of the tested predictors achieves 91% prediction accuracy for 1K table entries, but requires a change in the instruction set architecture.

- *Dual-path hybrid predictors*, use two two-level predictors of different path length, update both for each branch and use the prediction with the highest 2-bit confidence counter (see Section 9.3). This dynamically updated counter keeps track of the number of successes among the last 3 predictions of a table entry. After tuning, the best predictor achieves 91% prediction accuracy for 1K total table entries.

- *Cascaded predictors*, a new prediction architecture (see Section 9.4), use several stages of two-level predictors of different path length, and give precedence to the longest path length prediction available. In addition, pattern filtering reduces the number of table entries required to reach a particular prediction accuracy. When updating, a cascaded predictor prevents insertion of a pattern into the longer path length component if the shorter path length prediction is correct. The best combination with a BTB as first stage resulted in 92% prediction accuracy for 1K total table entries. A cascaded predictor reduces the cost of a two-level predictor by a factor four, for similar prediction accuracy. At 6K entries, a 3-stage cascaded predictor with tuned path lengths reaches 95% prediction accuracy, higher than the 94% accuracy achieved by a hypothetical two-level predictor with an unlimited, fully associative predictor table.

Combined, the software contributions minimize the cost of a polymorphic call in *number of instructions*, while the hardware advances minimize the cost of these instructions in *number of processor cycles.* At affordable off-chip memory costs and on-chip transistor budgets, well-tuned software and hardware support can bring the run-time overhead of polymorphic call resolution down to an insignificant fraction of total run time, as measured on various industry-size benchmarks.

1.4 Overview

This dissertation is organized in sections that correspond to the different fields the work touches upon. Section 2 describes variations of the polymorphic call construct and states the problem we address in this work. Section 3 presents the major software techniques for efficient polymorphic call resolution and compares their memory overhead and program environment constraints. Section 4 presents and measures a new algorithm for memory overhead reduction of dispatch tables.

The next two sections evaluate the run-time performance of software techniques by qualitative and quantitative analysis of the software-hardware interface. In Section 5 we establishes bounds on the performance of table-based techniques on different processor generations and estimate average cycle overhead of dynamic techniques. In Section 6 we measure the overhead of virtual function table dispatch by cycle-level simulation, and study the effect on call resolution overhead of processor characteristics like instruction issue, load latency, and indirect branch prediction.

The remaining sections deal with hardware techniques. In Section 7 we discuss the hardware context, present benchmark programs and our method of investigation. In Section 8 we measure and reduce history pattern interference misses, capacity and conflict misses of basic path-based predictors. Section 9 studies three classes of path-based hybrid predictors, and presents the cascaded predictor, a new architecture which reduces the cost of indirect branch predictor tables by a factor four. Section 10 discusses related work not mentioned in the survey chapters. Section 12 concludes. Section 11 discusses open problems and future work.

2 Polymorphic calls

"Polymorphism: In biology, the coexistence of two or more genetically distinct forms of an organism within the same interbreeding population, where the frequency of the rarest type is not maintained by mutation alone. Human eye colour is an example of a readily observable polymorphism, but there are many invisible polymorphisms detectable only by special techniques, such as DNA analysis",

The Cambridge Encyclopedia[15]

In this section we discuss the polymorphic call construct as it appears in different programming languages.

2.1 Basic construct

Polymorphism is a powerful programming construct because it allows a programmer to invoke an operation on a piece of data by specifying the intended effect, without having to worry about the implementation details on different kinds of datatypes. Using the webpage example introduced in Section 1, non-polymorphic code to draw the page could look as follows:

```
logoUCSB.drawPicture();
"Welcome to UCSB !".drawText();
for (i = 0; i < nrOfButtons; i++) {
   button[i].drawButton();
}
```

The programmer has to specify the right drawing operation for each screen object. Polymorphic code looks like this:

```
logoUCSB.draw();
"Welcome to UCSB !".draw();
for (i = 0; i < nrOfButtons; i++) {
   button[i].draw();
}
```

or even simpler:

```
for (i = 0; i < nrOfScreenObjects; i++) {
   screenObject[i].draw();
}
```

In polymorphic code, the programmer is free of the concern to match the right drawing operation to each screen object. The run-time system distinguishes different screen objects and invokes the right implementation of the *draw* operation. This allows rapid prototyping, and code that is easier to maintain, since the introduction of a new screen object, for example, a framed picture, does not force rewriting of all *drawPicture* invocations. A new implementation of *draw* must be provided, and from then on, all program code that manipulates pictures through the public interface of the *Picture* class is equally capable of dealing with framed pictures.

2.2 Polymorphic calls in procedural languages

Procedural languages give no support for polymorphism. However, since it is a useful abstraction tool, hand-crafted polymorphic calls are often found in procedural programs. The key mechanism is to remove the type-specific procedure invocations from the call point, replacing them by a single procedure invocation, and resolving the polymorphism in the called procedure. For example, a polymorphic *draw* procedure could be implemented as follows:

```
draw (screenObject o) {
   switch (o.type) {
      case picture: drawPicture(o); break;
      case text: drawText(o); break;
      case button: drawButton(o); break;
      default: error("draw undefined for object");
}
```

Though this construction removes implementation concerns from the calling point, the programmer must hand-craft polymorphism resolution procedures for every polymorphic operation. Implementation of call resolution on the run-time system level removes this repetitive burden from the programmer. It also allows more radical optimization because run-time implementation requires only a one time implementation effort.

Hand-crafted polymorphism has its flaws also from a maintenance perspective. Relationships between types are encoded ad-hoc in the polymorphic call procedures. When a new data type is introduced, all polymorphic procedures that handle the type must be augmented, and this code change is dispersed over many different places in the program. In object-oriented languages, a more systematic approach allows the programmer to define all the new code in one place, and take advantage of implementation inheritance to minimize the extra implementation effort.

2.3 Object-oriented message dispatch

Object-oriented languages allow the definition of a datatype and its associated operations in one place, typically called a class or a prototype object. In our example, the programmer provides each screen object with a *draw* implementation, and the system will ensure that the right implementation is called for each type. In object-oriented languages, like Smalltalk, polymorphic calls are known as *message sends* and polymorphic call resolution is called *message dispatch*. The different class-specific implementations of a message are called *methods*. Implementation inheritance allows default methods to be defined for sets of classes. For instance, the *draw* method for a *button* is identical to that of a picture, while the *click* method is redefined, or *overridden* (see Figure 5).

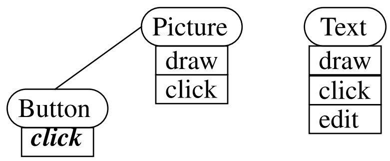


Figure 5. Single inheritance with multiple root classes

The semantics of message dispatch are best explained by the mechanics of the simplest polymorphic call technique: Dispatch Table Search (DTS). In this technique, each class stores the definitions of all the messages it implements (shown in Figure 5), and has a reference to its super class for the default behavior. For example, when the message *draw* is sent to a *Button*, its dispatch table has no implementation for *draw*. DTS therefore obtains the super class reference from *Button*, and continues the search in class *Picture*, where the method for *draw* is found and invoked. Though DTS implements the semantics of polymorphic calls in object oriented languages perfectly, its efficiency leaves much to be desired. We will discuss more efficient dispatch mechanisms in Section 3.

2.3.1 Static vs. dynamic typing

In a statically typed language, each variable in the program is declared with a type. The compiler makes sure that the only operations invoked on a certain variable are those defined for its declared type. For the example in Figure 5, the following code would be legal:

```
Picture pict;
...
pict = new Button;
pict.draw();
```

Since the type declaration for variable *pict* ensures that the object it refers to is an instance of *Picture* or any of its subclasses, the programming system is *guaranteed* to find a method for the draw message. At compile time the possibility of a "Message Not Understood" - error is removed. In a dynamically typed system, variables are untyped (but objects still have a type). The compiler will reject the following code fragment in a statically type language, but will allow it in a dynamically typed language:

```
pict = new Picture;
pict.edit();
```

However, at run time, the programming system would not find an method for *edit*, since *Picture* objects lack this operation. A run time error would result.

Statically typed languages make it easier for the programming system to implement message dispatch efficiently, since the number of different messages that can be sent to a variable is limited and a method always exists. In a dynamically typed language, any message can be sent to any object, so the number of possibilities is much larger, and the system must be able to handle the case where no method can be found. Dynamically typed languages offer greater flexibility, by not requiring that all possible receivers of a given message share a common ancestor that defines the message. For example, the following code is incorrect in a statically typed language, but both allowed and error free in a dynamically typed language (assuming the inheritance hierarchy in Figure 5):

```
if (test) {
   obj = new Text;
} else {
   obj = new Button;
}
obj.draw();
```

Although both *Text* and *Button* understand the message *draw*, static typing would not allow the assignment of instances of these classes to the same variable, because they do not share a common superclass in which *draw* is defined[1].

2.3.2 Single vs. multiple inheritance

Some object-oriented languages allow a class to inherit from more than one super class. This is called multiple inheritance. Multiple inheritance complicates message dispatch, since it becomes possible to reach two or more method for the same message, through different paths from a class to its ancestors. For example, suppose we extend the hierarchy in Figure 5 by adding classed *FramedPicture* and *FramedText*. The latter inherits from both *Text* and *Framed-Picture*, as depicted in Figure 6. Sending *draw* to a *FramedText* object now begs the question: which method is invoked?

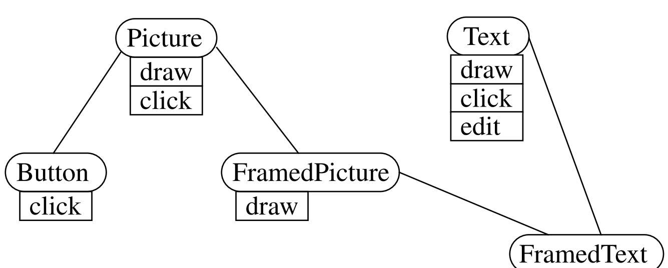


Figure 6. Multiple inheritance

[1] In Java, one can get around this restriction by declaring an interface with the *draw* message and specifying that *Button* and *Text* implement the interface.

Different programming languages give different answers: in CLOS [107], the programmer lists the super classes of a class in a certain order, and this tells the programming system which class takes priority. In C++, this kind of ambiguity causes a compile-time error. The programmer can specify which path takes precedence, for example by calling *Text::draw*. This practice partially breaks down the abstraction mechanism provided by polymorphism, since the programmer now has to limit the choice of method at the calling point. From an implementation perspective the semantics do not matter much, as long as the *draw* call resolves to a unique method.

Multiple inheritance complicates efficient polymorphic call resolution, since a class no longer extends an existing set of operations with a few extra messages, but merges two or more sets. Compile-time construction of compact dispatch tables becomes harder in that case, as discussed in Section 4.

2.3.3 Single vs. multiple dispatch

In most object-oriented languages, the type of one privileged argument, called the *receiver*, determines the method to invoke for a polymorphic call. Languages like CECIL [18] and CLOS [83] allow the types of other arguments to further refine the choice of method. This can be useful, for example to allow the draw operation to display graphical objects on different media, like a screen or a postscript printer. Both the type of the object to be drawn, and the medium on which it is drawn, determine the procedure to be invoked:

```
for (i = 0; i < nrOfObjects; i++) {
   object[i].draw(medium);
}
```

The standard solution to multiple dispatch in languages that only provide single argument dispatch is a technique called *double dispatch* [75]. The programmer writes each draw operation in a graphical object class as follows (for example in the *Picture* and *Text* classes, with media *Screen* and *PostscriptPrinter,* and where *THIS* is the object for which the draw operation was invoked):

```
class Picture
   draw (medium)
      medium.drawPicture(THIS);

class Text
   draw (medium)
      medium.drawText(THIS);
```

```
class Screen
   drawPicture (picture)
      // code for drawing a picture on a screen
   drawText (text)
      // code for drawing text on a screen

class PostscriptPrinter
   drawPicture (picture)
      // code for drawing a picture on a postscript printer
   drawText (text)
      // code for drawing text on a postscript printer
```

Dispatch over two arguments is emulated by two successive single-argument dispatches. This technique has several disadvantages. First, the programmer is responsible for the maintenance of double dispatch code, similar to the hand-crafted polymorphic call resolution procedures that implement single argument dispatch in procedural languages. Second, double dispatch is slower than multiple dispatch. Multiple dispatch executes parameter passing code only once, and can employ more advanced call sequence optimization. In practice, programmers do not often employ multiple dispatch, even in languages that support it. Single-argument dispatch remains the common case (see [44]). We will not explore multiple dispatch techniques in depth (however, see section 10.1.3 for an analysis).

2.3.4 Predicate dispatch

Ernst, Kaplan and Chambers present a unified model of dispatch [55], based on predicate expressions that serve as guards to multiple implementations of the same generic function. Since predicate expressions can be arbitrary side-effect free expressions, this model captures all variants defined above, as well as ML-style pattern matching, predicate classes, and classifiers. From a language design perspective, such a powerful dispatch mechanism is ideal, because it allows the programmer to use the style best fit to model a specific problem, or even to construct a personal style. From an implementation perspective, predicate dispatch presents a big challenge. We believe that a combination of compile-time and run-time techniques will be necessary to reduce its cost and make it attractive to performance-conscious practitioners. Single-argument message dispatch is likely to remain the most frequent operation in most programs, and a general dispatch mechanism for predicate dispatch must therefore incorporate an efficient technique for this common case. Chambers and Chen [20] present efficient implementation of multiple and predicate dispatch, which combines efficient techniques for single dispatching, multiple dispatching, and predicate dispatching. All these cases are implemented as a series of single dispatches where each single dispatch is implemented with the technique appropriate for the type of call. For example, if the number of targets is large, a table based implementation is chosen instead of a linear search.

3 Software techniques for efficient polymorphic calls

"Swiss Army Knife. A tool that makes the common task easy, while providing all the necessary tools for the most general task."
Jennifer West [132]

In this section, we describe the major techniques in current use for the optimization of polymorphic calls. We present the techniques and discuss run-time aspects like estimated memory cost and responsiveness. Detailed estimates of run-time call cost are delayed until Section 5, where we bring hardware into the picture.

3.1 Basic message dispatch in object-oriented languages

In object-oriented languages, polymorphic call resolution is called *message dispatch*. Message dispatch is a function that takes the message name (*selector*) and the class of its first argument (*receiver*), and matches this pair to the correct implementation, also called *method*. If method lookup speed was unimportant, dispatch could be performed by searching class-specific dispatch tables. When an object receives a message, the object's class is searched for the corresponding method, and if no method is found the lookup proceeds in the super class(es). Since it searches dispatch tables for methods, this technique is called Dispatch Table Search (DTS). The right-hand side of Figure 7 shows the dispatch tables of the class hierarchy on the left. Each entry in a dispatch table contains the method name and a number representing its address. As in all other figures, capital letters (*A*, *B*, *C*) denote classes and lowercase letters denote methods.

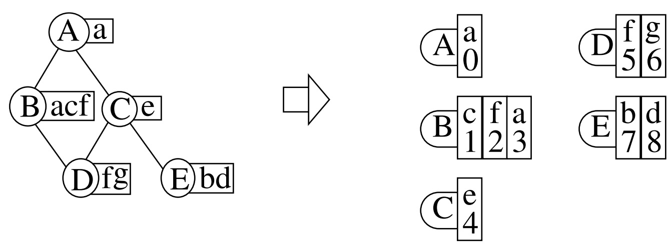


Figure 7. Class hierarchy with corresponding dispatch tables

Since the memory requirements of DTS are minimal (i.e., proportional to the number of methods in the system), DTS is often used as a backup strategy which is invoked when faster methods fail. Typically, DTS implementations employ hashing to speed up the table search.

All of the techniques discussed in the remainder of this chapter improve upon the speed of DTS by precomputing or caching lookup results. The dispatch techniques studied here fall into two categories. *Static techniques* precompute all data structures at compile or link time and do not change those data structures at run-time. Thus, static techniques only use information that can be statically derived from the program's source text. *Dynamic techniques* may precompute some information at compile or link time, but they dynamically update data structures at run-time, i.e., during program execution. Thus, dynamic techniques can exploit run-time information as well as static information. Table 2 lists the techniques we studied.

	Acronym	Full Name	Section
static techniques	DTS	Dispatch Table Search (used for illustration only)	3.1
	STI	Selector Table Indexing (used for illustration only)	3.3.1
	VTBL	Virtual Function Tables, single inheritance	3.3.2
	VTBL-MI	Virtual Function Tables, full version	3.3.2
	SC	Selector Coloring	3.3.3
	RD	Row Displacement	3.3.4
	CT	Compact Tables	3.3.5
dynam. techn.	LC	Lookup Caching	3.2.1
	IC	Inline Caching	3.2.2
	PIC	Polymorphic Inline Caching	3.2.3

Table 2. Overview of dispatch methods

3.1.1 Influence of dynamic typing

In dynamically-typed languages, a program may try to invoke an operation on some object for which the operation is undefined ("message not understood" error). Therefore, each message dispatch usually needs to include some form of run-time check to guarantee that such errors are properly caught and reported to the user. Most techniques that support static typing can be extended to handle dynamic typing as well. Our study shows the additional dispatch cost of dynamic typing for all dispatch mechanisms that can support it.

3.1.2 Influence of multiple inheritance

A system using multiple inheritance (MI) introduces an additional difficulty if compiled code uses hard-coded offsets when addressing instance variables. For example, assume that class *C* inherits directly from classes *A* and *B* (Figure 8). In order to reuse compiled code of class *A*, instances of *C* would have to start with the instance variables of *A* (i.e., *A*'s memory layout must be a prefix of *C*'s layout). But the compiled code in class *B* requires a conflicting memory layout (*B*'s instance variables must come first), and so it seems that compiled code cannot be reused if it directly addresses instance variables of an object.

Hard-coded offsets can be retained, in a statically typed setting, if the receiver object's address is adjusted just before a *B* method is executed, so that it points to the *B* sub-object within *C* [87, 53] (the *current* type is *B*). Within the called method, the *current* type's instance variable offsets and the position of its virtual function table are compile time constants. If the same method is called from different sub-objects, different pointer adjustments are required. Therefore each

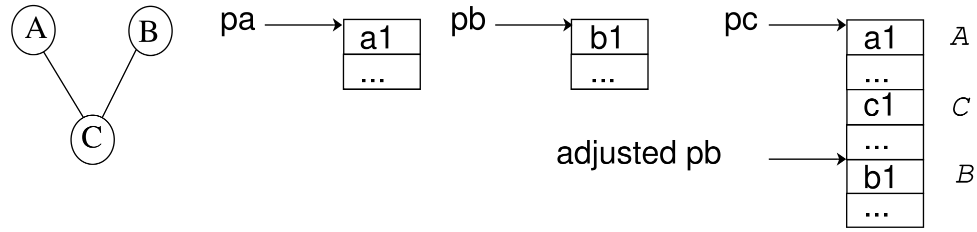


Figure 8. Memory layout of objects in the case of multiple inheritance.
The offset of instance variable *b1* is different in instances of *B* versus instances of *C*.

sub-object points to its own table, in which an extra entry for each function stores the size of the adjustment. However, in a dynamically-typed setting, the structure of the object is unknown at compile time. Therefore we assume that techniques capable of dealing with dynamic typing use the following approach to access instance variables in the multiple inheritance case:

- always use accessor messages, except to access instance variables from self[1].
- use hard-coded offsets for self instance variables, and maintain a different compiled version for every method of a class that accesses them, whenever it is combined with a different co-parent.

Under this constraint, all techniques studied here (except VTBL for older processors) show near-identical performance for single and multiple inheritance. We discuss both a single- and a multiple inheritance version for VTBL. The space overhead caused by multiple inheritance is treated, for all techniques, in Section 3.4.2.

Each description of a dispatch technique includes pseudo-code illustrating the run-time dispatch. Since our analysis separates out the influence of dynamic typing and multiple inheritance/hardwired instance variable offsets, two typographical conventions mark code used to support one of these functions. *Italic code* supports multiple inheritance, and **bold code** supports dynamic typing. Appendix A collects the assembly call sequences and instruction scheduling.

3.2 Dynamic techniques

Dynamic techniques speed up message lookup by using various forms of caching at run-time. Therefore, they depend on locality properties of object-oriented programs: caching will speed up programs if the cached information is used often before it is evicted from the cache. This section discusses two kinds of caching: global caching (one large cache per system) and inline caching (one one-entry cache per call site).

[1] This is done in SELF [16]. Indirect instance variable access is used in Eiffel and Sather [102].

3.2.1 Global lookup caches (LC)

First-generation Smalltalk implementations relied on a global cache to speed up method lookup [60,86]. The class of a receiver, combined with the message selector, hashes into an index in a global cache. Each cache entry consists of a class, a selector and a method address. If the current class and selector match the ones found in the entry, the resident method is executed. Otherwise, a dispatch table search finds the correct method, and the new class-selector-method triple replaces the old cache entry (direct-mapped cache). Any hash function can be used; to obtain a lower bound on lookup time, we assume a simple exclusive OR of receiver class and selector.

```
entry = cache[(object->class ^ #selector) & #mask];
if (entry.class == object->class && entry.selector == #selector) {
  entry.func(object, arguments); /* cache hit */
} else { /* cache miss: use DTS to find method, and update cache entry e */; }
```

Figure 9. Global lookup cache. [1]

Even though LC is considerably faster than dispatch table search (DTS), it still has to compute a hash function for each dispatch. As we shall see, this computation renders LC too slow compared to other techniques. However, LC is a popular fallback method for inline caching, and has recently been revived for dispatching of interface messages in Java.

3.2.2 Inline caches (IC)

Often, the type of the receiver *at a given call site* rarely varies; if a message is sent to an object of type *X* at a particular call site, it is likely that the next send will also go to an object of type *X*. For example, several studies have shown that the receiver type at a given call site remains constant 95% of the time in Smalltalk code [35][127][126]. This locality of type usage can be exploited by caching the looked-up method address at the call site. Because the lookup result is cached “in line” at every call site (i.e., no separate lookup cache is accessed in the case of a hit), the technique is called *inline caching*.

The previous lookup result is cached by changing the call instruction implementing the send, i.e., by modifying the compiled program on the fly. Initially, the call instruction calls the system’s lookup routine. The first time this call is executed, the lookup routine finds the target method. Before branching to the target, the lookup routine changes the call instruction to point to the target method just found (Figure 10). Subsequent executions of the send directly call the target method, completely avoiding any lookup. Of course, the type of the receiver could have changed, and so the prologue of the called method must verify that the receiver’s type is correct and call the lookup code if the type test fails.

Inline caches are very efficient in the case of a cache hit: in addition to the function call, the only dispatch overhead that remains is the check of the receiver type in the prologue of the target. The dispatch cost of inline caching critically depends on the hit ratio. In the worst case (0% hit

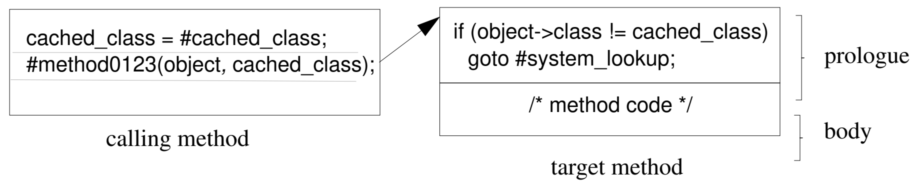


Figure 10. Inline cache

ratio) it degenerates to the cost of the technique used by the system lookup routine (often, a global lookup cache), plus the extra overhead of the instructions updating the inline cache. Fortunately, hit ratios are usually very good, on the order of 90-99% for typical Smalltalk or SELF code *[127]*, *[70]*. Therefore, many current Smalltalk implementations incorporate inline caches.

3.2.3 Polymorphic inline caching (PIC)

Inline caches are effective only if the receiver type (and thus the call target) remains relatively constant at a call site. Although inline caching works very well for the majority of sends, it does not speed up a polymorphic call site[1] with several equally likely receiver types because the call target switches back and forth between different methods, thus increasing the inline cache miss ratio. The performance impact of inline cache misses can become severe in highly efficient systems. For example, measurements of the SELF-90 system showed that it spent up to 25% of its time handling inline cache misses [68].

Polymorphic inline caches (PICs) [68] reduce the inline cache miss overhead by caching *several* lookup results for a given polymorphic call site using a dynamically-generated PIC routine. Instead of just switching the inline cache at a miss, the new receiver type is added to the cache by extending the stub routine. For example, after encountering receiver classes A and B, a send of message *m* would look as in Figure 11.

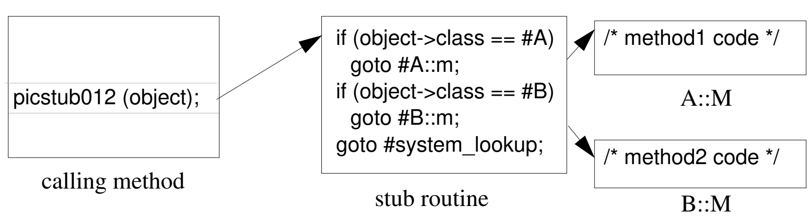


Figure 11. Polymorphic inline cache

A system using PICs treats monomorphic call sites like normal inline caching; only polymorphic call sites are handled differently. Therefore, as long as the PIC's dispatch sequence (a sequence of ifs) is faster than the system lookup routine, PICs will be faster than inline caches.

[1] We will use the term "polymorphic" for call sites where polymorphism *actually* occurs. Consequently, we will use "monomorphic" for call sites with only a single receiver type for the entire program run.

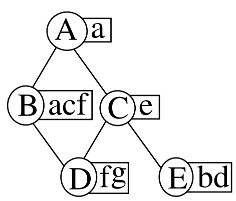
However, if a send is megamorphic (invokes many different methods), it cannot be handled efficiently by PICs. Fortunately, such sends are the exception rather than the rule.

3.3 Static techniques

Static method lookup techniques precompute their data structures at compile time (or link time) in order to minimize the work done at dispatch time. Typically, the dispatch code retrieves the address of the target function by indexing into a table and performing an indirect jump to that address. Unlike lookup caching (LC), static methods usually don't need to compute a hash function since the table index can be computed at compile time. Also, dispatch time usually is constant[1], i.e., there are no "misses" as in inline caching.

3.3.1 Selector table indexing (STI)

The simplest way of implementing the lookup function is to store it in a two-dimensional table indexed by class and selector codes. Both classes and selectors are represented by unique, consecutive class or selector codes; if a system has *c* classes and *s* selectors, classes are numbered *0..c-1* and selectors are numbered *0..s-1* (Figure 12). Unfortunately, the resulting dispatch table is very large (*O(c*s)*) and very sparse, since most messages are defined for only a few classes. For example, about 95% of the entries would be empty in a table for a Smalltalk image [40].



	a	b	c	d	e	f	g
A	0						
B	3		1			2	
C	0				4		
D	0		1		4	5	6
E	0	7		8	4		

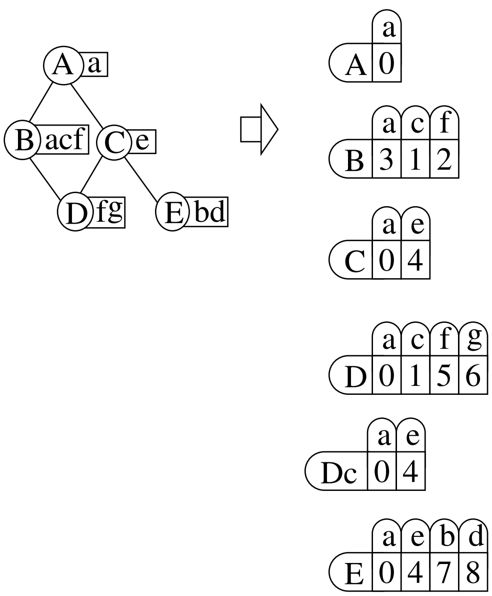
Figure 12. Selector Table Indexing

STI works equally well for static and dynamic typing, and its dispatch sequence is fast. However, because of the enormous space cost, no real system uses selector table indexing. All of the static techniques discussed below try to retain the idea of STI (indexing into a table of function pointers) while reducing the space cost by omitting empty entries in the dispatch table.

[1] Here, "constant time " means "constant number of instructions executed," not "constant in real time" (due to processor implementation effects, such as cache and branch prediction misses).

3.3.2 Virtual function tables (VTBL)

Virtual function tables were first used in Simula [32] and today are the preferred mechanism for virtual function call resolution in Java and C++ [53]. Instead of assigning selector codes globally, VTBL assigns codes only within the scope of a class. In the single-inheritance case, selectors are numbered consecutively, starting with the highest selector number used in the super class. In other words, if a class *C* understands *m* different messages, the class's message selectors are numbered *0..m-1*. Each class receives its own dispatch table (of size *m*), and all subclasses will use the same selector numbers for methods inherited from the super class. The dispatch process consists of loading the receiver's dispatch table, loading the function address by indexing into the table with the selector number, and jumping to that function (non-italic code in Figure 13).




```
object += #tableOffset;
entry = object->table[#selector];
object += entry.delta;
entry.func(object, arguments);
```

Figure 13. VTBL dispatch tables and method call code.

With multiple inheritance, keeping the selector code correct is more difficult. For the inheritance structure on the left side of Figure 13, functions *c* and *e* will both receive a selector number of 1 (they share the second column) since they are the second function defined in their respective class. *D* multiply inherits from both *B* and *C*, creating a conflict for the binding of selector number 1. In C++ [53], the conflict is resolved by using multiple virtual tables per class. An object of class *D* has two dispatch tables, *D* and *Dc* (see Figure 13).[1] Message sends will use dispatch table *D* if the receiver object is viewed as a *B* or a *D* and table *Dc* if the receiver is

[1] Due to limited space, we ignore virtual base classes in this discussion. They introduce an extra overhead of a memory reference and a subtraction [53].

viewed as a *C*. As explained in Section 3.1.2, the dispatch code will also adjust the receiver address before calling a method defined in *C* (the *italic* code in Figure 13)

VTBL depends on static typing: without knowing the set of messages sent to an object, the system cannot reuse message numbers in unrelated classes (such as using 0 for the first method defined in a top-level class). Thus, with dynamic typing, VTBL dispatch tables would degenerate to STI tables since any arbitrary message could be sent to an object, forcing selector numbers to be globally unique.

3.3.3 Selector coloring (SC)

Selector coloring, first proposed by Dixon et al. [38], and applied to Smalltalk by Andre and Royer [9], is a compromise between VTBL and STI. SC is similar to STI, but instead of using the selector to index into the table, SC uses the selector's *color*. The color is a number that is unique within every class where the selector is known, and two selectors can share a color if they never co-occur in a class. SC allows more compaction than STI, where selectors never share colors, but less compaction than VTBL, where a selector need not have a single global number (i.e., where the selector *m* can have two different numbers in unrelated classes).

	a	b	c	d	e	f	g
A	0						
B	3		1			2	
C	0				4		
D	0		1		4	5	6
E	0	7		8	4		

⇨

	a	b / c	d / f	e	g
A	0				
B	3	1	2		
C	0			4	
D	0	1	5	4	6
E	0	7	8	4	

```
entry = object->table[#color];
entry.func(object, #selector, args)
```

```
/* method prologue */
if (S != #mySelector)
    error("Message Not Understood");
```

Figure 14. Selector coloring tables, method call code, and method prologue (dynamic typing in **bold**)

Optimally assigning colors to selectors is equivalent to the graph coloring problem[1] which is NP-complete. However, efficient approximation algorithms can often approach or even reach the minimal number of colors (which is at least equal to the maximum number of messages understood by any particular class). The resulting global dispatch table is much smaller than in

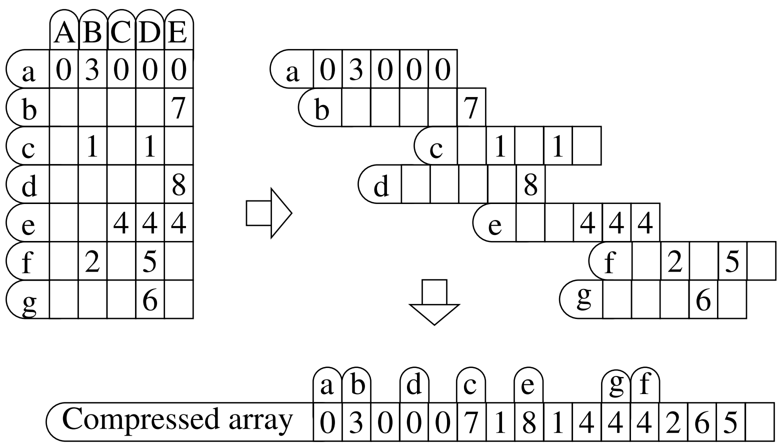
[1] The selectors are the nodes of the graph, and two nodes are connected by an arc if the two selectors co-occur in any class.

STI but still relatively sparse. For example, 43% of the entries are empty (i.e., contain “message not understood”) for the Smalltalk system [40]. As shown in Figure 14, coloring allows the sharing of columns of the selector table used in STI.

Compared to VTBL, SC has an important advantage: since selector colors are global, SC is applicable to a dynamically-typed environment since any particular selector will have the same table offset (i.e., color) throughout the system and will thus invoke the correct method for any receiver. To guard against incorrect dispatches, the prologue of the target method must verify the message selector, and thus the selector must be passed as an extra argument. Otherwise, an erroneous send (which should result in a “message not understood” error) could invoke a method with a different selector that shares the same color. For example, in Figure 14, message *c* sent to a *E* object would invoke *b* without that check.

3.3.4 Row displacement (RD)

Row displacement, treated in depth in Section 4, is another way of compressing STI’s dispatch table. It slices the *transposed* two-dimensional STI table into rows and fits the rows into a one-dimensional array so that non-empty entries overlap only with empty ones (Figure 15). Row




```
entry = object->table[#class];
entry.func(object, #selector, arguments);
```

```
/* method prologue */
if (S != #mySelector) error(“Message Not Understood”);
```

Figure 15. Row displacement tables, method call code, and method prologue

offsets must be unique (because they are used as selector identifiers), so no two rows start at the same index in the master array. The algorithm’s goal is to minimize the size of the resulting master array by minimizing the number of empty entries; this problem is similar to parse table minimization for table-driven parsers [34]. We first studied row displacement of the non-transposed STI table, a technique which generates a slice per class, using a selector numbering scheme that leaves only 33% of the entries empty for the Smalltalk image[39]. If the table is sliced according to selectors (as shown in Figure 15), the number of empty entries is reduced to 0.4% with an appropriate class numbering scheme [43]. We will discuss in depth why selector-based tables (row displacement of the *transposed* STI table) lead to better compression in Section 4. Like SC, RD is applicable to dynamically-typed languages. As in SC, a check is needed in the method prologue, this time to ensure that the method actually is part of the dispatch table of the receiver's class. Therefore, the selector number is passed as an argument to the method.

3.3.5 Compact selector-indexed dispatch tables (CT)

The third table compaction method, proposed by Vitek and Horspool [128], unlike the two previous methods, generates selector-specific dispatch code sequences. The technique separates selectors into two categories. *Standard selectors* have one main definition and are only overridden in the subclasses (e.g., *a* and *b* in Figure 16. *Conflict selectors* have multiple definitions in unrelated portions of the class hierarchy (e.g., *e* in Figure 16 which is defined in the unrelated classes *C* and *D*). CT uses two dispatch tables, a main table for standard selectors and a conflict table for conflict selectors.

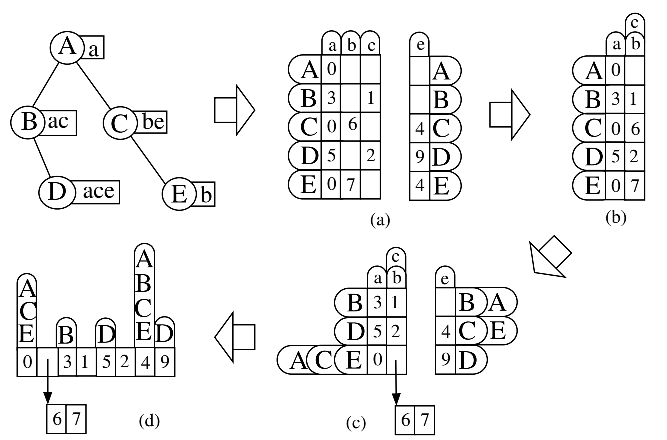


Figure 16. Construction of Compact Tables
(a)main dispatch table and conflict tables for *e*
(b)main dispatch table with *c* and *b* aliased
(c)conflict tables for (*A, B*) and (*C, E*) shared, main dispatch tables for (*A, C, E*) overloaded
(d) dispatch table fitted on a single master array, main dispatch table for (*A, C, E*) trimmed

Standard selectors can be numbered in a simple top-down traversal of the class hierarchy; two selectors can share a number as long as they are defined in different branches of the hierarchy. Such sharing is impossible for conflict selectors, and so the conflict table remains sparse. But the allocation of both tables can be further optimized. First, tables with identical entries (such as the conflict tables for *C* and *E*) can be shared. Second, tables meeting a certain similarity criterion—a parameter to the algorithm—can be *overloaded*; divergent entries refer to a code stub which selects the appropriate method based on the type (similar to PIC). In Figure 16 (a), the entry for selectors *c* and *b* of tables (*A, C, E*) is overloaded. The required level of similarity affects the compression rate (stricter requirements decrease the compression rate) as well as dispatch speed (stricter requirements decrease the number of overloaded entries and thus improve dispatch speed). Finally, dispatch tables are trimmed of empty entries and allocated onto one large master array as shown in Figure 16 (b).

Each class needs at least three fields: class identifier (*cid*), main dispatch table, and conflict table. Because of compression, all methods need a subtype test in the method prologue in dynamically-typed languages. For statically-typed languages, only the code stubs of overloaded entries need such a test. Subtype tests are implemented with a simple series of logical operations (a bit-wise AND and a comparison) [129]. Figure 17 shows the code for a call through a CT dispatch table.

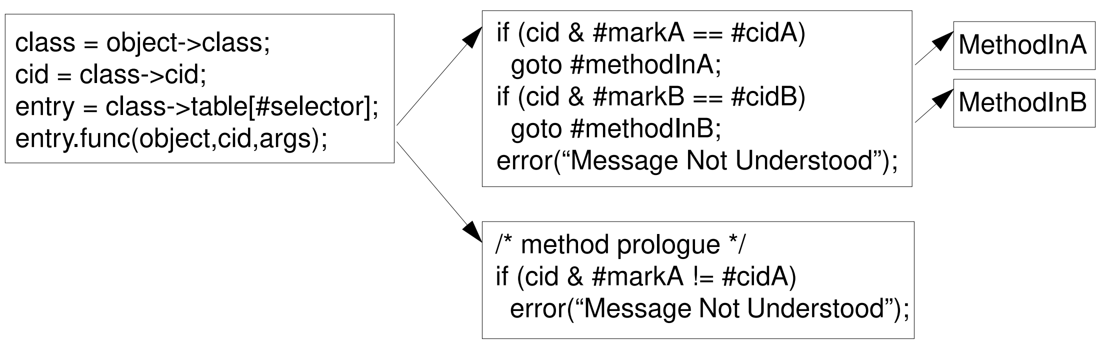


Figure 17. CT dispatch code.
The caller code calls a stub for overloaded entries (upper box). Single implementation entries only require a prologue (lower box) for dynamic typing.

This version of the algorithm (from [128]) only handles single inheritance, because of the lack of fast type inclusion test for multiple inheritance.[1]

[1] Krall, Vitek and Horspool tackle his problem in [85]. Vitek and Horspool [129] present a version of the algorithm which improves dispatch speed and shortens the calling sequence on average.

3.4 Memory cost

The space overhead of method dispatch falls into two categories: program code and dispatch data structures. Code overhead consists of the instructions required at call sites (see the assembly code sequences in Appendix A) and in method prologues; stub routines (PIC & CT) are counted towards the data structure cost. The analysis below ignores per-instance memory costs (such as keeping a type field in each instance), although such costs can possibly dominate all other costs (e.g., if more than one VTBL pointer is needed for a class with a million instances). The space analysis uses the parameters shown in Table 3. Most parameter values are

Variable	Value	Comments
m	8,540	total number of methods; from Smalltalk
c	35,042	total number of call sites; from Smalltalk
M	178,264	total number of valid (receiver class, selector) pairs; from Smalltalk
e	4096	entries in LC lookup cache
O_{DTS}	133%	DTS overhead factor = #total entries / #non-empty entries; from Smalltalk
O_{SC}	175%	single inheritance overhead factor for SC; lower bound, from Smalltalk
O_{RD}	101%	single inheritance overhead factor for RD; from [43]
O_{CT}	15%	single inheritance compression rate for CT [128]
k	3.2	average number of cases in a PIC stub; from SELF [71]
f	7.2%	polymorphic call sites, as a fraction of total; from SELF [71]
a	3.49	average number of functions in an overloaded entry (CT)
n	0.07%	overloaded entries, as fraction of total (CT)

Table 3. Parameters used for space cost analysis

taken from the ParcPlace Visualworks 1.0 Smalltalk system and thus model a fairly large application.

3.4.1 Single inheritance cost

Our analysis assumes that dispatch sequences are compiled in-line; the cost per call site is taken from the code sequences listed in Appendix A.[1] Code for secondary techniques (like LC for IC) is not counted since it only appears once and thus should be negligible. Table 4 shows the space cost computation for all techniques. In the formulas, the symbols D and C refer to data and code cost; D_{LC}, for instance, refers to the data structure cost of LC in *the same column*.

Figure 18 shows the space costs for single inheritance versions of the dispatch techniques, using the classes and methods of the ParcPlace Visualworks 1.0 Smalltalk system as an example.

[1] We choose not to include the subroutine call instruction in each dispatch sequence in the space cost since this instruction is required for direct function calls as well. To include the call instructions, just add c to each entry in Table 4.

	static typing		dynamic typing	
	code	data	code	data
DTS	2c	2m*O_{DTS}	same as static typing	
LC	14c	3e+D_{DTS}	same as static typing	
IC	3c+2m	D_{LC}	same as static typing	
PIC	3c+2m	3kfc+D_{LC}	same as static typing	
VTBL	2c	M	N/A	
SC	2c	M*O_{SC}	3c+2m	M*O_{SC}
RD	5c	M*O_{RD}	5c+4m	M*O_{RD}
CT	4c	M*O_{CT} *(1+an)	4c+7m	M*O_{CT} *(1+an)

Table 4. Formulas for approximate space cost (in words)

Surprisingly, the code space overhead dominates the overall space cost for six of the eight techniques. Most of that overhead consists of the per-call dispatch code sequence. Much of the literature has been concentrated on the size of dispatch tables, treating call code overhead as equivalent among different techniques (Dave Ungar's Smalltalk system evaluation is a notable exception [127]). As demonstrated by the above data, minimizing dispatch tables may not reduce the overall space cost if it lengthens the calling sequence, especially in languages with a high density of message sends, like Smalltalk.[1] Code size can be reduced for most techniques by moving some instructions from the caller to the callee, but only at the expense of a slower dispatch. (LC's code size requirements could be dramatically reduced by doing the lookup out-of-line.)

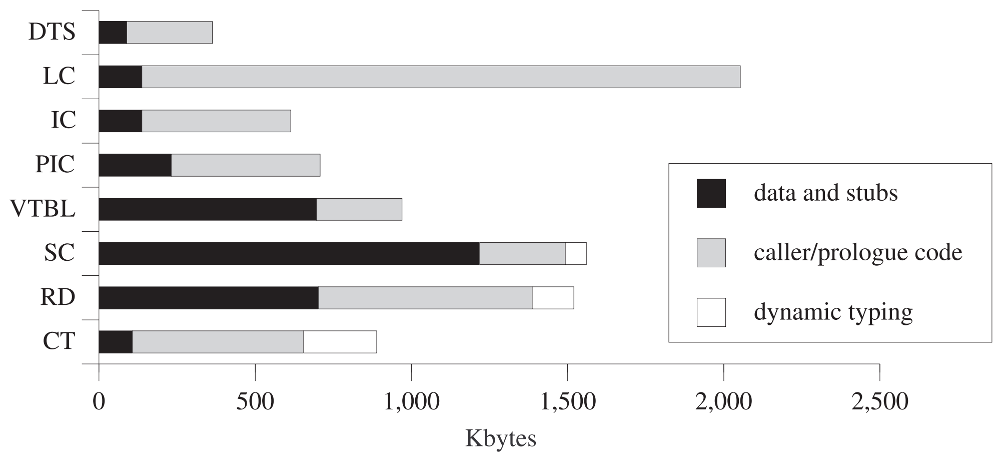


Figure 18. Space overhead for the Smalltalk system

The size of the immediate field in an instruction significantly impacts the code cost of SC, RD, and CT. This study assumes a 13-bit signed immediate field, limiting the range of immediates

[1] Increased code size can also impair execution performance by causing more instruction cache misses.

to -4096..4095.[1] The Smalltalk system measured had 5087 selectors, and thus the selector number fits into an immediate. SC needs only one instruction to load the selector code into a register (see Table A-4), but RD takes two instructions for the same action because the selector number is a memory address, pointing to the beginning of a selector table. The same phenomenon increases the method prologue overhead in both RD and CT.[2] In RD, the reduction in data structure size relative to SC is almost offset by a corresponding increase in code size. The data in Figure 18 are thus relative to the processor architecture. For example, for an architecture with larger immediates (or for smaller applications), CT's space advantage over VTBL would double. Of course, the data also depends on application characteristics such as the proportion of call sites versus number of classes, selectors, and methods.

Given these admonitions, IC and PIC apparently combine excellent average speed with low space overhead. The bounded lookup time of SC and RD is paid for with twice as much memory; VTBL is about one third smaller than those two. CT's small data structure size is offset by its code cost.

VTBL, RD, and SC require significantly more data space than DTS because they duplicate information. Each class stores all the messages it *understands*, instead of all the messages it *defines*. For example, in the Smalltalk system a class inherits 20 methods for each one it defines [40], so the number of entries stored in the class' dispatch table increases by a factor of 20.

Dynamic typing makes a relatively small difference in space cost. Dynamic techniques have no extra overhead because each dispatch already contains a run-time check to test for the cache hit. Static techniques[3] perform the run-time type check in the method prologue, so the overhead grows linearly with the number of defined methods, which is much smaller than the number of call sites.

3.4.2 Multiple inheritance cost

The space overhead of multiple inheritance is more difficult to measure than single inheritance space cost. For VTBL-MI, the call sequence consists of two extra instructions, and virtual function tables have an extra entry which stores the object pointer adjustment (see Section 3.1.2). This brings the code cost to 4c, and the data structure cost to 2M. However, every time a class inherits from an ancestor though different paths (the "diamond" problem), all functions inherited through that ancestor are stored twice (with different adjustments). This overhead depends entirely on the way multiple inheritance is used and is not quantifiable without appropriate code metrics. In [43], we calculated this overhead for two class libraries that use multiple inheritance extensively. The generated tables stored, respectively, 215% and 330% of the entries that a

[1] The size of immediates varies from architecture to architecture: for example, SPARC has 13 bits, Alpha 8 bits, and MIPS 16 bits.

[2] Here the crucial quantity is the number of bits necessary to represent a *cid* (16 bits for the Smalltalk example). For the same reason, CT's dynamic typing cost is higher.

[3] Excluding VTBL, which only works for statically-typed languages.

single table would use (assuming non-hard-coded offsets). Finally, the overhead of having multiple table pointers in each object can only be calculated for individual program runs, since it depends on the number of object allocated.

Static techniques (SC, RD) with one table per object (or per message selector), exhibit the compiled code cost of dynamic techniques, plus an additional overhead because multiple inheritance hierarchies generate tables that are harder to compress. For SC, multiple inheritance introduces conflicts between selector colors that are hard to deal with and that substantially increase the number of empty entries [9]. For two libraries with extensive use of multiple inheritance (Every class has on average two parent classes), the resulting tables contained 71% and 74% empty entries. For RD, multiple inheritance causes more irregular empty regions to appear in the original two-dimensional table, which makes the rows harder to fit tightly together. RD is more robust than both VTBL-MI and SC, however, since the resulting tables contained only 9% and 29% empty entries (see Section 4.3.3).

3.5 Programming environment aspects

The choice of a dispatch technique is influenced by considerations other than space cost and execution speed (discussed in Section 5). Code organization and demands put upon the programming environment can preclude some techniques from being used. The following aspects, though non-comprehensive, give an idea of the issues involved:

- *Responsiveness*. Some static techniques are less suitable for interactive programming environments that demand a high level or responsiveness (i.e., immediate feedback to programming changes). Most static techniques need to recompute some data structures after a programming change. For example, introducing a new method name forces VTBL to rearrange the dispatch table of the affected class and all its subclasses. SC, RD and CT even have to completely recompute their tables from scratch (except in rare cases, e.g., if the new method fits into an empty slot in the dispatch table). For the Smalltalk system, complete recomputation can take hours for SC, and seconds for RD and CT.

- *Support for run-time extensibility.* Many static techniques (e.g., SC, RD, CT) presume knowledge of the entire program, i.e., knowledge of all classes and methods. With dynamic link libraries (DLL), the complete set of classes and methods is not known until run time since the classes contained in dynamic libraries are unknown until they are loaded at program start-up time or later during the execution of the program. Thus, these techniques have to recompute the dispatch data structures each time a new library is dynamically loaded.

- *Sharing code pages*. Some operating systems (e.g., Unix) allow processes executing the same program to share the memory pages containing the program's code. With shared code pages, overall memory usage is reduced since only one copy of the program (or shared library) need be in memory even if it is used concurrently by many different users. For code to be shared, most operating systems require the code pages to be read-only, thus disallowing techniques that modify program code on-the-fly (e.g., IC and PIC).

Many of the dispatch techniques discussed here can be modified to address problems such as those outlined above. For example, static techniques can be made more incremental by introducing extra levels of indirection at run-time (e.g., by loading the selector number rather than embedding it in the code as a constant), usually at a loss in dispatch performance. For this study, only the simplest and fastest version of each technique was considered, but any variant can be analyzed using the same evaluation methodology.

3.6 Summary

Instructions (in particular, per-call code) can contribute significantly to the overall space requirements of message dispatch. In our example system, many techniques spend more memory on dispatch code sequences than on dispatch data structures. Thus, minimizing dispatch table size may not always be the most effective way to minimize the overall space cost, and may in some cases even increase the overall space cost.

Run-time system constraints can limit the choice of dispatch techniques available to practitioners. For static caching techniques, table construction time can be prohibitive. Inline caching techniques require modifiable code.

4 Row displacement compression

"I visualize structures, graphs, data structures. Its seems to come a easier than a lot of other things"

Robert E. Tarjan [115]

Table-based dispatch is attractive because its cost is bound by a constant. The critical path of selector table indexing (STI), the simple, fast, but memory-greedy static technique, consists of two dependent loads and an indirect branch instruction. On older processors, this sequence takes about 5 cycles, 4 if the branch delay slot can be filled. On newer processors, the sequence takes in the worst case (branch mispredict), two load latencies plus a branch misprediction penalty (about 8 cycles on the P96 processor used as state-of-the-art representative in Section 6).

Therefore, the simplest form of table-based dispatch is fast and reliable. Its main problem is its prohibitive memory cost. A straightforward implementation uses a two-dimensional table of size S * C, where S is the number of message selectors (10K for SELF) and C the number of classes (1.8K in SELF). For mature programming environments, this table simply takes too much memory (18M addresses = 72Mbytes on a SPARC). Since each class responds to only a fraction of all message selectors (typically between 2% and 10%), the table could be brought down to a reasonable size, if only the non-null entries were stored (1M addresses for SELF = 4Mbytes on a SPARC).

In statically typed languages like C++ or Java, virtual function tables (VTBL) avoid storage of null entries because of a numbering scheme that appends new entries to an existing table. VTBL dispatch takes no more instructions than STI. With single parent class inheritance, this scheme is optimal. No null entries are stored. When multiple inheritance is employed, the numbering scheme does not guarantee zero overhead because of the necessity to merge two or more tables. However, table compression becomes expedient only with high occurrence of multiple inheritance. Multiple inheritance incurs a small overhead of two extra instructions in the dispatch sequence, but these instructions are independent of the critical path instructions, and can therefore be hidden in the extra instruction issue of a superscalar processor[1].

In dynamically typed languages, each class must respond to all messages, since the compiler does not guarantee that a particular object implements a message sent to it. This "message not understood" error is caught at run time. Semantically speaking, the default implementation of all messages is *MessageNotUnderstood*. Table compression of dispatch tables aims to store only the non-default implementations (the 2% to 10% non null entries in the STI table). Absence of an implementation must be detected by the dispatch code at run time, and trigger the invocation of *MessageNotUnderstood*. For the table-based dispatch we discuss in this chapter,

[1] However, the dependency of those instructions on the availability of the receiver's address can incur delays in some programs (see section 6.4.2)

one extra instruction in the call sequence is necessary, and two instructions in the target procedure prologue that check for the default case. Since *MessageNotUnderstood* is a program bug, most of the time this check fails, so the conditional branch is very predictable. Therefore, only a small overhead is usually added to the dispatch cost (one or two cycles), resulting in a bound of about ten cycles, when a indirect branch misprediction penalty is taken.

The remaining problem in table-based dispatch for dynamically typed languages is how to achieve a high compression rate. In the asymptotically best case, the tables store no null entries using only as much memory as VTBL's in statically typed languages. In this section, we show how get within 1% of this bound for a broad selection of large single-inheritance class libraries. The compression scheme is then generalized to handle multiple inheritance, reducing the null-entry overhead for those libraries to less than 10% for all but one benchmark. Finally, we discuss implementations of the row displacement algorithm that allow dispatch table construction of these large samples to take place in a few seconds.

4.1 Class-based row displacement

Class-based row displacement dispatch tables, introduced as "sparse arrays" [39][1], compress the two-dimensional selector index table of Figure 12 by slicing it according to classes and fitting the resulting rows into a one-dimensional array. The offset of a row in the compressed array serves as a class id. After compression, the sum of class offset and selector number gives the index of the desired entry in the compressed array.

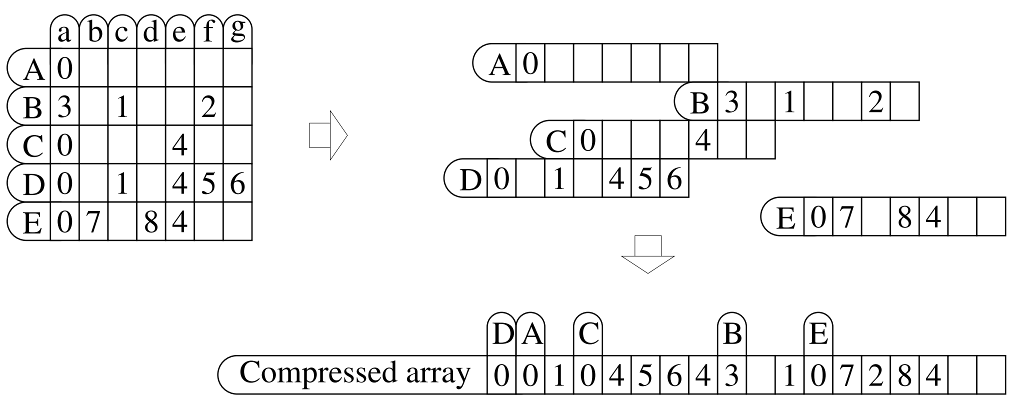


Figure 19. Class-based row displacement tables.

In this setup, we can improve the compression rate by choosing selector numbers (column numbers) so that the rows become easier to fit together. Initially, we used an ad-hoc heuristic:

[1] In [34], the term "row displacement compression" was coined for a very similar approach in parser table compression. We also want to avoid confusion with [122], where the term "sparse array" is used for a one-dimensional array that is compressed by dividing it in equally large chunks and not storing the empty ones.

we numbered the message selectors defined in meta-classes in the Smalltalk library first. These classes are numerous (half of the total number of classes), and each one understands a large number of messages (300 on average, compared to 160 for regular classes), taken from a limited set of selectors (about 1300, compared to 4000 for regular classes). Therefore, first numbering those selectors sequentially results in a large number of rows with densely packed entries, making the fitting process efficient. After fitting meta-classes, the wide, sparsely filled regular class rows caused a resulting overhead of 40% [39].

We then designed a heuristic based on a generalization of this ad-hoc rule for Smalltalk classes. This results in a better compression rate, but 33% of the table still stores undefined entries [40].

4.2 Selector-based row displacement

Selector-based row displacement functions the same way as class-based row displacement, but we slice the two-dimensional table up according to selectors, instead of classes. Figure 20 illustrates the process on the same class hierarchy as used in Figure 19: the compression algorithm works on the *transposed* selector index table. At run time, the lookup process is similar, with the role of classes and selectors reversed. However, for the "message not understood"-test, the method still tests whether the called and defined *selector* are equal.

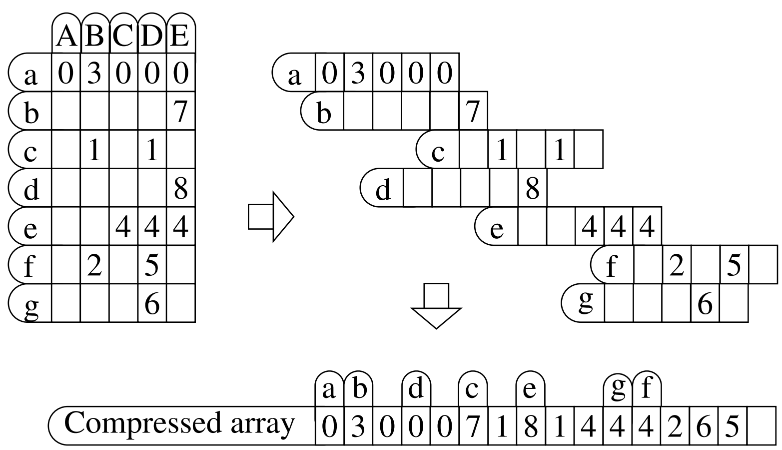


Figure 20. Selector-based row displacement tables

In this setup, the numbering of classes determines the shape of the rows to be fitted in the compressed array. This simple change dramatically improves the fill rate (i.e., reduces the size) of the compressed array. Selector-based tables have less overhead than class tables for two reasons: their row size distribution makes it easier to tightly pack rows together and their message definitions cluster in a more regular way under inheritance, which reduces the amount of gaps within rows. Both of these aspects can be exploited by an appropriate class numbering scheme, as explained in the next sections.

4.2.1 Distribution of row sizes

Figure 21a shows the two-dimensional dispatch table for the Smalltalk class *Magnitude* and its subclasses. Although this example is an artificial collection of classes (it is not a complete

Figure 21. Table size distribution for *Magnitude* (18 classes, 240 selectors).
(a) STI table from alphabetically sorted classes and selectors.
(b) selector-based table size: occupied entries in black, intermittent spaces in gray, leading and trailing spaces in white (not visible). (c) same as (b), but sorted by size
(d) and (e) same as (b) and (c), but class-based

program), it does demonstrate the characteristic size distributions found in larger, realistic samples.

To begin with, the number of classes is much smaller than the number of message selectors. Therefore the average size of class-based rows will be much larger than that of selector-based rows. In [41], we found that row displacement compression works better, in general, with many small rows than with few large ones. Thus a selector-based table is likely to be compressed better than a class-based table. Although not all the cases we tested exhibit as large a difference as *Magnitude*, usually the number of selectors is at least twice the number of classes. This asymmetry accounts for part of the better fill rate of selector tables.

In addition to the average row size, the *distribution* of row sizes also has an impact on the fill rate. Figure 21b illustrates why. Figure 21b is constructed from 21a by squeezing occupied entries (in black) together.[1] The width of the black rows thus indicates the number of occupied entries of that row. The gray area represents unoccupied space that is not leading or trailing the row, but is enclosed by occupied entries to its left and right. Thus black and gray area together give the maximum space that a row can occupy under row displacement. We call this quantity the width of a row vector. The white area is free because leading and trailing empty space can overlap with other rows. Figure 21c is the same as 21b, but with selectors reordered by size to better show the distribution. Figures 21d and 21e show the same for a class-based table, rotated 90 degrees.

The black area represents a lower bound on the size of the compressed array; black and gray together represent an upper bound. If all rows were placed consecutively, the size of the compressed array would be the sum of all the row widths. Gray area can thus be considered *potential* overhead.

When comparing Figure 21c with Figure 21e, it becomes clear why a selector-based table does better than a class-based table: almost one third of the selector-based rows have only one occupied entry. These ultra-small rows are ideal to fill up gaps left by other rows. Class-based rows on the other hand have a minimal size of nine. The difference tends to get worse as libraries grow larger. For example, in VisualWorks 1.0 every class understands at least 100 messages, but 45% of the messages are only known by one class, and another 39% by less than ten classes. Furthermore, the potential overhead (gray area) for the selector-based table in Figure 21 is much smaller than that of a class-based table. Thus a selector-based table has less area to fill up, and better material to fill it up with.

Without further efforts, selector-based row displacement already outperforms the most sophisticated class-based row displacement schemes. For *Magnitude*, the best fill rate reached in [40] is 80%, while the structure shown in Figure 21b gives a compressed array that is 87% occupied. This difference becomes larger as class libraries grow larger, as we will see in section 4.3.

---

[1] This figure does not correspond to an actual data structure in the compression algorithm. It is given only to illustrate the characteristics of rows in a selector-based table.

4.2.2 Fitting order

The previous section demonstrated that class- and selector-based tables have different row size distributions. This section and the next explains how to exploit this difference to its fullest in an actual implementation.

Since it is not practical to look for the best possible configuration, i.e., to look for offsets that minimize total overhead,[1] we fill the compressed array in one pass. Rows are inserted into the first place that fits, but we still have the freedom to choose the order in which rows are inserted. Figure 21b shows the alphabetical order, which is an arbitrary one from an algorithmic perspective; as mentioned before, it achieves 87% fill rate. Figure 21c shows the rows ordered by size. We start with the largest rows and work from there to the smaller ones. Intuitively, this arrangement gives smaller rows a better chance to fill up holes left by bigger ones. For *Magnitude* the fill rate does indeed improve to 93%. Larger samples exhibit a smaller difference but consistently favor the size-ordered scheme.

Surprisingly, sorting rows according to descending size does not slow down the algorithm. Because the time needed to fit all rows is proportional to the average number of unoccupied entries that is checked before a fitting space is found, a denser compression is reached in a shorter time. Thus the gain in speed caused by the better fill rate compensates for the extra time needed to sort rows.

4.2.3 Class numbering for single inheritance

Now that we have established the ordering of selectors, this section will show how classes can be numbered to enhance the fill rate further.

If we look back at Figure 20, exchanging column D and C makes the fitting process trivial, because no gaps are left in any rows. All empty entries are found either in the beginning or the end of a row. Merely placing these rows back to back in the compressed array solves the compression problem. A class numbering scheme that minimizes the number of gaps in a selector-based table is illustrated in Figure 22.

The aim is to make sure that all classes that understand a certain message have consecutive numbers. Our scheme numbers classes by traversing the inheritance tree in depth-first preorder. This numbering scheme ensures that every subtree in the inheritance structure corresponds to a consecutive block of class numbers. Since most message selectors are understood by exactly a subtree of classes, most rows in the selector-based table consist of exactly one consecutive block of occupied entries. For the single-inheritance hierarchy of Figure 22, the new class numbering reduces the number of gaps from four to one. The only selector with a gap in its row is *e*, because *e* is defined separately in two classes (*D* and *F*), without being defined in a common super class.

[1] This problem is NP-complete [121] and thus takes too long to solve, or even to approximate [41].

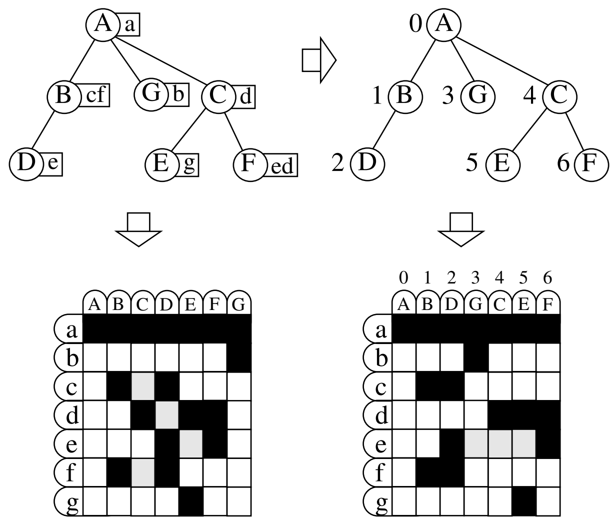


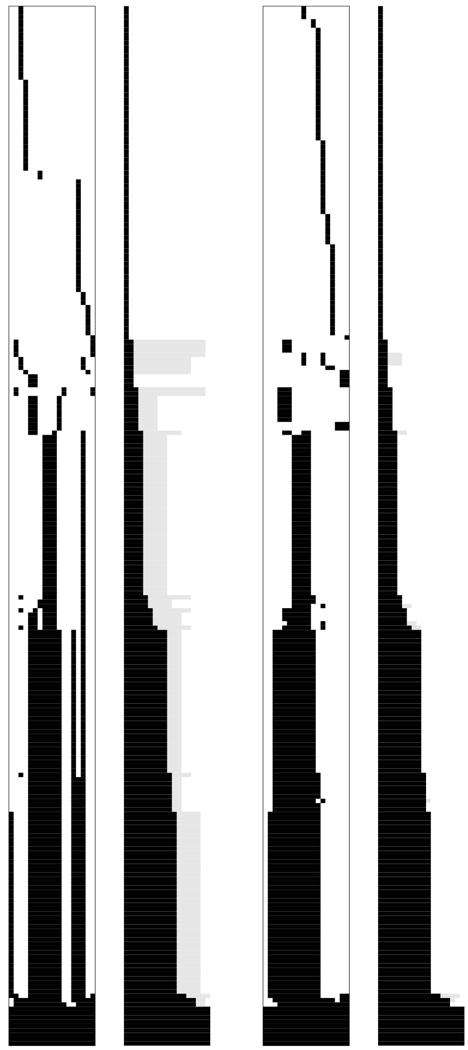
Figure 22. Alphabetic and depth-first class numbers

Figure 23 shows the effect of the numbering scheme on *Magnitude*. Figure 23a shows a selector-based table, with the rows ordered in increasing size. 35% of the total area is potential overhead (gray). As mentioned before, this table resulted in 93% fill rate, so 7% of the 35% became real overhead. After renumbering classes, only 1.6% potential overhead is left (Figure 23b), and compression reduces this gray area to 0.5% real overhead. Out of an array with 1388 entries, only 3 are unoccupied, and even these gaps are caused by contention over row offsets by one-entry rows, not by true fitting conflicts (remember that all rows need to have unique offsets in the compressed array). Later, in section 4.3, we will present compression results for several other single inheritance class libraries.

4.2.4 Class numbering for multiple inheritance

The depth-first numbering scheme can easily be applied to a multiple inheritance class library. The only difference to single inheritance is that a class with more than one direct super class (a.k.a. base class) will be visited more than once. Since a class is numbered the first time it is encountered, its number will depend on the order in which subclasses are traversed. We found that an essentially random choice is good enough if multiple inheritance is not used frequently, as in the *Unidraw/Interviews* sample of section 4.3.

However, if multiple inheritance is used extensively, it is worthwhile to spend time on a better numbering scheme. We construct a single-inheritance hierarchy from the multiple inheritance hierarchy by considering only the dominant super class link. The dominant super class is the class that makes the largest contribution to the dispatch table. For example, in Figure 24, class *B* is the dominant super class of class *E* because *E* inherits more messages through *B* than



(a) (b)

Figure 23. *Magnitude*: selector-based tables
with classes numbered:
(a) alphabetically
(b) depth-first preorder

through any other of its super classes. It is easy to see why this choice produces the fewest number of gaps: if a class inherits from several classes, it will be numbered out of sequence in the subtrees of all classes except one.[1] Thus it will cause a gap in the rows of all message selec-

[1] Exception: if a class number appears at the edge of a consecutive block it may, by pure chance, be adjacent to the number of another of its base classes.

tors it inherits, except the ones inherited through its dominant base class. Therefore, choosing the class through which a class inherits the largest number of messages avoids more gaps than any other choice.

Figure 24 shows the effect of two different choices for a small example. Note that this rule minimizes the *number* of gaps, not necessarily the sum total of gap space (gray area). In fact, on the largest multiple inheritance sample, *Geode*, our method enlarges gap space by 13%, but increases fill rate by 11.2%.

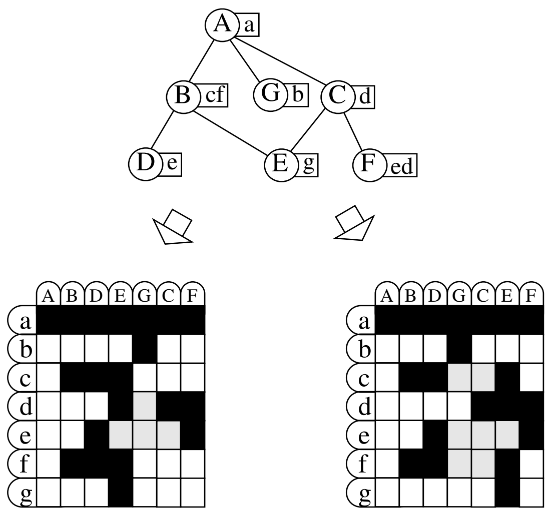


Figure 24. Multiple inheritance class numbering
E is numbered as subclass of B (left) and as subclass of C (right). Because selectors c and f are inherited through B, and only d is inherited through C, choosing B avoids two gaps and causes one.

4.2.5 Summary

In general, row displacement compression is a difficult combinatorial optimization problem. However, appropriate heuristics exploit the regularities that inheritance imposes on dispatch tables, and give excellent compression rates in practice. These "rules of thumb" are the following:

- Slice the class/selector table by selector instead of by class, because there are many more selectors than classes and most selectors are understood by only a few classes. These characteristics give rise to many small rows, which are easier to fit tightly together.

- Fit rows by decreasing size, to give small rows the opportunity to fill up gaps left by larger rows.

- Number classes according to a depth-first pre-order traversal of the inheritance structure, to force occupied entries to cluster together.
- For multiple inheritance libraries: ignore all base classes in the numbering scheme, except the base class which understands the largest number of messages.

4.3 Compression results

In this section we compare four variants of selector-based row displacement with other table-based message dispatch techniques on a number of large class libraries. We will first discuss the way space overhead is calculated, then briefly outline the test samples, and finally discuss the results.

4.3.1 Methodology

To evaluate the effectiveness of table compression techniques, we measure how close the table size approaches *M*: the sum, over all classes, of the number of messages understood by a class. This sum is the total of legitimate class-selector combinations of a class library. A message dispatch technique that finds a method in a constant, small amount of time needs to store each one of these combinations. We define the *fill rate* of a technique as *M* divided by the actual number of entries for which storage is allocated. For instance, dividing *M* by the product of the number of classes and the number of selectors calculates the fill rate of the two-dimensional class/selector table employed by STI (see Section 3.3.1).

Selector coloring (SC) [38][9] expresses table compression as a graph coloring problem. The graph represents selectors by nodes. An edge between two nodes means that the corresponding two selectors occur in the same class. The aim of the coloring algorithm is to assign a color to each node of the graph so that adjacent nodes have different colors, with as few colors as possible. What this technique boils down to in terms of the two-dimensional dispatch table is the following: selector coloring compresses the table by overlapping columns. Every selector is assigned a column number. Two selectors can share a column if none of their occupied entries have the same index (i.e., they do not occur together in any class).

A lower bound for the number of columns is the size of the largest row. This row corresponds to the class that understands the largest number of messages. Multiplying this number by the total number of classes gives a lower bound to the number of entries of the resulting data structure. Dividing *M* by this lower bound then gives an upper bound to the fill rate, which is the quantity we show under column SC. It is independent of the coloring algorithm used[1].

Virtual function tables (VF) [32], the preferred implementation of message dispatch in C++ compilers [53], have no overhead for single inheritance class libraries. Multiple inheritance incurs space overhead because every base class requires its own virtual function table, dupli-

[1] Since graph coloring is NP-complete, optimal coloring schemes are approximated by polynomial-time algorithms.

cating entries that are common to two or more base classes. For example, in Figure 24, class *E* has two virtual function tables, both of which store *a*. The size of a class's virtual function tables equals the sum of its parents' virtual tables plus the number of newly defined (not overridden) messages.

For class-based row displacement (CR) we took the fill rate reached by the heuristic described in [40]. This heuristic performed best on all samples except Object, where Horn's algorithm reached a better fill rate (67% instead of 64%).

Selector-based row displacement is shown for three variations in class numbering: alphabetical (AL), depth-first pre-order traversal for a single-inheritance hierarchy (SI), and depth-first pre-order traversal for a multiple-inheritance hierarchy (MI). Fill rates are shown for the fastest implementation of row displacement, which trades some fitting tightness for speed, as explained in section 4.4.1.

4.3.2 Samples

We choose sample class libraries taken from real systems for three reasons: it facilitates comparison with other methods, it eliminates the extra step of verifying whether the real world behaves the same as the artificial samples, and as an aside it gives us the opportunity to gather some statistics from programs with hundreds of classes, which is interesting in its own right.

On the other hand, the data points do not cover the realm of possible class libraries as evenly as we would wish. Self-contained programs are usually large, consisting of several hundred classes. To get an impression of how the different techniques behave on smaller examples, we also used subsets of the Smalltalk inheritance structure. These are the first six samples of Table 5. The next six samples are comprised of all the classes of different Smalltalk images: Parcplace Visualworks 1.0 and 2.0, Digitalk Smalltalk 2.0 and 3.0, and VisualAge Smalltalk 2.0, kernel and complete library. The next two samples are NextStep, written in Objective-C, and the SELF system 4.0. The two C++ samples are the ET++ Application Framework [131] and Unidraw/InterViews [92]. Only the latter uses multiple inheritance, in only 10 classes (the average number of base classes is smaller than 1 because 147 of the classes in this library have no base class). The last two samples are from LOV, a proprietary language based on Eiffel. These two make extremely heavy use of multiple inheritance: on average *every* class inherits from two super classes.

4.3.3 Compression results

Table 5 shows the result of our measurements. Selector-based row displacement performs very well on single-inheritance samples (all samples with "-" in column P). Fill rates are higher than 99.5% for all self-contained examples, and more than 98% for the smaller ones. The technique scales up well: contrary to selector coloring and class-based row displacement, compression improves as libraries grow in size. The class numbering scheme is partly responsible for this

								Other techniques			Selector-based row displacement		
System	Library	C	S	M	m	P	2D	SC	VF	CR	AL	SI	MI
Parcplace Smalltalk	Set	9	94	450	144	-	53	65	-	81	90	98.3	-
	Stream	16	126	1,122	210	-	56	65	-	82	93	99.7	-
	Magnitude	18	240	1,381	568	-	32	52	-	78	93	99.2	-
	Collection	51	402	4,926	805	-	24	64	-	59	81	99.0	-
	VisualComponent	53	529	4,253	875	-	15	60	-	56	91	98.8	-
	Object w/o metaclasses	383	4,026	61,775	6,835	-	4.0	62	-	48	95	99.4	-
	Object (Parcplace1)	774	5,086	178,230	8,540	-	4.5	57	-	64	77	99.6	-
	Parcplace2	1,956	13,474	608,456	23,720	-	2.3	57	-	55	72	99.7	-
Digitalk Smalltalk	Digitalk ST/V 2.0	534	4,482	154,585	6,853	-	6.5	43	-	56	78	99.6	-
	Digitalk ST/V 3.0	1,356	10,051	613,654	17,097	-	4.5	42	-	50	71	99.8	-
IBM Smalltalk	IBM Smalltalk 2.0	2,320	14,009	485,321	25,994	-	1.5	32	-	63	56	99.5	-
	VisualAge 2.0	3,241	17,342	1,045,333	37,058	-	1.9	43	-	55	47	99.7	-
Objective-C	NextStep	310	2,707	71,334	4,324	-	8.5	53	-	51	89	99.6	-
SELF	Self System 4.0	1,801	10,103	1,038,514	29,411	1.02	5.7	60	-	47	67	99.7	99.8
C++	ET++	370	628	14,816	1,746	0.76	6.4	29	100	78	46	97.6	97.6
	Unidraw/Interviews	613	1,146	13,387	3,153	0.78	1.9	23	100	62	44	95.6	95.7
LOV	Lov+ObjectEditor	436	2,901	36,052	5,007	1.78	2.9	29	46.5	52	64	75.2	91.1
	Geode	1,318	6,555	302,717	14,202	2.11	3.5	26	30.3	45	58	57.9	70.8

Table 5. Compression results (in fill rate %)

C: number of classes S: number of selectors M: total of legitimate class-selector combinations m: total number of defined methods P: average number of parents per class
Other techniques:
2D: uncompressed 2-dimensional class/selector tableSC: selector coloring
VF: virtual function tablesCR: class-based row displacement
Selector-based row displacement for different class numbering schemes:
AL: alphabetical order SI: single-inheritance scheme MI: multiple-inheritance scheme

trend, as fill rates decrease similarly to selector coloring, though not as fast, when classes are numbered alphabetically. For dynamically-typed languages, no other method comes close to the fill rate of selector-based row displacement with depth-first class numbering.

Multiple inheritance samples come in two kinds. If multiple inheritance is rarely used, the results are similar to those of single inheritance. Virtual function tables have no overhead for single inheritance (i.e., a 100% fill rate), and therefore perform best on such samples, with row displacement a close second. With heavy use of multiple inheritance, fill rates decrease for all methods, but by different amounts. As anticipated in [39], selector coloring does not handle multiple inheritance well. Virtual function tables do a little better. Selector-based row displacement has the best fill rate, though it also starts to show substantial overhead for the largest sample.

To conclude, our experiments show that selector-based row displacement tables have a very low space overhead compared to other table-based methods. The technique scales up well, improving the fill rate with larger class libraries. Finally, it also handles multiple inheritance in a robust way, outperforming other methods by a factor of two if multiple inheritance is heavily used.

4.4 Optimizing table compression speed

In this section we describe the algorithm used to achieve adequate performance of table compression.

4.4.1 Compression algorithms

We went through a number of implementations of the dispatch table compression algorithm. In a nutshell, this algorithm assigns to each row an offset o in the compressed array, so that two conditions hold:

- o is not shared with any other row
- For every non-empty entry at index i, $o+i$ is not shared with any non-empty entry of any other row

Section 4.2 explained how a class numbering scheme determines the indices within each row. Rows are fitted in one run in order of decreasing size. Now the problem is to find an offset for a row in a partially filled compressed array in the least possible amount of time. This problem is reminiscent of allocating a block of memory in a memory management system (see [8] for an overview), with the added complication that blocks are fragmented.

Figure 20 suggests a simple algorithm for fitting a row (represented by a collection of indices i): start with offset $o = 0$, check if all entries $o + i$ are empty. If not, increment o and continue until a match occurs. Then check if offset o is in the set of offsets that are already used by fitted rows. If so, continue the search; if not, insert the row at o and add o to the set of used offsets.

A simple improvement of the algorithm hops over used space. Each unused entry in the compressed array stores the index of the next empty entry. The algorithm follows these links instead of checking every possible offset. This reuse of free memory is similar to the “freelist” concept in memory management systems. By utilizing free memory resources to keep track of free memory, the only cost associated with a smarter algorithm is the time required for maintenance of the freelist. When a row is fitted into the compressed array, the entries it occupies are removed from the freelist. This saves time, since a used entry is never checked again, while in the simple algorithm it is checked once for each row that fits to its right. The algorithm outlined so far comprises the SIO algorithm in Table 6, which stands for "singly-linked freelist with index-ordered single-entry rows". We will explain the latter denotation later.

Although rows can be arbitrarily fragmented in principle, in practice they usually consist of a large block with a few "satellite" pieces if an adequate class numbering scheme is used, as demonstrated in section 4.2. Therefore, smarter, faster allocation schemes can be adapted from memory management techniques that deal with variable sized blocks. The key trick is to test first for the largest consecutive block of a row, and to organize the freelist so that one can easily enumerate free blocks with a certain minimum size. Then the highly fragmented free space which tends to accumulate in the filled portion of the compressed array is skipped when fitting large rows. In [40], we implemented and tested an algorithm (BBBF) built around a freelist which was actually a binary search tree, ordered by size. The main disadvantage of this technique is that free blocks must have a size of six or more consecutive entries or they cannot be linked into the tree. Blocks of size five and smaller are not linked at all. This not only complicates the maintenance of the free list, but, more importantly, renders the algorithm impractical for selector-based rows. As demonstrated in section 4.2, the majority of selector-based rows is smaller than six, whereas class-based rows have a minimum size far exceeding that. The algorithm in [40] is also more general than necessary, since it can deal with arbitrarily large blocks. The maximum block size in this particular problem is equal to the size of the largest row, which is the number of selectors in the system (10K in Self, a large system).[1]

Here we use the simpler approach of linking together equally sized blocks. An array indexed by block size contains pointers to the beginning of each separate freelist. The algorithm proceeds as follows: to fit a row, first determine the largest consecutive block of indices. We call this the primary block. The row is represented by the first index and the length of the primary block, and the list of remaining indices. Start with the non-empty freelist with block size greater than or equal to the size of the primary block. Run through this freelist and test, for each offset that positions the primary block within the current free block, whether the remaining indices match. If no match is found, try the freelist with the next larger block size. Compared to the singly-linked freelist algorithm, a match test is more efficient since the entries of the primary block do not need to be checked. Moreover, no time is wasted on free blocks smaller than the primary block.

When a match occurs, the current block is removed from its freelist, and, if it is larger than the primary block, the space that remains left or right is inserted in the freelist of the appropriate size. The free blocks over which the remaining indices are positioned are also split up. Then the row is copied into the compressed array.

To remove them efficiently from their respective freelists, blocks have to be doubly-linked. This implies that the minimum size for a free block is two. After all tables with primary block size of two or greater are inserted, the algorithm reverts to the singly-linked freelist algorithm outlined before, to fit the remaining single-entry tables. In Table 6, the complete algorithm is

[1] Except for the huge chunk of free space in the right part of the compressed array, which can be dealt with separately.

denoted by DIO, which stands for “doubly-linked freelist with index-ordered single-entry tables”. It performs better than SIO in a number of cases, but for Smalltalk samples in particular, it is still puzzlingly slow.

Profiling revealed that the algorithm spent an excessive amount of time (up to 80%) checking uniqueness of row offsets in the Smalltalk samples. This check only happens after a match is

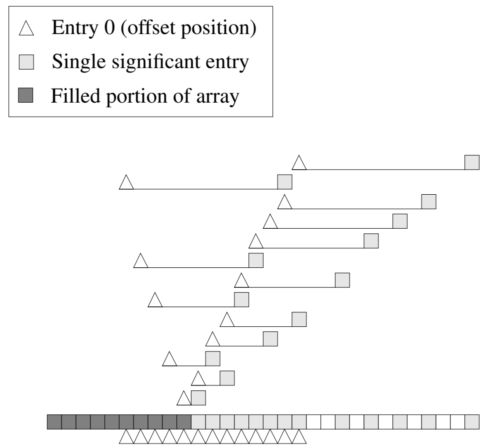


Figure 25. One-entry row fitting

Rows with entries from 1 to 13, if fitted in that order, cause a dense packing of offsets. The last row checks on 8 free positions before a free offset is found.

found, and a hash table implementation makes it an efficient operation. Moreover, it should almost always succeed, since only one in about fifty positions in the compressed array is an offset. However, the offsets are not spread randomly. Single-entry rows are fitted last to fill the holes in the occupied part of the compressed array. If there are more of such rows than necessary to fill the remaining empty space, they cluster at the right end. Finding open space for a one-entry row is trivial, but finding a unique offset becomes time-consuming, as illustrated in Figure 25.

We tried reordering the tables in a number of ways to prevent the offsets from clustering prematurely. Rows with more than one entry are still sorted in decreasing size, and ties are broken by putting rows with smaller width first. For single-entry rows, the most spectacular improvement in speed, for a modest decrease in fill rate, is reached by ordering the single-entry rows in decreasing index order (of the only non-nil element). In Table 6, the resulting algorithm is indicated by DRO, which stands for “doubly-linked freelist with reverse index-ordered single-entry rows”.

4.4.2 Compression speed

Table 6 shows the fill rates of the three variants, and timings as the average over 20 runs. Cache

					Fill rate (in %)			Timing (in seconds)		
System	Library	C	S	M	SIO	DIO	DRO	SIO	DIO	DRO
Parcplace Smalltalk	Object w/o metaclasses	383	4,026	61,775	99.5	99.5	99.4	0.3	0.9	0.2
	Object (Parcplace1)	774	5,086	178,230	99.7	99.7	99.6	1.7	4.3	0.4
	Parcplace2	1,956	13,474	608,456	99.7	99.7	99.7	14.6	25.3	2.1
Digitalk Smalltalk	Digitalk2	534	4,482	154,585	99.7	99.7	99.6	0.8	3.3	0.7
	Digitalk3	1,356	10,051	613,654	99.8	99.8	99.8	5.9	13.5	1.6
IBM Smalltalk	Smalltalk	2,320	14,009	485,321	99.8	99.8	99.5	32.3	11.4	5.2
	VisualAge2	3,241	17,342	1,045,333	99.7	99.7	99.7	167.3	13.6	12.0
Objective-C	NextStep	310	2,707	71,334	99.7	99.7	99.6	0.5	2.1	0.2
SELF	Self System 4.0	1,801	10,103	1,038,514	99.8	99.8	99.8	9.4	2.4	2.5
C++	ET++	370	628	14,816	98.5	98.4	97.6	0.04	0.05	0.04
	Unidraw/Interviews	613	1,146	13,387	97.6	95.8	95.7	0.2	0.1	0.05
LOV	Lov+ObjectEditor	436	2,901	36,052	95.8	91.1	91.1	0.8	0.3	0.2
	Geode	1,318	6,555	302,717	74.9	70.8	70.8	49.0	9.6	9.0

Table 6. Compression speed (in seconds, on a 60Mhz SPARCstation-20)
C: number of classesS: number of selectorsM: total of legitimate class-selector combinations
SIO: singly-linked freelist with index-ordered single-entry rows
DIO: doubly-linked freelist with index-ordered single-entry rows
DRO: doubly-linked freelist with reverse index-ordered single-entry rows

effects caused a variation of less than 3%. We omitted the smaller samples because the time was too small to be reliably measured.

For all but the C++ and LOV samples, the fill rate is largely independent of the particular algorithm used. Fill rates vary slightly because the ordering of single-entry tables trades memory for speed. The difference between single and doubly-linked freelists, which is most pronounced in the Geode sample, is caused by the different order in which offsets are checked. SIO puts a row in the first place that fits, starting from the left edge of the compressed array. DIO and DRO also go from left to right, but start with the freelist with the smallest block size. There may be larger blocks to the left of a smaller block, causing a table to be fitted further to the right than strictly necessary.

The fastest algorithm in almost all cases is DRO. For samples from the same programming environment, there seems to be a linear relation between the number of classes and the time needed to compress the tables. For SIO, the relation appears quadratic (i.e., twice as many classes take four times as long).

Absolute performance is excellent, especially compared to previous techniques. For example, the Object sample takes 0.4 seconds to compress (on a SPARCstation-20) compared to the fastest class-based row displacement algorithm in [40] which took 36 minutes on a Mac IIfx.

For Object, selector coloring on a Sun-3/80 took about 80 minutes to build the conflict graph and 12 minutes to color it [9]. These timings are measured on different hardware and with different compilers, so that they cannot really be compared directly. However, we believe it is safe to say that on equivalent hardware, selector-based row displacement compression constructs dispatch tables at least an order of magnitude faster than these previous techniques. Furthermore, the data shows that row displacement compression is now a practical technique. Though compression may still take too long for an interactive environment, it can be postponed, as outlined in the next section.

4.5 Applicability to interactive programming environments

In an ideal interactive programming environment any change to a program is reflected without noticeable delay. Subsecond response time is required to optimize programmer productivity. As discussed above, both selector coloring and row displacement compression are not fast enough to hide global refitting of dispatch tables from the user.

Selector-based tables improve on class-based tables, since a global reorganization is only necessary when a new class is defined, because this adds a column to the two-dimensional table and thus affects all rows. The definition of a new message just adds a row. Presuming that the compressed array has space for it, this does not affect the other rows. For class-based tables, the situation is similar but reversed: defining a new message can affect all rows and cause reorganization. Since new messages are defined more often than new classes, selector-based row displacement is better tuned to a development environment.

Still, when it occurs, global reorganization can be painful. As outlined in [39], global refitting can be postponed until there is time and opportunity, by having a second-stage table that is searched when the main table would deliver a "message not understood" error. For selector-based tables, this second table holds the newly defined classes. Message sends to instances of new classes are slower than normal, until the classes are incorporated in the main table.

Thus selector-based row displacement tables can be employed in an interactive programming environment, if the table fitting costs are postponed by using a second-stage table, and at the cost of slower dispatch for recently defined classes.

4.6 Summary

Selector-based row displacement makes table-based message dispatching practical for dynamically-typed languages, due to the following two properties:

- *Compact dispatch* tables. For large single-inheritance class libraries, row displacement tables are less than 0.5% larger than virtual function tables. If multiple inheritance is used extensively, row displacement outperforms virtual function tables by a factor of two on the tested samples.

- *Fast dispatch table construction*. On a 1995 midrange workstation, compression takes less than 2.5 seconds for most of the large class libraries measured. This performance makes the technique practical even for interactive programming environments, especially if a second-stage dispatch table is used to postpone global reorganization.

Thus, row displacement dispatch tables provide the dispatch efficiency of virtual function tables, with comparable memory cost and low compile time overhead, for dynamically-typed object-oriented programming languages.

For multiple inheritance languages, row displacement performs better than virtual function tables by more than a factor two (in remaining overhead)

5 Analysis of dispatch sequences on modern processor architectures

"Fallacy: increasing the depth of pipelining always increases performance."
Hennessy and Patterson [65]

Dispatch cost is intimately coupled with processor implementation. The same dispatch sequence may have different cost on different processor implementations, even if all of them implement the same architecture (e.g., the SPARC instruction set). In particular, processor pipelining and superscalar execution make it impossible to use the number of instructions in a code sequence as an accurate performance indicator. This section characterizes the run-time performance of dispatch mechanisms on modern pipelined processors by determining the performance impact of branch latency and superscalar instruction issue. We do this by analyzing call code sequences, optimally scheduled for the given instruction issue. In addition to providing specific numbers for three example architectures, our analysis allows bounds on dispatch performance to be computed for a wide range of possible (future) processors. With the rapid change in processor design, it is desirable to characterize performance in a way that makes the dependence on certain processor characteristics explicit, so that performance on a new processor can be estimated accurately as long as the processor's characteristics are known.

5.1 Parameters influencing performance

To evaluate the performance of the dispatch mechanisms, we implemented the dispatch instruction sequence of each technique on a simple RISC-like architecture.[1] Then, we calculated the cost of the dispatch sequences for three hypothetical processor implementations. P92 represents a scalar implementation as it was typical of processor designs in 1992. P95 is a superscalar implementation that can execute up to two integer instructions concurrently, representative of current state-of-the art processor designs. Finally, P97 is an estimate of a 1997 superscalar processor with four-instruction issue width and a deeper pipeline. Table 7 lists the detailed processor characteristics relevant to the study.

In essence, these processors are abstractions of current commercial processors that have been reduced to their most important performance features, namely

- *Superscalar architecture*. The processor can execute several instructions in parallel as long as they are independent. Since access paths to the cache are expensive, all but P97 can execute at most one load or store per cycle.

[1] Assuming a RISC architecture simplifies the discussion, but our results are not restricted to RISC architectures since pipelined CISC machines have similar performance characteristics. For example, the Intel Pentium processor closely resembles P95.

- *Load latency.* Because of pipelining, the result of a load started in cycle i is not available until cycle $i + L$ (i.e., the processor will stall if the result is used before that time).

- *Branch penalty.* The processor predicts the outcome of a conditional branch; if the prediction is correct, the branch incurs no additional cost. However, if the prediction is incorrect, the processor will stall for B cycles while fetching and decoding the instructions following the branch [65]. We assume that indirect calls or jumps cannot be predicted and always incur the branch penalty.[1]

	P92	P95	P97
max. integer instructions/ cycle	1	2	4
max. loads or stores / cycle	1	1	2
max. control transfers (branch, call) / cycle	1	1	1
load latency (L)[a]	2	2	2
branch prediction	no	yes	yes
branch miss penalty (B)	1[b]	3	6
examples of equivalent commercial CPUs	[103][31]	[96][63]	N/A

Table 7. Processor characteristics

[a] To simplify the analysis, we assumed L > 1; to the best of our knowledge, this assumption holds for all RISC processors introduced since 1990.

[b] No penalty if the branch's delay slot can be filled. (To improve readability, the instruction sequences in Appendix A are written without delay slots.) On P95/97, delay slots cannot hide branch latency due to multi-cycle branch penalties and superscalar instruction issue, and thus have no performance benefit.

Virtually all processors announced since 1993 exhibit all three characteristics. We also assumed out-of-order execution for the superscalar machines (P95 and P97). To determine the number of cycles per dispatch, we hand-scheduled the dispatch instruction sequences for optimal performance on each processor. In most cases, a single instruction sequence is optimal for all three processors.

Variable	Typical value	Comments
h_{LC}	98%	lookup cache hit ratio ([30] lists 93% for a very small cache size)
$miss_{LC}$	250[a]	LC miss cost (find method in class dictionaries); conservative estimate based on data in [127]
h_{IC}	95%	inline caching hit ratio; from [127] and [68]
$miss_{IC}$	80[a]+L+LC	IC miss cost; from [68]
m	66%	fraction of calls from monomorphic call sites (PIC) [68][14]
k	3.54	dynamic number of type tests per PIC stub (from SELF [71])
p	10%	average branch misprediction rate (estimate from [65])

Table 8. Additional parameters influencing performance

[1] But see section 5.5.

Variable	Typical value	Comments
M	1%	fraction of calls from highly polymorphic call sites (> 10 receiver types); conservative estimate (in SELF, M < 0.1% [71])
$miss_{PIC}$	150[a]+L+LC	PIC miss cost; based on $miss_{IC}$ (overhead for updating the PIC)
s	99.93%	percentage of single (non-overloaded) entries in CT [128]
a	2.25	number of tests per overloaded entry in CT

Table 8. Additional parameters influencing performance

[a] Cycles on P92; 20% less on P95 and 33% less on P97.

The performance of some dispatch techniques depends on additional parameters (listed in Table 8). In order to provide some concrete performance numbers in addition to the formulas, we chose typical values for these parameters (most of them based on previously published performance studies). However, it should be emphasized that these values merely represent one particular data point. Different systems, applications, or languages may well exhibit different parameter values. Thus, the *numbers* below are specific to an example configuration, but the *formulas* in Tables 9 to 11 are general.

5.2 Dispatch cost calculation

Appendix A lists all assembly call sequences analyzed in this study, with their delays for optimally scheduled code. Here we demonstrate the methodology on the most popular table-based technique, VTBL.

5.2.1 VTBL call code scheduling

Figure 26 illustrates the cycle cost calculation for VTBL-MI (which happens to be the same as VTBL). Data dependencies are indicated with arrows, control dependencies with dashed arrows. Instructions handling multiple inheritance are enclosed by dashed circles. The figure shows the order in which instructions are issued into the processor pipeline.[1] An instruction with a dependency on a load instruction executing in cycle i cannot execute before cycle $i + L$ (where L is the load latency). For example, in P92 instruction 2 cannot execute before cycle L because it depends on instruction 1 (Figure 26). Similarly, instruction 5 can execute at $L + L$ or $L + 2$ (one cycle after the previous instruction), whichever is later. Since we assume $L > 1$, we retain $2L$. The schedule for P92 also shows that instruction 3 (which is part of the multiple inheritance implementation) is *free*: even if it was eliminated, instruction 5 could still not execute before $2L$ since it has to wait for the result of instruction 2. Similarly, instruction 4 is free because it

[1] More precisely, the cycle in which the instruction enters the EX stage (this stage calculates the result in arithmetic operations or the effective address in memory operations and branches). For details on pipeline organization, we refer the reader to [65].

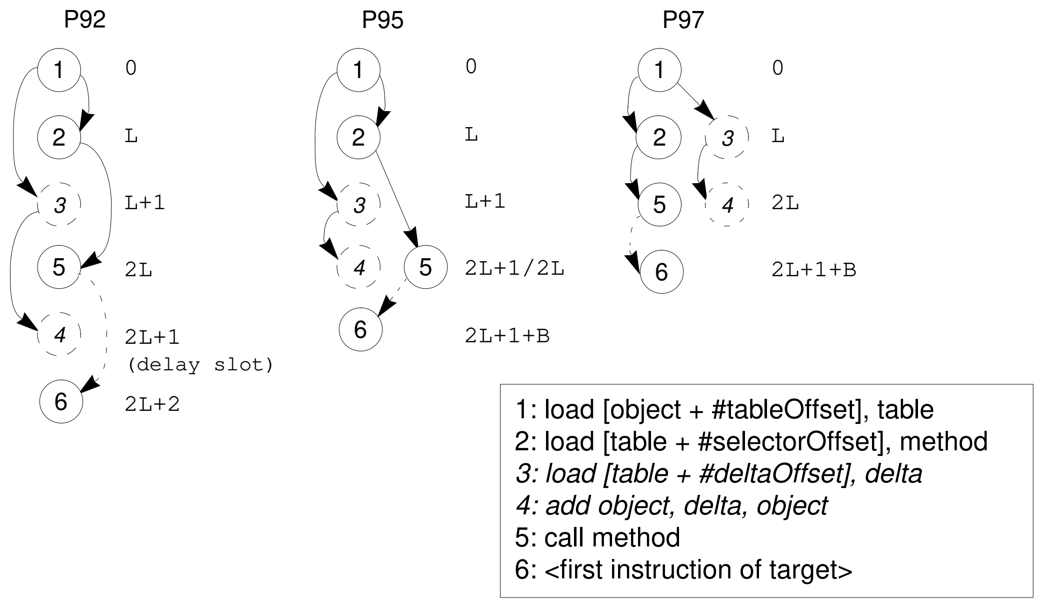


Figure 26. VTBL-MI schedules and dependencies, cycle counts and assembly code.

executes in the delay slot of the call (instruction 5).[1] As a result, VTBL incurs no overhead for multiple inheritance: both versions of the code execute in *2L + 2* cycles (see Table 9).

P95 (middle part of Figure 26) can execute two instructions per cycle (but only one of them can be a memory instruction, see Table 7). Unfortunately, this capability doesn't benefit VTBL much since its schedule is dominated by load latencies and the branch latency *B*. Since VTBL uses an indirect call, the processor does not know its target address until after the branch executes (in cycle *2L*). At that point, it starts fetching new instructions, but it takes *B* cycles until the first new instruction reaches the EX (execute) stage of the pipeline [65], resulting in a total execution time of *2L+B+1*. Finally, P97 can execute up to 4 instructions per cycle, but again this capability is largely unused, except that instructions 2 and 3 (two loads) can execute in parallel. However, the final cycle count is unaffected by this change.

5.2.2 Other techniques

Table A-2 to 8 in Appendix A list call code sequences in assembly for all dispatch techniques, and corresponding optimal code schedules on P92 and P97. We used these schedules to derive the formulas in Tables 9 to 11: we express dispatch cost as a function of processor parameters (L and B) and algorithmic parameters such as miss ratios, etc. Table 12 contains the raw dispatch times in cycles.

---

[1] Recall that P92 machines had a branch latency B = 1, which can be eliminated using explicit branch delay slots; see [65] for details. Since we use a fixed branch penalty for P92, B does not appear as a parameter in Table 4.

	static typing	dynamic typing
LC	h_{LC}*(9+max(7,2L))+(1-h_{LC})*$miss_{LC}$	
VTBL	2+2L	N/A
VTBL-MI	2+2L	N/A
SC	2+2L	4+2L
RD	3+L + max(L, 3)	7+L+max(3,L)
CT	s*(2+3L)+(1-s) *(3+3L+7a)	s*(8+3L)+(1-s) *(3+3L+7a)
IC	h_{IC} * (2+max(3,L)) + (1-h_{IC}) * $miss_{IC}$	
PIC	m * (2+max(3,L)) + (1-m) * (2+L+2k) + M * $miss_{PIC}$	

Table 9. P92

	static typing	dynamic typing
LC	h_{LC} * (7+2L+B) +(1-h_{LC}) * $miss_{LC}$	
VTBL	1+2L+B	N/A
VTBL-MI	1+2L+B	N/A
SC	1+2L+B	3+2L+B
RD	1+2L+B	3+2L+B
CT	s*(1+3L+B)+(1-s)* (2+3L+B+a(4 +pB))	s*(5+3L+B)+(1-s)* (2+3L+B+a(4 +pB))
IC	h_{IC} * (1+L) + (1-h_{IC}) * $miss_{IC}$	
PIC	m * (1+L)+(1-m)*(2+L+k(1 +pB)) + M * $miss_{PIC}$	

Table 10. P95

	static typing	dynamic typing
LC	h_{LC} * (6+2L+B) +(1-h_{LC}) * $miss_{LC}$	
VTBL	1+2L+B	N/A
VTBL-MI	1+2L+B	N/A
SC	1+2L+B	2+2L+B
RD	1+2L+B	3+2L+B
CT	s*(1+3L+B)+(1-s)* (2+3L+B+a(3 +pB))	s*(4+3L+B)+(1-s)* (2+3L+B+a(3 +pB))
IC	h_{IC} * (1+L) + (1-h_{IC}) * $miss_{IC}$	
PIC	m * (1+L) + (1-m) * (1+L+k(1 +pB)) + M * $miss_{PIC}$	

Table 11. P97

Figure 27a shows the execution time (in processor cycles) of all dispatch implementations on the three processor models, assuming static typing and single inheritance. Not surprisingly, all techniques improve significantly upon lookup caching (LC) since LC has to compute a hash function during dispatch. The performance of the other dispatch mechanisms is fairly similar, especially on P95 which models current hardware. VTBL and SC are identical for all processors; RD and VTBL are very close for all but the P92 processor. Among these techniques, no clear winner emerges since their relative ranking depends on the processor implementation. For example, on P92 VTBL performs better than IC, whereas on P97 IC is better. (Section 5.4 will examine processor influence in detail.) For dynamic typing, the picture is qualitatively the same (Figure 27b).

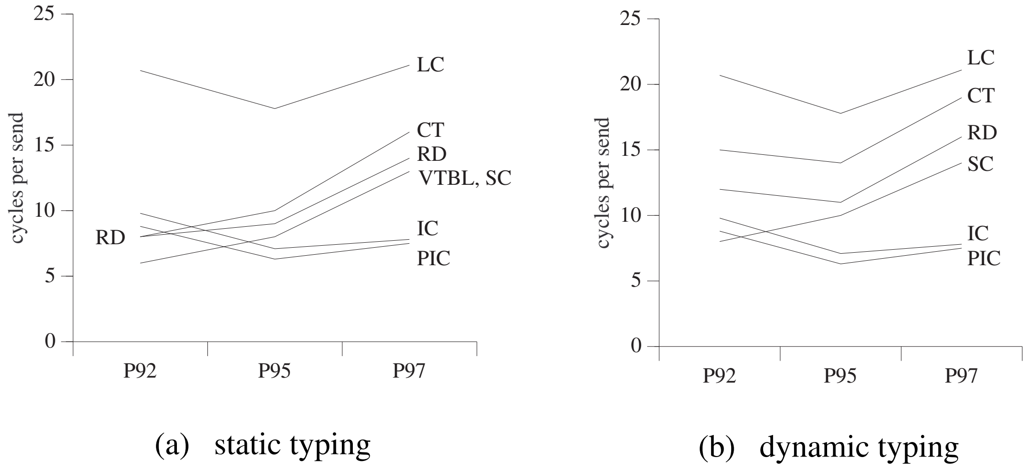


Figure 27. Performance of dispatch mechanisms (single inheritance)

5.3 Cost of dynamic typing and multiple inheritance

A closer look at Tables 9 to 11 and Figure 27 shows that supporting dynamic typing is surprisingly cheap for all dispatch methods, especially on superscalar processors like P95 and P97. In several cases (LC, IC, PIC), dynamic typing incurs no overhead at all. For the other techniques, the overhead is still low since the additional instructions can be scheduled to fit in instruction issue slots that would otherwise go unused. Typical overheads are two cycles per dispatch on P95 and one or two cycles on P97. Thus, on superscalar processors dynamic typing does not significantly increase dispatch cost.

The cycle cost of supporting hard-coded offsets for multiple inheritance in VTBL-MI is zero. However, recall that we have simplified the discussion of VTBL-MI for C++ by ignoring virtual base classes. Using virtual base classes can significantly increase dispatch cost in VTBL-MI. For all other techniques, the cost of multiple inheritance appears not as overhead of message dispatch, but when referencing an instance variable (see section 3.1.2).

Table 12 shows timings as calculated according to the formulas. Since the performance variations between the two scenarios are so small and do not qualitatively change the situation, we will only discuss the case using static typing and single inheritance. Of course, dynamic typing and multiple inheritance can affect other aspects of dispatch implementation, as discussed in section 3.4 and 3.5.

	P92		P95		P97	
	static	dyn.	static	dyn.	static	dyn.
LC	20.7		17.8		21.1	
VTBL	6.0	N/A	8.0	N/A	13.0	N/A
VTBL-MI	6.0	N/A	8.0	N/A	13.0	N/A
SC	6.0	8.0	8.0	10.0	13.0	14.0
RD	8.0	12.0	8.0	10.0	13.0	15.0
CT	8.0	14.0	10.0	14.0	16.0	19.0
IC	9.8		7.1		7.8	
PIC	8.8		6.3		7.2	

Table 12. Dispatch timings (in cycles)

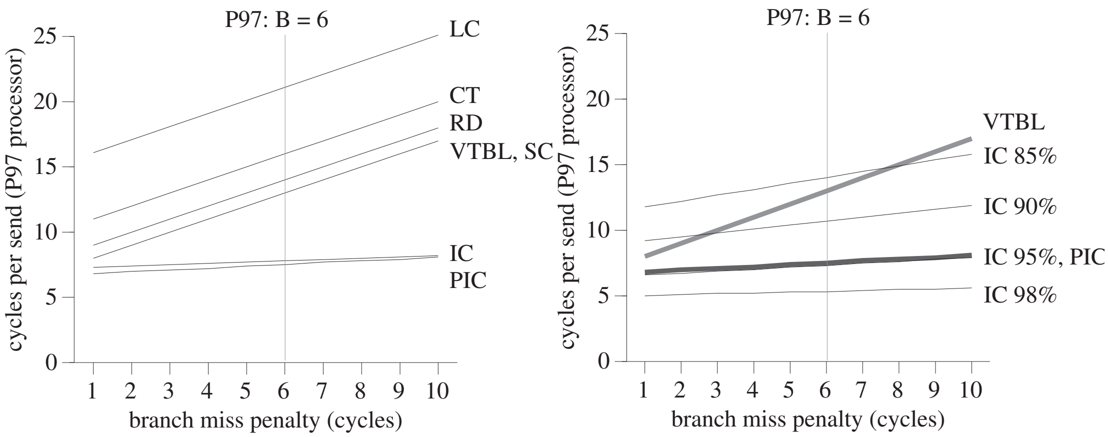
5.4 Influence of processor implementation

According to Figure 27, the cost (in cycles) of many dispatch techniques drops when moving from a scalar processor like P92 to a superscalar implementation like P95. Apparently, all techniques can take advantage of the instruction-level parallelism present in P95. However, when moving to the more aggressively superscalar P97 processor, dispatch cost *rises* for many dispatch techniques instead of falling further as one would expect.[1]

Figure 28a shows that the culprit is the penalty for mispredicted branches. It rises from 3 cycles in P95 to 6 cycles in P97 because the latter processor has a deeper pipeline in order to achieve a higher clock rate and thus better overall performance [65]. Except for the inline caching variants (IC and PIC), all techniques have at least one unpredictable branch even in the best case, and thus their cost increases with the cost of a branch misprediction. IC's cost increases only slowly because it has no unpredicted branch in the hit case, so that it suffers from the increased branch miss penalty only in the case of an inline cache miss. PIC's cost also increases slowly since monomorphic calls are handled just as in IC, and even for polymorphic sends its branches remain relatively predictable.

Based on this data, it appears that IC and PIC are attractive dispatch techniques, especially since they handle dynamically-typed languages as efficiently as statically-typed languages. However, one must be careful when generalizing this data since the performance of IC and PIC depends on several parameters. In particular, the dispatch cost of IC and PIC is variable—unlike most of the table-based techniques such as VTBL, the number of instructions per dispatch is not constant. Instead, dispatch cost is a function of program behavior: different programs will see different dispatch costs if their polymorphism characteristics (and thus their inline cache hit ratios) vary. The data presented so far assume a hit ratio of 95% which is typical for Smalltalk programs [127] but may not represent other systems. For example, Calder et al. [14] report

[1] Even though the number of cycles per dispatch increases, dispatch time will decrease since P97 will operate at a higher clock frequency. Thus, while the dispatch cost rises relative to the cost of other operations, its absolute performance still increases.



(a) all techniques (b) inline caching variants vs. VTBL

Figure 28. Influence of branch misprediction penalty on dispatch cost in P97

inline cache hit ratios for C++ programs that vary between 74% and 100%, with an average of 91%. Thus, the performance characteristics of IC and PIC deserve a closer investigation.

Figure 28b compares VTBL with PIC and IC for several inline cache miss ratios. As expected, IC's cost increases with decreasing hit ratio. If the hit ratio is 90% or better, IC is competitive with static techniques such as VTBL as long as the processor's branch miss penalty is high (recall that P97's branch miss penalty is 6 cycles). In other words, if a 91% hit ratio is typical of C++ programs, IC would outperform VTBL for C++ programs running on a P97 processor.

PIC outperforms VTBL independently of the processor's branch penalty, and it outperforms IC with less than a 95% hit ratio. The performance advantage can be significant: for P97's branch miss penalty of 6 cycles, PIC is twice as fast as VTBL. Again, this result is dependent on additional parameters that may vary from system to system. In particular, PIC's performance depends on the percentage of polymorphic call sites, the average number of receiver types tested per dispatch, and the frequency and cost of "megamorphic" calls that have too many receiver types to be handled efficiently by PICs. On the other hand, PIC needs only a single cycle per additional type test on P97, so that its efficiency is relatively independent of these parameters. For example, on P97 PIC is still competitive with VTBL if every send requires 5 type tests on average. As mentioned in section 3.2.3, the average degree of polymorphism is usually much smaller. Therefore, PIC appears to be an attractive choice on future processors like P97 that have a high branch misprediction cost.

Nevertheless, the worst-case performance of PIC is higher than VTBL, and PIC doesn't handle highly polymorphic code well, so some system designers may prefer to use a method with lower worst-case dispatch cost. One way to achieve low average-case dispatch cost with low worst-case cost is to combine IC with a static technique like VTBL, SC, or RD. In such a system, IC would handle monomorphic call sites, and the static technique would handle polymorphic sites. (Another variant would add PIC for moderately polymorphic call sites.) The combination's efficiency depends on the percentage of call sites that are handled well by IC. Obviously, call sites with only one target fall in this category but so do call sites whose target changes very infrequently (so that the rare IC miss doesn't have a significant performance impact). The scheme's dispatch cost is a linear combination of the two techniques' cost. For example, Calder's data [14] suggest that at least 66% of all virtual calls in C++ could be handled without misses by IC, reducing dispatch cost on P97 from 13 cycles for a pure VTBL implementation to 13* 0.34 + 4 * 0.66 = 5.6 cycles for VTBL+IC. In reality, the performance gain might be even higher since calls from call sites incurring very few misses could also be handled by IC. Even though this data is by no means conclusive, the potential gain in dispatch performance suggests that implementors should include such hybrid dispatch schemes in their list of dispatch mechanisms to evaluate.

5.5 Limitations

The above analysis leaves a number of issues unexplored. Three issues are particularly important: cache behavior, application code surrounding the dispatch sequence, and hardware prediction of indirect branches.

We do not consider memory hierarchy effects (cache misses); all results assume that memory references will always hit the first level memory cache. If all dispatch techniques have similar locality of reference, this assumption should not distort the results. However, without thorough benchmarking it remains unsubstantiated.

Application instructions surrounding the dispatch sequence (e.g., instructions for parameter passing) can be scheduled to fit in the "holes" of the dispatch code, lowering the overall execution time, and thus effectively lowering dispatch overhead. Therefore, measuring dispatch cost in isolation (as done in this study) may overestimate the true cost of dispatch techniques. Unfortunately, the effect of co-scheduling application code with dispatch code depends on the nature of the application code and thus is hard to determine. Furthermore, the average basic block length (and thus the number of instructions readily available to be scheduled with the call) is quite small, usually between five and six [65]. On superscalar processors (especially on P97) most dispatch sequences have plenty of "holes" to accommodate that number of instructions. Thus, we assume that most techniques would benefit from co-scheduled application code to roughly the same extent. In Section 6 we will study two variants of VTBL in order to determine the effect of co-scheduling.

A branch target buffer (BTB) [65] allows hardware to predict indirect calls by storing the target address of the previous call, similar to inline caching. The study in this section assumes that processors do not use BTBs. Therefore, the formula's for table-based techniques calculate a bound on the cost of message dispatch (ignoring memory effects). This worst case cost demonstrates the importance of accurate indirect branch prediction in hardware. BTBs are relatively expensive (since they have to store the full target address, not just a few prediction bits). However, future processors are likely to incorporate BTBs since they will have enough transistors available to accommodate a reasonably-sized BTB, and because indirect calls can occur frequently in procedural programs as well[1] (see Section 7). Some processors (most notably, Intel's Pentium processor and its successor P6) have small BTBs today. Interestingly, BTBs behave similar to inline caches—they work well for monomorphic call sites but badly for highly polymorphic call sites. For example, the performance of VTBL on such a processor would be similar to the VTBL+IC scheme discussed above. The impact of BTBs on dispatch performance can be estimated by reducing the value of branch penalty B in the formulas of Tables 10 and 11, but the extent of the reduction depends on the BTB miss ratio (i.e., inline cache miss ratio) of the application. We also explore this effect in Section 6.

5.6 Summary

We have evaluated the dispatch cost of a range of dispatch mechanisms, taking into account the performance characteristics of modern pipelined superscalar microprocessors. On such processors, objectively evaluating performance is difficult since the cost of each instruction depends on surrounding instructions and the cost of branches depends on dynamic branch prediction. In particular, some instructions may be "free" because they can be executed in parallel with other instructions, and unpredictable conditional branches as well as indirect branches are expensive (and likely to become more expensive in the future). On superscalar architectures, counting instructions to estimate performance is highly misleading. We have studied dispatch performance on three processor models designed to represent the past (1992), present (1995), and future (1997) state of the art in processor implementation.

We have analyzed the run-time performance of dispatch mechanisms as a function of processor characteristics such as branch latency and superscalar instruction issue, and as a function of system parameters such as the average degree of polymorphism in application code. The resulting formulas allow dispatch performance to be computed for a wide range of possible (future) processors and systems. In addition, we also present formulas for computing the space cost of the various dispatch techniques. Our study has produced several results:

- The relative performance of dispatch mechanisms varies with processor implementation. Whereas some mechanisms become relatively more expensive (in terms of cycles per dispatch) on more aggressively superscalar processors, others become less expensive. No single dispatch mechanism performs best on all three processor models.
- Mechanisms employing indirect branches (i.e., all table-based techniques) may not perform well on current and future hardware since indirect branches incur multi-cycle pipeline stalls, unless some form of indirect branch prediction is present. Inline caching variants

[1] Procedure returns occur frequently even in non-polymorphic programs, but these can be handled more efficiently by a return address prediction buffer [63].

pipeline very well and do not incur such stalls. On deeply pipelined superscalar processors like the P97, inline caching techniques may substantially outperform even the most efficient table-based techniques.

- Hybrid techniques combining inline caching with a table-based method may offer both excellent average dispatch cost as well as a low worst-case dispatch cost.

- On superscalar processors, the additional cost of supporting dynamic typing is small (often zero) because the few additional instructions usually fit into otherwise unused instruction issue slots.

Even though selecting the best dispatch mechanism for a particular system is still difficult since it involves many factors, the data presented here should allow dispatch cost bounds to be estimated for a wide range of systems. Therefore, we hope that this study will be helpful to system implementors who need to choose the dispatch mechanism best suited to their needs.

6 Measurement of virtual function call overhead on modern processors

"This virtual call mechanism can be made essentially as efficient as the 'normal function call' mechanism."

Bjarne Stroustrup [119]

In this section we measure the direct cost of virtual function table lookup for a number of realistic C++ programs running on superscalar processors employing co-scheduling and simple indirect branch prediction, and identify the processor characteristics that most affect this cost. In Section 5.2 we saw that, when analyzed in isolation, the cost of dispatch sequences of table-based techniques are similar to virtual function tables. Therefore VTBL serves as a representative technique for table-based dispatch in this quantitative analysis.

6.1 Virtual function tables and the thunk variant

Figure 29 shows the five-instruction code sequence that a C++ compiler typically generates for

```
1: load [object_reg + #VFToffset], table_reg
2: load [table_reg + #deltaOffset], delta_reg
3: load [table_reg + #selectorOffset], method_reg
4: add  object_reg, delta_reg, object_reg
5: call method_reg
```

Figure 29. Instruction sequence for VFT dispatch

a virtual function call. The first instruction loads the receiver object's VFT pointer into a register, and the subsequent two instructions index into the VFT to load the target address and the receiver pointer adjustment (delta) for multiple inheritance. The fourth instruction adjusts the receiver address to allow accurate instance variable access in multiple inherited classes. Finally, the fifth instruction invokes the target function with an indirect function call.

Instructions 2 and 4 (in *italics*) in Figure 29 are only necessary when the class of the receiver has been constructed using multiple inheritance. Otherwise, the offset value loaded into the register *delta_reg* in instruction 2 is zero, and the add in instruction 4 has no effect. It would be convenient if we could avoid executing these useless operations, knowing that the receiver's class employs only single inheritance. Unfortunately, at compile time, the exact class of the receiver is unknown. However, the receiver's virtual function table, which stores the offset values, "knows" the exact class at run time. The trick is to perform the receiver address adjustment only after the virtual function table entry is loaded. In the GNU GCC thunk implementation, the virtual function table entry contains the address of a parameterless procedure (a thunk), that adjusts the receiver address and then calls the correct target function (see Figure 30). In the

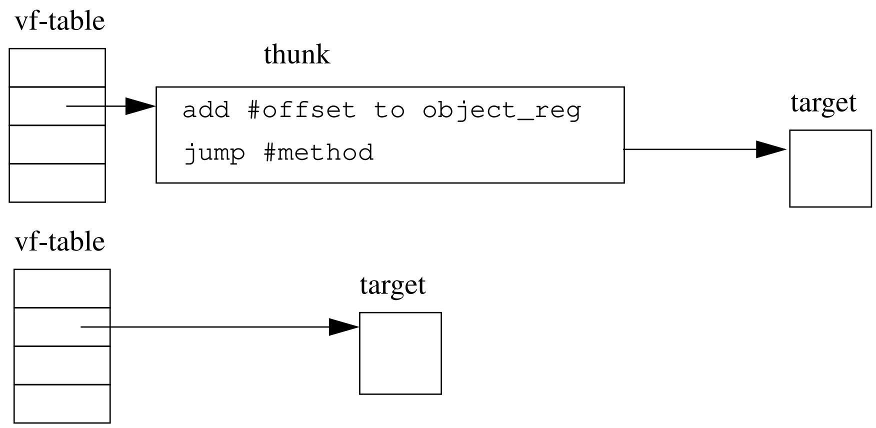


Figure 30. Thunk virtual function tables Multiple inheritance case (above)
Single inheritance case (below)

single inheritance case, the virtual function table entry points directly to the target function. Instead of always loading the offset value and adding it to the *this* pointer (the address of the receiver object), the operation only happens when the offset is known to be non-zero. Since multiple inheritance occurs much less frequently than single inheritance, this strategy will save two instructions for most virtual function calls[1]. Therefore, barring instruction scheduling effects, thunks should be at least as efficient as standard virtual function tables.

6.2 Superscalar processors

How expensive is the virtual function call instruction sequence? A few years ago, the answer would have been simple: most instructions execute in one cycle (ignoring cache misses for the moment), and so the standard sequence would take 5 cycles. However, on current hardware the situation is quite different because processors try to exploit instruction-level parallelism with *superscalar execution*. Figure 31 shows a simplified view of a superscalar CPU. Instructions are fetched from the cache and placed in an instruction buffer. During every cycle, the issue unit selects one or more instructions and dispatches them to the appropriate functional unit (e.g., the integer unit).

The processor may contain multiple functional units of the same type. For example, the processor in Figure 31 has three integer units and thus can execute up to three integer instructions concurrently. The number of instructions that can be dispatched in one cycle is called the issue width. If the processor in Figure 31 had an issue width of four (often called "four-way superscalar"), it could issue, for example, two integer instructions, one load, and a floating-point instruction in the same cycle.

[1] In the GNU GCC implementation for SPARC executables, one of the offset instructions is usually replaced by a register move. The latter is necessary to pass the *this* pointer in register %o0 to the callee.

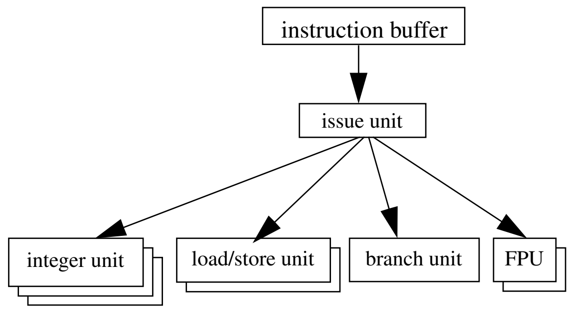


Figure 31. Simplified organization of a superscalar CPU

Of course, there is a catch: two instructions can only execute concurrently if they are independent. There are two kinds of dependencies: data dependencies and control dependencies. *Data dependencies* arise when the operands of an instruction are the results of previous instructions; in this case, the instruction cannot begin to execute before all of its inputs become available. For example, instructions 2 and 3 of the VFT dispatch sequence can execute concurrently since they are independent, but neither of them can execute concurrently with instruction 1 since they both use the VFT pointer loaded in instruction 1.

The second form of dependencies, *control dependencies*, result from the fact that some instructions influence the flow of control. For example, the instructions following a conditional branch are not known until the branch executes and determines the next instruction to execute (i.e., whether the branch is taken or not). Therefore, even if an instruction after the branch has no data dependencies, it cannot be executed concurrently with (or before) the branch itself.

Both forms of dependencies may carry an execution time penalty because of pipelining. Whereas the result of arithmetic instructions usually is available in the next cycle (for a latency of one cycle), the result of a load issued in cycle *i* is not available until cycle *i+2* or *i+3* (for a *load latency L* of 2 or 3 cycles) on most current processors even in the case of a first-level cache hit. Thus, instructions depending on the loaded value cannot begin execution until L cycles after the load. Similarly, processors impose a *branch penalty* of B cycles after conditional or indirect branches: when a branch executes in cycle *i* (so that the branch target address becomes known), it takes B cycles to refill the processor pipeline until the first instruction after the branch reaches the execute stage of the pipeline and produces a result.

To summarize, on ideal hardware (with infinite caches and an infinite issue width), the data and control dependencies between instructions impose a lower limit on execution time[1]. If N instructions were all independent, they could execute in a single cycle, but if each of them depended on the previous one they would take at least N cycles to execute. Thus, the number of

[1] A new technique, *value prediction* [93], which is currently still in a research stage, can remove this limit. A correct prediction allows the instruction that uses a value to execute before the instruction that produces the value, thus bypassing data-dependencies.

instructions becomes only an *approximate* predictor of execution time on superscalar processors. Even though actual processors do not have infinite resources, this effect still is significant as we shall see later in this paper.

In a previous study Section 5.2 we approximated the dispatch cost of several techniques by analyzing the call sequence and describing their cost as a function of load latency and branch penalty, taking into account superscalar instruction issue. However, this approximation (2L + B + 1 for VFT dispatch) is only an upper bound on the true cost, and the actual cost might be lower. The next few sections explain why.

6.2.1 BTB branch prediction

Since branches are very frequent (typically, every fifth or sixth instruction is a branch [65]) and branch penalties can be quite high (ranging up to 15 cycles on the Intel Pentium Pro processor [101]), superscalar processors try to reduce the average cost of a branch with branch prediction. Branch prediction hardware guesses the outcome of a branch based on previous executions and immediately starts fetching instructions from the predicted path. If the prediction is correct, the next instruction can execute immediately, reducing the branch latency to one cycle; if predicted incorrectly, the processor incurs the full branch penalty B. Predictions are based on previous outcomes of branches. Typically, the branch's address is used as an index into a prediction table. For conditional branches, the result is a single bit indicating whether the branch is predicted taken or not taken, and typical prediction hit ratios exceed 90% [65].

For indirect branches, the prediction mechanism must provide a full target address, not just a taken/not taken bit. A *branch target buffer* (BTB) accomplishes this by storing the predicted address in a cache indexed by the branch address (very similar to a data cache). When the processor fetches an indirect branch, it accesses the BTB using the branch instruction's address. If the branch is found, the BTB returns its last target address and the CPU starts fetching instructions from that address before the branch is even executed. If the prediction is wrong, or if the branch wasn't found, the processor stalls for B cycles and updates the BTB by storing the branch and its new target address.

BTBs affect the cost of the VFT dispatch sequence: if the virtual call was executed previously, is still cached in the BTB, and invokes the same function as in the previous execution, the branch penalty is avoided, reducing the sequence's cost to 2L + 1. We explore more sophisticated and accurate branch prediction schemes in Section 7 and beyond.

6.2.2 Advanced superscalar execution

To improve performance, modern processors employ two additional techniques that can decrease the performance impact of dependencies.

First, instructions may be executed *out of order*: an instruction *I* that is waiting for its inputs to become available does not stall all instructions after it. Instead, those instructions may execute *before I* if their inputs are available. Additional hardware ensures that the program semantics are preserved; for example, if instructions I_1 and I_2 write the same register, I_1 will not overwrite the result of I_2 even if I_2 executes first. Out-of-order execution increases throughput by allowing other instructions to proceed while some instructions are stalled.

Second, *speculative execution* takes this idea one step further by allowing out-of-order execution across conditional or indirect branches. That is, the processor may speculatively execute instructions before it is known whether they actually should be executed. If speculation fails because a branch is mispredicted, the effects of the speculatively executed instructions have to be undone, again requiring extra hardware. Because branches are so frequent, speculating across them can significantly improve performance if branches can be predicted accurately.

Of course, the processor cannot look arbitrarily far ahead in the instruction stream to find instructions that are ready to execute. For one, the probability of fetching from the correct execution path decreases exponentially with each predicted branch. Also, the issue units must select the next group of instructions to be issued from the buffer within one cycle, thus limiting the size of that buffer. The most aggressive designs available today select their instructions from a buffer of about 30-40 instructions [100][101], so that instructions have to be reasonably "near" the current execution point in order to be issued out-of-order.

6.2.3 Co-scheduling of application code

With speculative, out-of-order execution the cost of the VFT dispatch sequence is not only highly variable (depending on the success of branch prediction), but it cannot be computed in isolation from its surrounding code. For example, if many other instructions precede the dispatch sequence, they could execute during the cycles where the processor would otherwise lay idle waiting for the loads to complete. Or vice versa, the dispatch instructions could fit into empty issue slots of the rest of the basic block. This co-scheduling of the application and dispatch code may reduce the overall cost significantly, possibly to the point where completely removing the dispatch code would not speed up the program at all (since all dispatch instructions fit into otherwise empty issue slots). Thus, at least in theory, a dispatch implementation may reach zero overhead (i.e., add no cycles to the execution time) even though it does introduce extra instructions.

6.2.4 Summary

While all of the processor features discussed above improve performance on average, they also increase the variability of an instruction's cost since it depends not only on the instruction itself (or the instruction and its inputs), but also on the surrounding code. Most processors sold today (e.g., the Intel Pentium and Pentium Pro processors, as well as virtually all RISC processors introduced since 1995) incorporate several or all of these features. As a result, it is hard to predict how expensive the average C++ virtual function call is on a current-generation PC or workstation. The experiments described in the rest of this paper aim to shed some light on this question.

6.3 Method

It is hard to measure virtual function call cost directly since we cannot usually run program P_{ideal} (the program without any dispatch code). Although it is fairly easy to count the number of instructions executed on behalf of dynamic dispatch, this measure does not accurately reflect the cost in processor cycles. On modern pipelined processors with multiple instruction issue the cost of an instruction may vary greatly. For example, on a 4-way superscalar processor with a branch penalty of 6, an instruction can take anywhere between 0.25 and 7 cycles[1].

Therefore we measure the direct cost of virtual function table lookup by *simulating* the execution of P and P_{ideal}. Using an executable editor and a superscalar processor simulator, we compute the execution times of both programs, thus arriving at the direct cost of dispatch. In addition to allowing dispatch cost to be measured at all, simulation also facilitates exploring a broad range of possible processor implementations, thus making it possible to anticipate performance trends on future processors.

6.3.1 Simulation scheme

Figure 32 shows an overview of our experimental approach: first, the C++ program compiled by an optimizing compiler (we used GNU gcc 2.6.3 and 2.7.2 with options -O2 -msupersparc). Then, an application that uses the EEL executable editing library [88] detects the dispatch instructions and produces a file with their addresses. Using this file as well as a processor description, the superscalar processor simulator then runs the benchmark.

The simulator executes SPARC programs using the *shade* tracing tool [29]. Shade always executes all instructions of the program so that programs produce the same results as if they were executed on the native machine. Each instruction executed is then passed to a superscalar processor simulator that keeps track of the time that would be consumed by this instruction on the simulated processor. Optionally, the simulation of dispatch instructions can be suppressed (i.e., they are executed but not passed to the timing simulator), thus simulating the execution of P_{ideal} , the program using the perfect, zero-cost dynamic dispatch scheme.

The cycle-level simulator schedules instructions from a 32-entry instruction window. As soon as an instruction is eligible for execution (the instructions it depends on have been executed and an appropriate functional unit is available), its latency is calculated, taking into account caching an branch prediction effects. For this purpose we adapted the PowerSim simulator (for the PowerPC ISA) used by Adam Talcott [120], to the SPARC architecture.

---

[1] In the absence of cache misses.

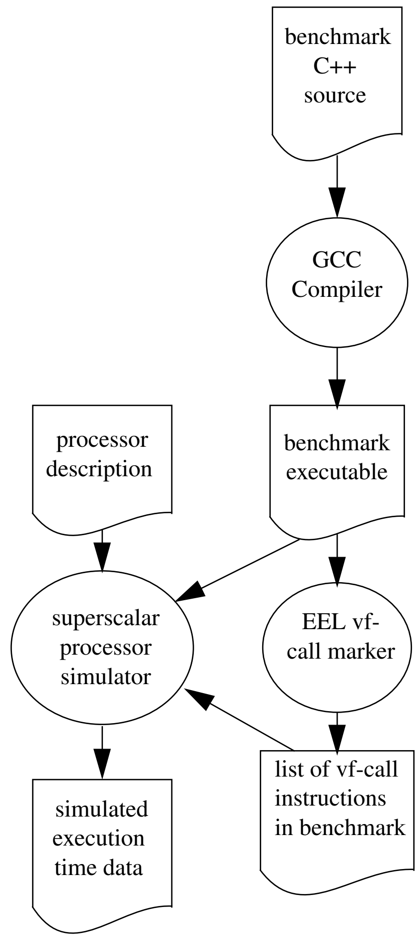


Figure 32. Overview of experimental setup

Although we currently use only benchmarks for which we have the source, this is not strictly necessary. Provided that the vf-call marker program detects all virtual calls correctly, any executable can be measured. The source language does not even have to be C++, as long as the language under consideration uses VFT dispatch for its messages. Compared to a tool that detects dispatches at the source code level, a tool based on binary inspection may be harder to construct, but it offers a significant advantage even beyond its source, compiler, and language independence. In particular, it is non-intrusive, i.e., does not alter the instruction sequence, and is thus more accurate.

The vf-call marker program detects the virtual function call code sequence discussed in section 3.3.2. This code sequence consists of the five instructions in Figure 29 and any intervening register moves. They may appear in different orderings (but with the correct dependencies), possibly spread out over different basic blocks. Since the code sequence is highly characteristic, the marker program is very accurate, detecting virtual calls exactly for most programs.[1] For three benchmarks the marker is slightly imprecise, erring by 0.4% or less. Only in *ixx*, 2.3% of the calls went undetected so that our measurements slightly underestimate the direct dispatch cost for this benchmark.

6.3.2 Benchmarks

We tested a suite of two small and six large C++ applications totalling over 90,000 lines of code (Table 13). In general, we tried to obtain large, realistic applications rather than small, artificial benchmarks. Two of the benchmarks (*deltablue* and *richards*) are much smaller than the others; they are included for comparison with earlier studies (e.g., [72][62]). *Richards* is the only synthetic benchmark in our suite (i.e., the program was never used to solve any real problem). We did not yet test any programs for which only the executables were available.

name	description	lines
deltablue	incremental dataflow constraint solver	1,000
eqn	type-setting program for mathematical equations	8,300
idl	SunSoft's IDL compiler (version 1.3) using the demonstration back end which exercises the front end but produces no translated output.	13,900
ixx	IDL parser generating C++ stubs, distributed as part of the Fresco library (which is part of X11R6). Although it performs a function similar to IDL, the program was developed independently and is structured differently.	11,600
lcom	optimizing compiler for a hardware description language developed at the University of Guelph	14,100
porky	back-end optimizer that is part of the Stanford SUIF compiler system	22,900
richards	simple operating system simulator	500
troff	GNU groff version 1.09, a batch-style text formatting program	19,200

Table 13. Benchmark programs

For every program except *porky*[2] we also tested an "all-virtual" version (indicated by "-av" suffix) which was compiled from a source in which all member functions except operators and destructors were declared virtual. We chose to include these program versions in order to simulate programming styles that extensively use abstract base classes defining virtual functions only (C++'s way of defining interfaces). For example, the Taligent CommonPoint frameworks provide all functionality through virtual functions, and thus programs using CommonPoint (or similar frameworks) are likely to exhibit much higher virtual function call frequencies. Lacking real, large, freely available examples of this programming style, we created the "all virtual" programs to provide some indication of the virtual function call overhead of such programs.

[1] We cross-checked this by using VPROF, a source-level virtual function profiler for GCC [5].

[2] Porky cannot be compiled as "all virtual" without a large effort of manual function renaming.

program	version	instructions	virtual calls	instructions per virtual call
deltablue	original	40,427,339	615,100	65
	all-virtual	79,082,867	5,145,581	15
eqn	original	97,852,301	100,207	976
	all-virtual	108,213,587	1,267,344	85
idl	original	91,707,462	1,755,156	52
	all-virtual	99,531,814	3,925,959	25
ixx	original	30,018,790	101,025	297
	all-virtual	34,000,249	606,463	56
lcom	original	169,749,862	1,098,596	154
	all-virtual	175,260,461	2,311,705	75
richards	original	8,119,196	65,790	123
	all-virtual	15,506,753	1,146,217	13
troff	original	91,877,525	809,312	113
	all-virtual	114,607,159	3,323,572	34
porky	original	748,914,861	3,806,797	196

Table 14. Basic characteristics of benchmark programs (dynamic counts)

These versions can also be used to approximate the behavior of programs written in languages where (almost) every function is virtual, e.g., Java or Modula-3.

For each benchmark, Table 14 shows the number of executed instructions, the number of virtual function calls, and the average number of instructions between calls. All numbers are dynamic, i.e., reflect run-time execution counts unless otherwise mentioned. All programs were simulated in their entire length as shown in Table 14. Simulation consumed a total of about one CPU-year of SPARCstation-20 time.

6.3.3 Processors

Table 16 shows an overview of recently introduced processors. Since we could not possibly simulate all of these processors and their subtle differences, we chose to model a hypothetical SPARC-based processor that we dubbed P96 because it is meant to resemble the average processor introduced today.

Processor	Ultra SPARC	MIPS R10K	DEC Alpha 21164	Power PC 604	Intel Pentium Pro
Shipping date	95	95	95	95	95
Size of BTB	0	0	0	64	512
Size of BHT[a]	2048	512	2048	512	8192[b]
Branch Penalty	4	4	5	1-3	11-15
Issue Width	4	5	4	4	3
Load Latency	2	2	2	2	3
Primary I-cache[c]	16Kx2	32Kx2	8Kx1	32Kx4	8K
Primary D-cache	16Kx1	32Kx2	8Kx1	32Kx4	8K
Out-of-order?	Y	Y	Y	Y	Y
Speculative?	Y	Y	Y	Y	Y

Table 15. Characteristics of recently introduced processors

[a] BTB = branch target buffer size; BHT = branch history table size (branch histories are used to predict the direction of conditional branches)
[b] In the Pentium, BTB and BHT are joined. Each entry in the table stores the branch target and 16 2-bit counters, one for each possible 4-bit local history
[c] 16Kx2 means the cache is 16K bytes, and 2-way associative

For our experiments, we ran all benchmarks on P96 to obtain the base results for the dispatch overhead. To examine the effects of the most important processor features, we then varied each parameter while keeping all others constant. Finally, we also measured a few individual configurations that resemble existing processors (Table 16). P96-noBTB resembles the UltraSPARC in that it lacks a BTB, i.e., does not predict indirect branches. P96-Pro resembles the Pentium Pro in its branch configuration, in that it has a high branch penalty. Finally, P2000 is an idealized processor with essentially infinite hardware resources; we use it to illustrate the impact of the branch penalty on a processor that has virtually no other limitations on instruction issue.

Processor	P96	P96-noBTB	P96-Pro	P2000/bp1	P2000/ bp10
Size of BTB	256	0	512	1024	1024
Size of BHT	1024	1024	0	1024	1024
Branch Penalty	4	4	15	1	10
Issue Width	4	4	4	32	32
Load Latency	2				
Primary I-cache	32K, 2-way associative				
Primary D-cache	32K, 2-way associative				
Out-of-order?	Y				
Speculative?	Y				

Table 16. Characteristics of simulated processors

It should be noted that none of these processors is intended to exactly model an existing processor; for example, the Intel Pentium Pro's instruction set and microarchitecture is very different from P96-Pro, and so the latter should not be used to predict the Pentium Pro's performance on C++ programs. Instead, we use these processors to mark plausible points in the design space, and their distance and relationship to illustrate particular effects or trends.

6.4 Results

This section first examine the cost of dynamic dispatch on the baseline architecture, P96, and then examines the impact of individual architectural parameters (branch penalty/prediction, load latency, and issue width).

6.4.1 Instructions and cycles

First, we will examine the cost of dynamic dispatch on the baseline architecture, P96. Recall that we define the cost as the additional cycles spent relative to a "perfect" dispatch implementation that implements each dispatch with a direct call. Figure 33 and Figure 34 show the

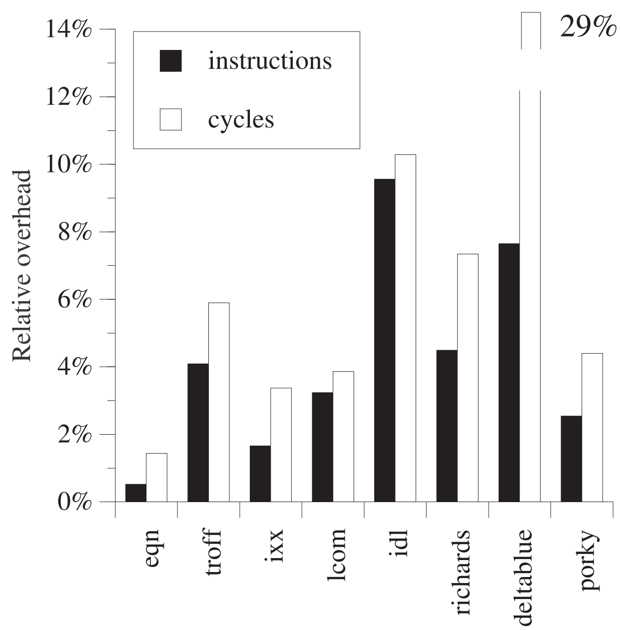


Figure 33. Direct cost of standard VFT dispatch (unmodified benchmarks)

results. On the standard benchmarks, the cost varies from 1.4% for *eqn* to 29% for *deltablue*, with a median overhead of 5.2%. For the all-virtual versions, the overhead increases to between 4.7% and 47% with a median overhead of 13%. The standard benchmarks spend a median 3.7% of their instructions on dispatch, and the all-virtual versions a median of 13.7%. For the standard benchmarks the cycle cost is larger than the cost in the number of instructions executed; on average, it is a median 1.7 times larger. This difference confirms that the VFT dispatch sequence does not schedule well on a superscalar processor, compared to non-dispatch code. However, this effect varies substantially between benchmarks. The largest difference is found in *eqn* (2.8 times) and deltablue (3.8 times). Since the dispatch sequence is always the same, this indicates that the instructions surrounding a call can significantly affect the cost of virtual function lookup, or that virtual calls are more predictable in some programs than in others. We will explore these questions shortly.

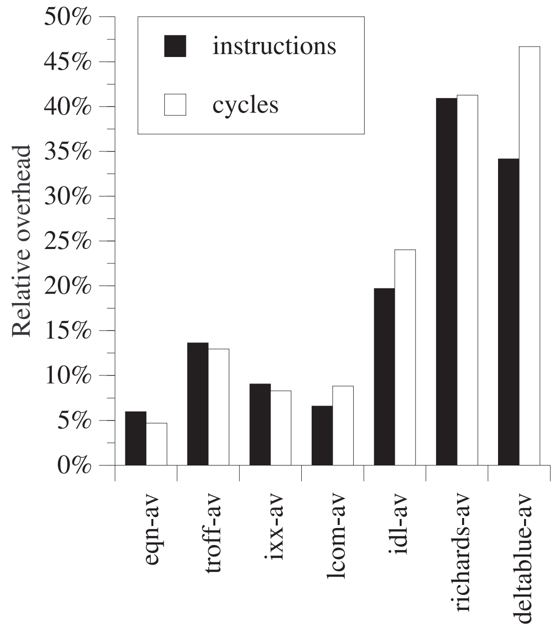


Figure 34. Direct cost of standard VFT dispatch (all-virtual benchmarks)

6.4.2 Thunks

Figure 35 compares the cycle cost of standard and thunk implementations for the unmodified benchmarks[1]. Thunks have a smaller cycle overhead than regular tables for all benchmarks, using a median of 79% of the cycles of the regular implementation. Figure 36 shows the cycle cost for the all-virtual benchmarks. Here, thunks have 72% of the regular overhead. The exact amount of the gain varies greatly between benchmarks. For example, the thunk overhead for *ixx* and *deltablue* is only 15% and 47% of the regular overhead, while for *troff*, thunks use almost as many cycles as standard tables (98%).

How can thunks, in some cases, improve dispatch performance by more than a factor of two? One reason for the difference is the unnecessary receiver address adjustment that is avoided with thunks (instructions 2 and 4 in Figure 29). In the thunk implementation, instructions that

[1] Since GCC cannot compile *idl*, *idl-av*, and *lcom-av* with thunks, these benchmarks are missing from Figure 35 and Figure 36.

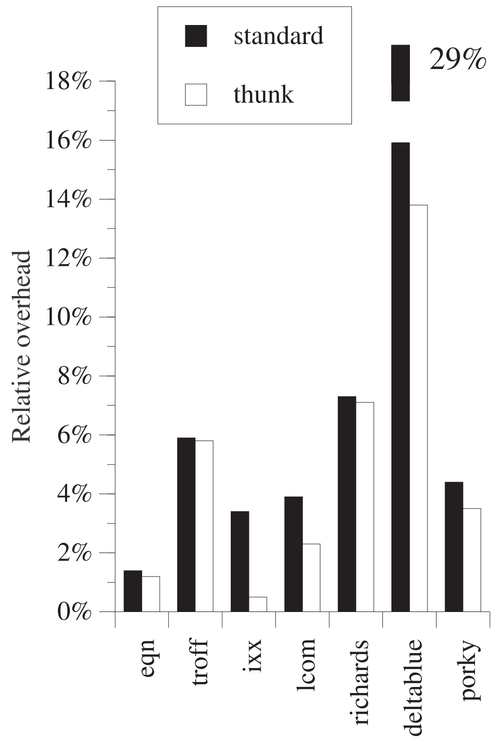


Figure 35. Cycle cost of standard and thunk variants (unmodified benchmarks)

depend on the receiver's address do not have to wait for the virtual function call to complete, if the target is predicted accurately. In contrast, in the standard implementation instructions 2 and 4 create a dependency chain from instruction 1 to any instruction that needs the receiver's address. In *deltablue*, the added dependencies stretch the inner loop from 9 cycles (thunks) to 12 cycles (standard), where a direct called implementation would use 8 cycles (all times exclude cache misses). Thus the overhead of thunks is only 25% of the overhead of standard tables for a large part of the execution, so that the removal of only two instructions out of five can avoid more than half the virtual function call overhead in particular cases. This effect is particularly pronounced in all-virtual benchmarks that contain many calls to accessor functions (i.e., functions that just return an instance variable).

Another part of the difference is due to memory hierarchy effects: with perfect caching[1], thunk overhead for *ixx* and *deltablue* rises to 48% and 54%.

6.4.3 Generalization to other processors

How specific are these measurements to our (hypothetical) P96 processor? Figure 37 compares the relative dispatch overhead of standard tables on P96 with that of the other processors listed

[1] By perfect caching we mean that there are no cache miss, not even for cold starts.

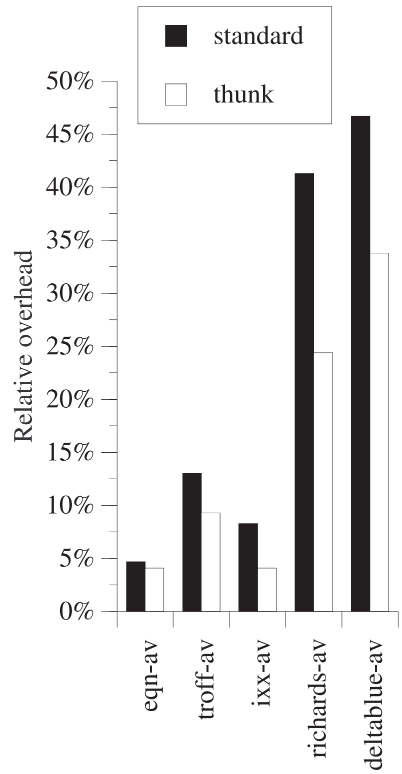


Figure 36. Cycle cost of standard and thunk variants (all-virtual benchmarks)

in Table 16. Clearly, the processor configuration affects performance: longer branch penalties combined with less ambitious branch prediction (P96-Pro) and the absence of a BTB (P96-noBTB) both impact dispatch performance negatively so that all programs spend a larger percentage of their time in dispatch code. Even P2000 with its 32-instruction issue CPU shows relative overheads that are a median 28% higher than in P96, due to its higher branch penalty (10 instead of 4 cycles). Thus, we expect future processors to exhibit higher dispatch overheads for most C++ programs

To explain these differences in more detail, the next few sections present the effects of several processor characteristics on the direct cost of dynamic dispatch. In particular, we will investigate the impact of the branch penalty, the size of the branch target buffer (BTB), and the issue width. In each experiment, we vary the feature under investigation while keeping all other characteristics constant. To illustrate the trends, we show cost in two ways, each of them relative to P96. The first graph in each section compares absolute cost, i.e., the number of dispatch cycles relative to P96. The second graph compares relative cost, i.e., the percentage of total execution time (again relative to P96) spent in dispatch. The two measurements are *not* absolutely correlated: if the absolute overhead increases, the relative cost may decrease if the rest of the application is slowed down even more than the dispatch code. Similarly, the absolute cost may decrease while the relative cost increases because the absolute cost of the rest of the application decreases even more. .

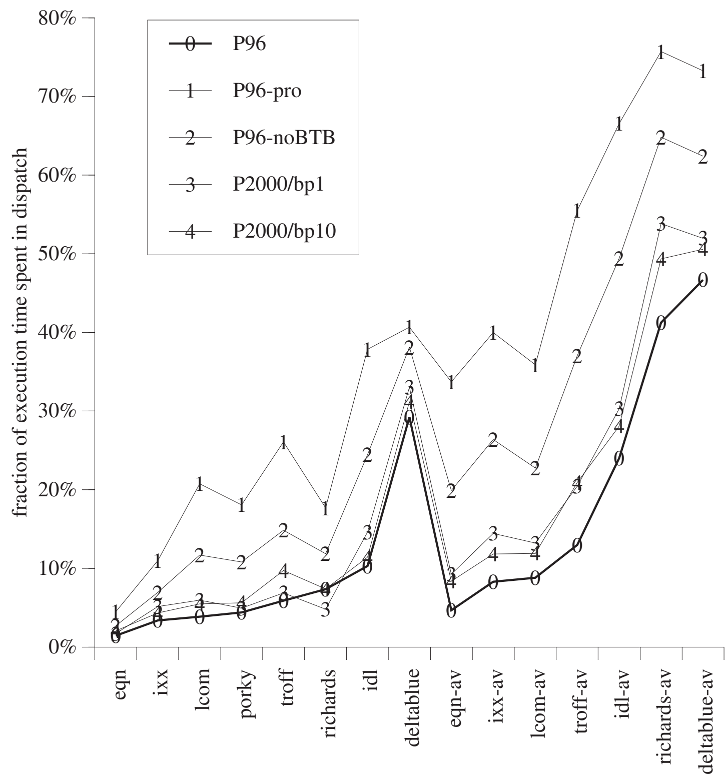


Figure 37. Dispatch overhead in P96 vs. P96-noBTB and P96-Pro

6.4.4 Influence of branch penalty

Since one of the five instructions in the dispatch sequence is an indirect branch, the branch misprediction penalty directly affects the cost of virtual function dispatch. Since each dispatch contains a single indirect branch, we would expect the absolute overhead to increase proportionally to the number of mispredicted branches. And since the number of mispredictions is independent of the branch penalty, the cost should increase linearly with the branch penalty.

Figure 38 confirms this expectation. For small branch penalties, the actual penalty can be smaller than expected if the branch penalty is filled with instructions preceding the branch which have not yet completed (e.g. because they are waiting for their inputs to become available). This effect appears to be small.

The slope of the overhead lines increases with the BTB miss ratio, i.e., the fraction of mispredicted calls. *Richards* and *troff* have large BTB miss ratios (54% and 30%), which account for

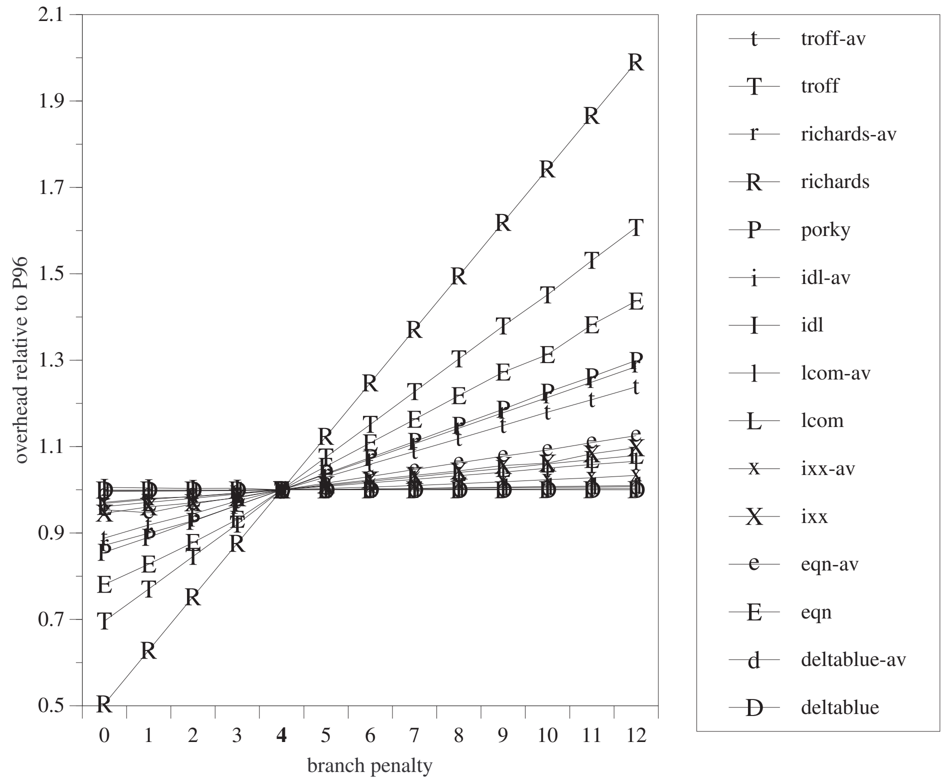


Figure 38. Overhead in cycles (relative to P96) for varying branch penalties

their steep cost curves. Most of the other benchmarks have a misprediction rate of 10% or less, which dampens the effect of branch penalty on cycle cost.

Figure 39 shows that the *fraction* of execution time spent in dispatch can actually decrease with increasing branch penalty. For example, *ixx* has many indirect calls that are not part of virtual function calls, and these branches are very unpredictable (with a BTB miss ratio of 86%). Consequently, the relative overhead of virtual calls in *ixx* decreases with larger branch penalties since the cost of the rest of the program increases much faster.

However, for most benchmarks the relative overhead differs less than 20% between the extreme branch penalty values (0 and 10), indicating that the VFT branches are about as predictable as the other branches in the applications. Thus, the relative dispatch costs given earlier in Figure 33 and Figure 34 are quite insensitive to branch penalty variations.

6.4.5 Influence of branch prediction

As discussed in section 6.2.1, branch target buffers (BTBs) predict indirect (or conditional) branches by storing the target address of the branch's previous execution. How effective is this branch prediction? Our baseline processor, P96, has separate prediction mechanisms for condi-

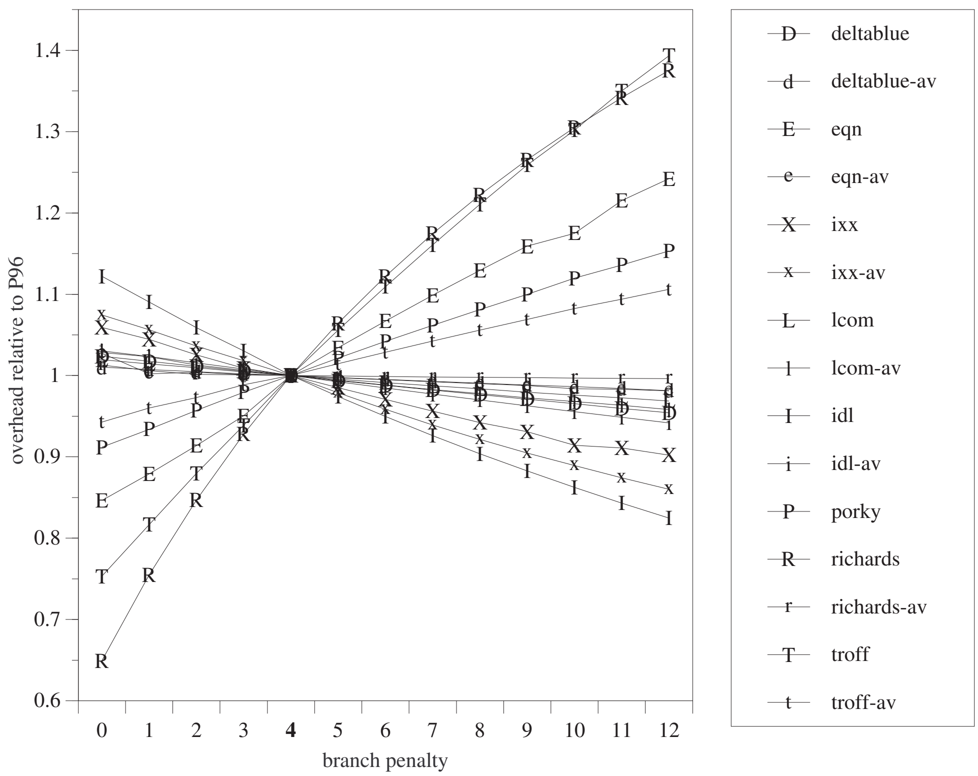


Figure 39. Overhead in % of execution time (relative to P96) for varying branch penalties

tional and indirect branches since the former can better be predicted with local history-based two-level predictors [135]. The BTB is used exclusively to predict indirect branches. Thus, varying the size of the BTB will affect only indirect branches, thus directly illustrating the BTB's effect on dispatch overhead.

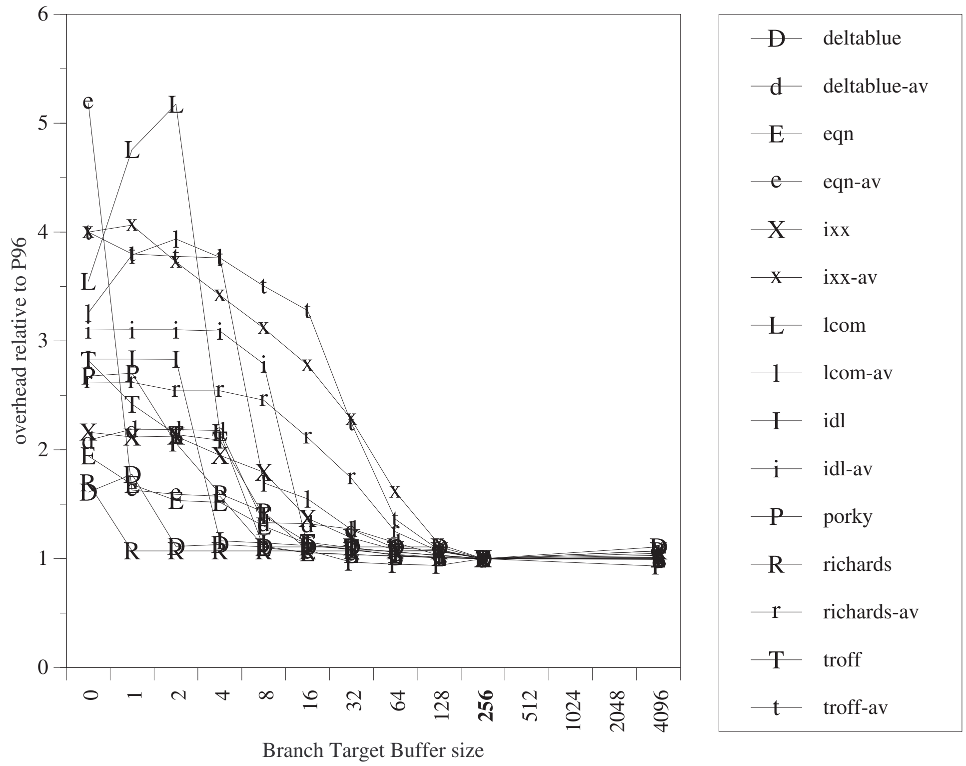


Figure 40. Overhead in cycles (relative to P96) for varying Branch Target Buffer sizes

In general, smaller BTBs have lower prediction ratios because they cannot store as many individual branches. Recall that the processor uses the branch instruction's address to access the BTB (just like a load instruction uses the data address to access the data cache). If the branch isn't cached in the BTB, it cannot be predicted. Naturally, the smaller the BTB, the fewer branches it can hold, and thus the larger the fraction of branches that can't be predicted because they aren't currently cached in the BTB. Figure 40 confirms this expectation: in general, smaller BTBs increase dispatch overhead. Apparently, a BTB size of 128 entries is large enough to effectively cache all important branches, as the dispatch overhead does not decrease significantly beyond that BTB size.

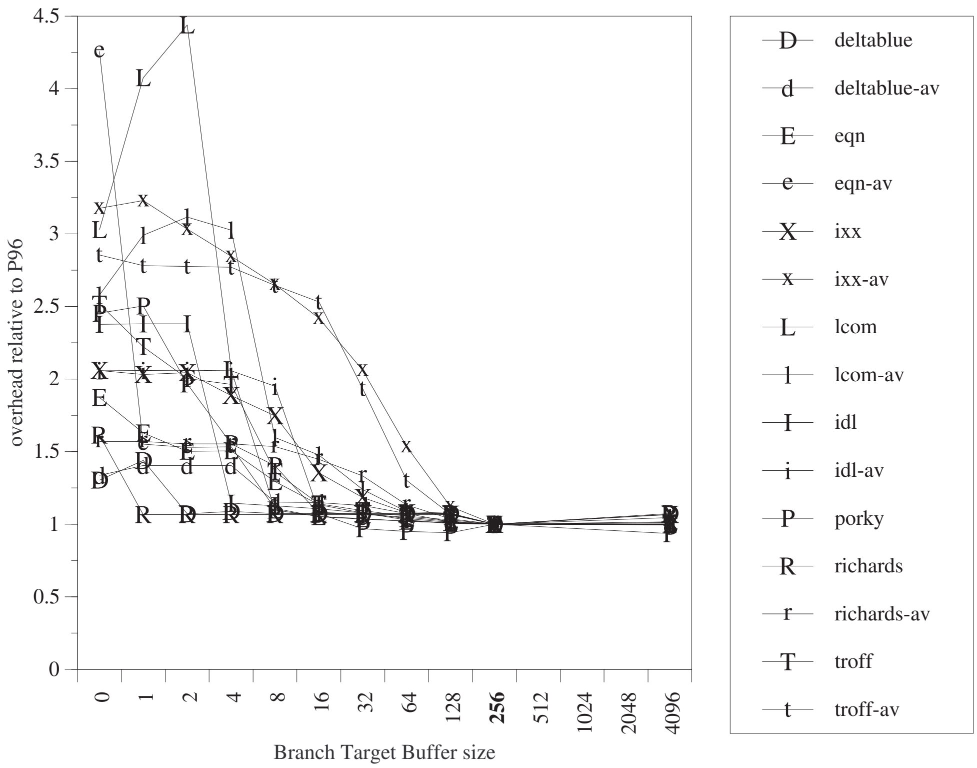


Figure 41. Overhead in% of execution time (relative to P96) for varying BTB sizes

Figure 41 shows the dispatch overhead as a fraction of execution time. In general, the relative overhead varies in tandem with the absolute overhead, i.e., smaller BTBs increase dispatch overhead. For processors with BTBs with 128 or more entries, P96 should accurately predict the BTB's impact on dispatch performance.

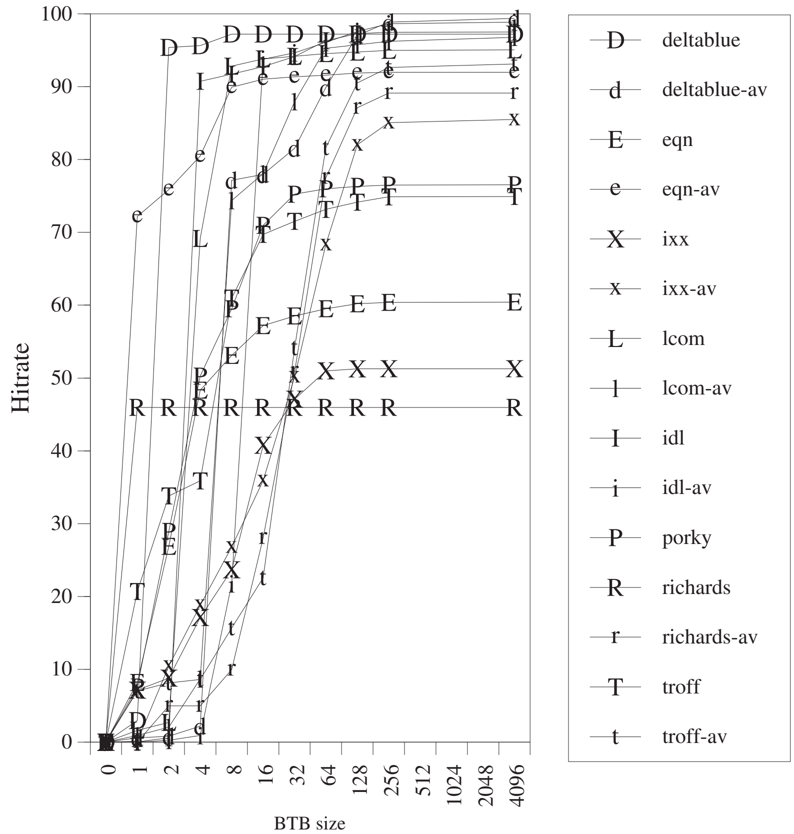


Figure 42. Indirect branch prediction ratio as a function of BTB size

Finally, Figure 42 shows the prediction ratio as a function of the BTB size. The ratio starts at zero (without a BTB, indirect branches cannot be predicted) and asymptotically reaches a final value around a BTB size of 128. Generally, smaller benchmarks need fewer BTB entries to reach asymptotic behavior since they have fewer active call sites.

The asymptotic prediction ratio corresponds to the hit ratio of an inline cache[1] [35]. For some benchmarks, prediction works very well, with 90% or more of the calls predicted correctly. But several benchmarks (especially *richards*, *ixx*, *eqn*, and *troff*) show much lower prediction ratios even with very large BTBs because their calls change targets too frequently. For example, the single virtual call in *richards* frequently switches between four different receiver classes, each of which redefines the virtual function. No matter how large the BTB, such calls cannot be predicted well. The median prediction ratio for the standard benchmarks is only 76% vs. 92%

[1] Since an inline cache stores a target separately for each call site, its hit rate mirrors that of a branch target buffer of infinite size with no history prediction bits.

for the all-virtual versions; the latter are more predictable because many calls only have a single target and thus are predicted 100% correctly after the first call.

Indirect branch prediction substantially reduces the average cost of virtual function calls. In section 6.4.8 we compare average call cost between different prediction regimes.

6.4.6 Influence of load latency

Load latency influences dispatch cost since the VFT dispatch sequence contains two dependent load instructions. Thus, higher load latencies should lead to higher dispatch overhead. Our measurements confirm this assumption: compared to the baseline load latency of two, increasing the load latency to three increases absolute dispatch cost by a median of 51%; the relative cost increases by 31%. Similarly, with a load latency of one the absolute overhead decreases by 44% and the relative overhead by 37%. (Processors are unlikely to have load latencies larger than three, so we did not simulate these.)

Clearly, load latency affects the efficiency of dispatch code more than that of “normal” code sequences. Furthermore, it appears that there are not enough surrounding application instructions to effectively hide the latency of the loads in the dispatch sequence, even for small load latencies.

6.4.7 Influence of issue width

The final factor, issue width (i.e., the number of instructions that can be issued to the functional units in one cycle) can also influence dispatch performance. Figure 43 shows that issue width has a strong impact for small values. On a scalar processor (issuing at most one instruction per cycle), programs spend a much smaller fraction of their time in dispatch. Of course, absolute performance would be worse than on P96 since execution would consume many more cycles (for example, *lcom* is three times slower on the one-issue processor than on the four-issue processor). With larger issue widths the relative overhead increases more slowly, reaching an asymptotic value of 26% (median) more than on P96. Thus, on wider-issue processors, the relative cost of dynamic dispatch will increase slightly because the application code benefits more from the additional issue opportunities than the dispatch code[1]

[1] For a few benchmarks (e.g., *richards*) the relative overhead decreases with high issue widths. We assume that these benchmarks benefit from higher issue rates because they allow critical dispatch instructions to start earlier, thus hiding part of their latency.

.

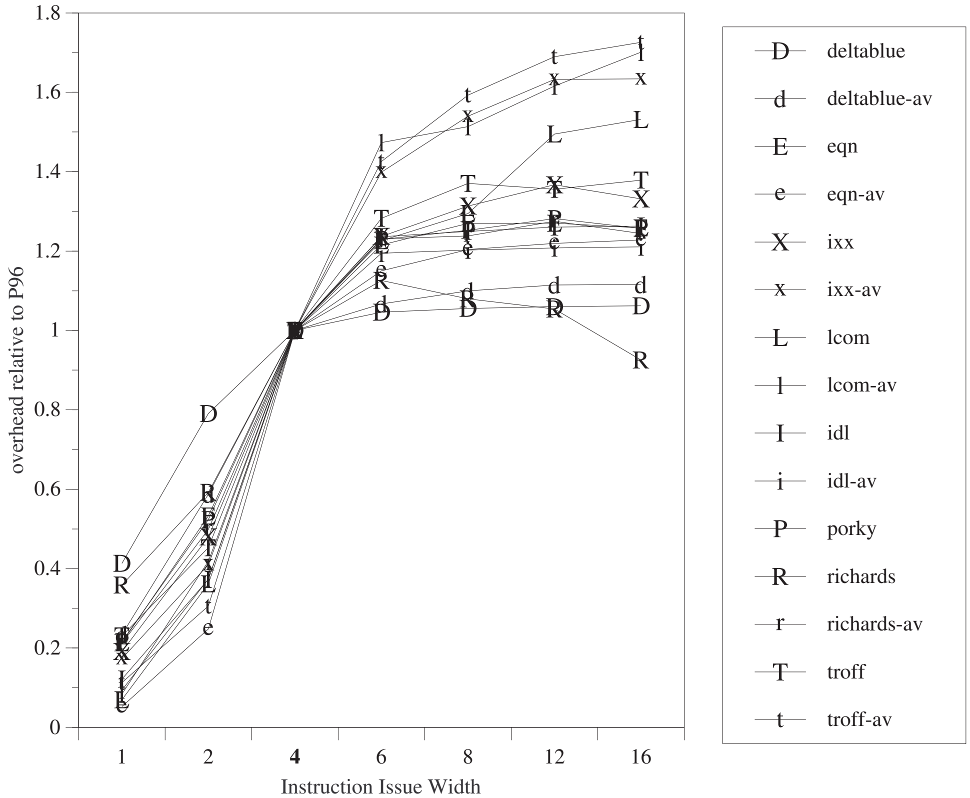


Figure 43. Overhead in % of execution time (relative to P96) for varying issue widths

6.4.8 Cost per dispatch

In Section 5.2 we predicted the cost of a single VFT dispatch to be 2L + B + 1, i.e., two load delays plus a branch penalty; for P96, this adds up to 9 cycles. How accurate is this prediction? Figure 44 shows the cost in cycles per dispatch for all benchmarks under different BTB prediction schemes (no prediction, prediction by a 16-entry BTB, by a 128-entry BTB and with perfect prediction[1]). No prediction of indirect branches, as in the analytical model, gives a median overhead of 8.6 cycles, very close to the analytical model, which ignored co-scheduling of non-dispatch instructions. This means that less than one cycle is saved by co-scheduling. In other words, the virtual function call sequence forms a critical path in the code. Indirect branch prediction can side-step this path, allowing instructions from the predicted target subroutine to

[1] The perfect prediction cost was not measured, but estimated from the BTB256 miss rate and the overhead in cycles for BTB256 and Noprediction., assuming that the reduction in overhead is proportional to the reduction in miss rate.

execute speculatively, in parallel with the call itself. In this section, we studied this effect for BTB prediction.

A BTB reduces the effective branch penalty since the full penalty B is only incurred upon a misprediction. The analytical cost model could be improved by using the effective branch penalty B_{eff} = B * btb_misprediction_ratio. For the standard benchmarks, with a median misprediction ratio of 24%, this model predicts a cost of 6.0 cycles, which still overestimates the real cost (median 3.9 cycles). Considering all benchmarks, the median misprediction ratio of 8% results in an estimated cost of 5.3 cycles per dispatch, which overestimates the actual median of 2.8 cycles / dispatch by about a factor two.

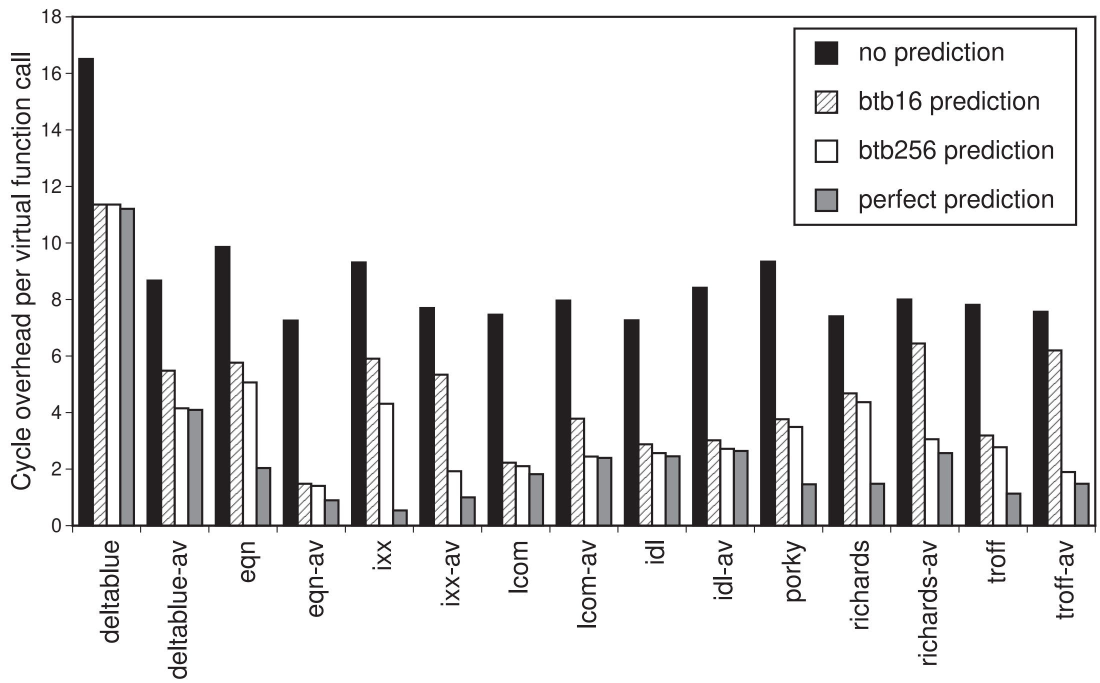


Figure 44. Cycles per dispatch under various BTB prediction regimes

Figure 44 demonstrates the importance of indirect branch prediction. Even a small BTB with 16 entries reduces the median cost of the standard benchmarks from 8.5 to 4.2 cycles, a factor two improvement. A BTB of 256 entries reduces this cost slightly to 3.9 cycles. Beyond 256 entries, a BTB no longer improves prediction accuracy. More sophisticated prediction mechanisms have the potential to reduce call cost by another factor two, as shown by the median 1.6 cycles per call that results from a perfect (i.e. 100% accurate) predictor.

Dispatch cost varies widely over the different benchmark programs: a single dispatch costs 2.1 cycles in *lcom* but 11.4 cycles in *deltablue*, a difference of a factor of 5.4. This variation illustrates the combined effects of the factors discussed previously, such as the BTB hit ratio and the co-scheduling of application code. The dispatch cost of the all-virtual programs varies much less since the average cost is dominated by very predictable monomorphic calls (i.e., call sites invoking the same function every time).

6.5 Discussion

What does this detailed dispatch performance analysis tell us? Will dispatch performance improve with future hardware? Should programmers write their applications differently to improve performance?

First, the median dispatch overheads we observed for P96 (5.2% for the standard benchmarks and 13.7% for the all-virtual versions) can be used as a bound on the dispatch performance improvements one can hope to obtain. Thus, no matter how good a dispatch mechanism is, we cannot hope for much more than a performance improvement of around 5-10% for the C++ measured here. Java programs, where all member functions are declared virtual, can gain between 5% and 50%. Any further improvement must come from other optimizations such as customization or inlining [16][70]. Given that better optimizing compilers are possible [6], it hardly seems appropriate for programmers to compromise the structure of their programs to avoid dispatch.

Many object-oriented systems use or could use VFT-like dispatch mechanisms (e.g., implementations of Java, Modula-3, Oberon-2, and Simula), and thus this study bears some significance for those languages as well. While the characteristics of typical programs may differ from the C++ programs measured here, the general trends should be similar. Together, the standard and all-virtual programs represent a wide spectrum of program behaviors and call frequencies, and thus we expect many programs written in other languages to fall somewhere within that spectrum. Furthermore, the dependency structure (and thus performance on superscalar processors) of many other dispatch mechanisms (e.g., selector coloring or row displacement) is similar to VFT, as we have shown in section 5.2. Therefore, the measurements presented here should apply to these dispatch mechanisms as well.

Although simulations provide accurate numbers, they are inordinately expensive and complicated. As discussed in section 6.4.8, the analytical model for VFT dispatch cost developed in section 5.2 already establishes a *bound* on dispatch cost fairly well using only two parameters.

Finally, will dispatch overhead increase in the future? We believe so. As Figure 43 showed, the relative overhead will increase as processors issue more instructions per cycle. At an issue width of 16, the median overhead increases by about 26%. Future processors might also have longer load latencies, further increasing dispatch cost. General compiler optimizations may also influence dispatch performance. Much current research focuses on compilation techniques to increase instruction-level parallelism. If compilers successfully reduce execution time on wide-issue processors, the effective dispatch overhead could further increase for programs with unpredictable VFT calls. The most important aspect of future hardware designs, when it comes to virtual function call cost, is the optimization of indirect branch execution by branch target prediction. Without target prediction, virtual function calls are twice as expensive, compared to a processor with a sufficiently large BTB. In summary, over the next few years, we expect the relative dispatch cost to rise, though the exact extent is hard to predict.

6.6 Summary

We have analyzed the direct dispatch overhead of the standard virtual function table (VFT) dispatch on a suite of C++ applications with a combination of executable inspection and processor simulation. Simulation allows us to precisely define dispatch overhead as the overhead over an ideal dispatch implementation using direct calls only. On average, dispatch overhead is significant: on a processor resembling current superscalar designs, programs spend a median overhead of 5.2% and a maximum of 29% executing dispatch code. However, many of these benchmarks use virtual function calls quite sparingly and thus might underrepresent the actual "average" C++ program. For versions of the programs where every function was converted to a virtual function to simulate programming styles that extensively use abstract base classes defining virtual functions only (C++'s way of defining interfaces), the median overhead rose to 13.7% and the maximum to 47%. On future processors, this dispatch overhead is likely to increase moderately.

On average, thunks remove a fourth of the overhead associated with the standard implementation of virtual function calls. For some programs the difference is much higher since thunks remove a data dependency chain that inhibits instruction level parallelism.

To our knowledge, this study is the first one to quantify the direct overhead of dispatch in C++ programs, and the first to quantify superscalar effects experimentally. In addition to measuring bottom-line overhead numbers, we have also investigated the influence of specific processor features. Although these features typically influence the absolute dispatch cost considerably (i.e., the number of cycles spent in dispatch code), the relative cost (the percentage of total execution time spent in dispatch code) remains fairly constant for most parameters except for extreme values. Thus, the overheads measured here should predict the actual overhead on many current processors reasonably well.

Since many object-oriented languages use virtual function tables for dispatch, and since several other dispatch techniques have identical execution characteristics on superscalar processors, we believe that our study applies to these languages as well, especially if their application characteristics fall within the range of programs studied here.

Since indirect branch prediction, studied here in the form of a BTB branch target predictor, has such a large impact on virtual function call cost (see section 6.4.8), we study indirect branch prediction mechanism in the remainder of this work.

7 Hardware techniques for efficient polymorphic calls

"We should give up the attempt to derive results and answers with complete certainty"
Michael O.Rabin [115]

In the previous section we established the importance of indirect branch misprediction in the call cost of virtual functions in C++. Superscalar processors are likely to hide most of the cost of table-based polymorphic call resolution by co-scheduling surrounding instructions, but only when the indirect branch is predicted correctly. Since the performance of a branch target buffer maxed out at about 128 entries, more accurate prediction cannot be bought even if a larger transistor budget is available, unless better prediction techniques can be implemented in hardware. Therefore we studied more complex prediction mechanisms in the remainder of this work

7.1 Software vs. hardware prediction

When prediction of indirect branches is unavailable in hardware, software techniques that adapt to observed behavior and predict that it will reoccur can outperform static table-based techniques like Virtual Function Table dispatch (VTBL). The latter offer better worst-case performance but fail to exploit temporal locality at a call site. The simple form of prediction implemented in an inline cache (jump to the last observed target) is sufficient to achieve better performance on average than all forms of dispatch tables on a high-frequency, deeply pipelined processor without indirect branch prediction (Section 5.2).

However, when indirect branch prediction is implemented in a processor's micro-architecture, table-based techniques become competitive with inline caches. They expose the polymorphism of the call to the hardware prediction architecture in the form of an indirect branch, where the inline cache provides its own prediction in the form of a direct jump. The hardware counterpart of an inline cache is a Branch Target Buffer (BTB). A BTB stores the most recent target address of a branch.

Inline caching has several advantages over a BTB:

- No cache size limitation: since the predicted target addresses are stored inline, the cache grows with the program. A BTB is limited in size by the transistor budget allocated to branch prediction.
- No conflict misses: in an inline cache, every polymorphic call site has its own predicted target. A BTB has limited associativity, and will therefore suffer from interference on the branch address.
- Ease of implementation: an inline cache can easily be combined with other dispatch techniques. Especially in dynamically typed languages, a slow call resolution technique can

be optimized dramatically by placing it as backup technique to an inline cache, with a relatively small programming effort. A BTB requires a static, table-based dispatch technique in order to work. In dynamically typed languages, such techniques require substantial implementation effort. They also incur a compile time cost in the form of a table compression algorithm (see Section 4).

A BTB, on the other hand, has the following advantages over an inline cache:

- A potentially higher hitrate: if the class of the receiver object of a polymorphic call is not identical to the last observed class, an inline cache will treat this as a misprediction, *even if the actual target method is the same* (for instance, when the class is a subclass of the last observed class and does not override the message definition). A BTB will treat this as a hit (in the absence of conflict and capacity misses).

- A faster miss case: if the inline cache misses, a backup call resolution technique finds the correct target and replaces the inlined direct jump target. The instructions that accomplish this execute in competition with regular code, where hardware-based prediction will update a call target in parallel, without interfering with regular program execution. A miss therefore incurs a cost bound only by the branch misprediction penalty.

- Greater potential for hit rate improvement: hardware-based prediction can use history-based information, such as the trace of recently executed calls, without slowing down regular execution. As shown in Sections 8 and 9, this information can reduce the misprediction rate by a large factor. Though an inline cache can also use history information to obtain more accurate prediction, the instructions that maintain this information run in competition with regular code, implying an extra overhead for every polymorphic call (not just the miss cases). This does not seem cost-effective.

- Wider applicability: hardware-based indirect branch prediction also speeds up polymorphic calls in non-object oriented languages (for instance, large switch statements or calls through a function pointer table). Even object-oriented programs often execute indirect branches that are not generated by message dispatch instructions. For example, 52% and 66% of the indirect branches in *ixx* and *eqn* are generated by switch statements (see Table 17).

In summary, if hardware-based prediction is unavailable or inadequate because of large program working sets, software-based prediction is a good, easy to implement alternative. With adequate hardware-based prediction, software-based prediction is to be avoided, since it denies the hardware the opportunity to do a more accurate prediction with a faster miss case.

7.2 Hardware indirect branch prediction

Prediction has a long history in hardware. Caches, filebuffers, RAM disks, virtual memory systems, disk drivers, all use the principle: what has occurred in the past is likely to occur again, usually pretty soon.

Indirect branch prediction is nothing more than target address caching and prefetching. It is only a little more specialized than an instruction cache, which, in current practice, already incorporates conditional branch prediction. For example, the UltraSparc I-cache associates a 2-bit counter and a “next” field with every 4 instructions [124]. This field points to the next I-cache line to be fetched if a conditional branch is predicted as taken. The UltraSparc therefore merges functionality of a BTB with the I-cache. Many of problems we encountered in the course of this work resemble problems encountered in cache design. For example, many of the solutions in Section 8 come straight out of cache design literature [65]. Others are inspired by good conditional branch designs, for example the GSHARE predictor [98].

To see which of these solutions carry over to indirect branch predictors, we have to measure their predictor performance on real programs. We use misprediction rate, defined as branch misprediction/frequency, as the metric to evaluate designs. Minimization of misprediction rate ensures that often executed branches carry more weight.

7.3 Indirect branch frequency

How important are indirect branches? We answered this question partly in the previous section, since every virtual function call in C++ has an indirect branch as its core instruction. The overhead of 5% and 14%, was mostly due to mispredicted indirect branches, since the other instructions have no large impact on performance. However, not all indirect branches correspond to virtual function calls. Of the seven C++ programs in our benchmark suite, discussed in the next section, two execute more switch statements than virtual function calls. In non-object oriented languages, the only way to have polymorphic calls is to hand-code polymorphic call resolution procedures, which use large switch statements or function pointer tables, both of which generate indirect branches.

7.3.1 Benchmark overview

Our main benchmark suite consists of large object-oriented C++ applications ranging from 8,000 to over 75,000 non-blank lines of C++ code each (see Table 17), and *beta*, a compiler for the Beta programming language [90], written in Beta. We also measured the SPECint95 benchmark suite with the exception of `compress` which executes only 590 branches during a complete run. Together, the benchmarks represent over 500,000 non-comment source lines.

All C and C++ programs except `self`[1] were compiled with GNU gcc 2.7.2 (options -O2 -multrasparc plus static linking) and run under the `shade` instruction-level simulator [29] to obtain traces of all indirect branches. Procedure returns were excluded because they can be predicted accurately with a return address stack [82]. All programs were run to completion or until six million indirect branches were executed.[1] In `jhm` and `self` we excluded the initialization phases by skipping the first 5 and 6 million indirect branches, respectively.

For each benchmark, Table 17 lists the number of indirect branches executed, the number of instructions executed per indirect branch, the number of conditional branches executed per indirect branch, and the percentage of indirect branch executions that correspond to the branch classes used in Section 9.2.1 and Section 9.2.2, as well as the number of branch sites responsible for 99%, and 100% of the branch executions. For example, only 5 different branch sites are responsible for 99% of the dynamic indirect branches in `go`. The SPECint95 programs are dominated by very few indirect branches, with less than ten interesting branches for all programs except `gcc`. Four of the SPEC benchmarks execute more than 1,000 instructions per indirect branch. Since the impact of branch prediction will be very low for the latter four benchmarks, we exclude them when optimizing predictor performance (by minimizing the AVG misprediction rate).

[1] `self` does not execute correctly when compiled with -O2 and was thus compiled with "-O" optimization. Also, `self` was not fully statically linked; our experiments exclude instructions executed in dynamically-linked libraries.

[1] We reduced the traces of three of the SPEC benchmarks in order to reduce simulation time. In all of these cases, the BTB misprediction rate differs by less than 1% (relative) between the full and truncated traces, and thus we believe that the results obtained with the truncated traces are accurate.

Name	Description	Style	K lines of code	# of indirect branches	instr. / indirect	cond. / indirect	virtual %	switch %	indirect %	1 target %	2 targets %	> 2 targets %	active branches	
													99 %	100 %
idl	IDL compiler[a]	OO	14	1,883,641	47	6	93.2	3.2	3.6	97.1	0.1	2.8	70	543
jhm	JHM[b] 6-12M	OO	15	6,000,000	47	5	93.6	1.2	5.2	58.7	1.4	39.9	34	155
self	Self-93 VM: 5-6M	OO	77	1,000,000	56	7	76.0	4.4	19.6	40.1	31.6	28.3	848	1855
xlisp	SPEC95	C	55	6,000,000	69	11	0.0	0.1	99.9	38.9	9.0	52.1	4	13
troff	GNU groff 1.09	OO	19	1,110,592	90	13	73.7	12.5	13.8	41.9	13.6	44.5	61	161
lcom	HDL[c] compiler	OO	14	1,737,751	97	10	63.2	36.8	0.0	33.5	54.0	12.5	87	328
AVG-100: instr/indirect < 100			24	2,955,331	68	9	66.6	9.7	23.7	51.7	18.3	30.0	184	509
perl	SPEC95	C	21	300,000	113	17	0.0	31.7	68.3	41.2	0.0	58.8	7	24
porky	scalar optimizer[d]	OO	23	5,392,890	138	19	70.6	23.8	5.6	15.6	8.1	76.3	89	285
ixx	IDL parser[e]	OO	12	212,035	139	18	46.5	52.2	1.3	37.1	6.4	56.5	91	203
edg	C++ front end	C	114	548,893	149	23	0.0	62.4	37.6	7.9	29.6	62.5	186	350
eqn	equation typesetter	OO	8	296,425	159	25	33.8	66.2	0.0	4.2	37.8	58.0	58	114
gcc	SPEC95	C	131	864,838	176	31	0.0	31.5	68.5	0.8	1.7	97.5	95	166
beta	BETA compiler	OO	72	1,005,995	188	23	0.0	2.3	97.7	18.7	28.1	53.2	135	376
AVG-200: 100 < instr/indirect < 200			55	1,231,582	152	22	21.6	38.6	39.9	17.9	16.0	66.1	94	217
AVG: instr/indirect < 200			40	2,027,158	113	16	42.4	25.3	32.4	33.5	17.0	49.5	136	352
AVG-OO: OO, instr/indirect < 200			28	2,071,037	107	14	61.2	22.5	16.3	38.5	20.1	41.3	164	447
AVG-C: C, instr/indirect < 200			68	1,928,433	127	21	0.0	31.4	68.6	22.2	10.1	67.7	73	138
m88ksim	SPEC95	C	12	300,000	1.8K	233	0.0	46.2	53.8	2.9	10.3	86.8	5	17
vortex	SPEC95	C	45	3,000,000	3.5K	525	0.0	30.7	69.3	23.1	16.9	60.0	10	37
ijpeg	SPEC95	C	17	32,975	5.7K	441	0.0	97.8	2.2	96.7	3.2	0.1	7	60
go	SPEC95	C	29	549,656	56K	7123	0.0	99.0	1.0	0.2	0.0	99.8	5	14
AVG-infreq: instr/indirect > 200			26	970,658	17K	2081	0.0	68.4	31.6	30.7	7.6	61.7	7	32

Table 17. Benchmarks and commonly shown averages (arithmetic means)

[a] SunSoft version 1.3
[b] Java High-level Class Modifier
[c] hardware description language compiler
[d] SUIF 1.0
[e] Fresco X11R6 library

7.3.2 Branch frequency measurements

Programs with the highest average branch frequency, as shown in Table 17, execute 47 instructions per indirect branch. However, in some program phases, indirect branches occur much more frequent. Figure 45 shows the portion of indirect branches executions that occurs less than five instructions apart, between five and nine, and so on in steps of five instructions. The histograms show that although the average number of instructions between branches is 113, the mean distance between branches is much less than that. Only 29% of indirect branches execute more than 100 instructions apart. 25% execute within 20 instructions.

For the indirect branch prediction architectures discussed in later sections, this may reduce the actual hitrate attained in practice. These architectures use the resolved targets of the most recently executed indirect branches before the current branch. The most recent branch may not be resolved, on a deeply pipelined superscalar processor, when the next branch is predicted. A possible solution entails the use of the *predicted* target of recent branches, instead of the *resolved* target (predict ahead). However, this practise may reduce the hitrate. The loss in prediction accuracy due to the use of predicted targets in the history buffer has been explored for conditional branches, by Hao, Chang and Patt [64]. They report that using speculatively targets in the branch history decreases performance less than omitting unresolved branches from the history buffer.

We optimize prediction accuracy by using the indirect branch trace, without simulating other instructions, because this allows us to explore about two orders of magnitude more points in the predictor design space. Unfortunately, this rules out cycle-level simulation, making it hard to estimate the impact of indirect branch clustering

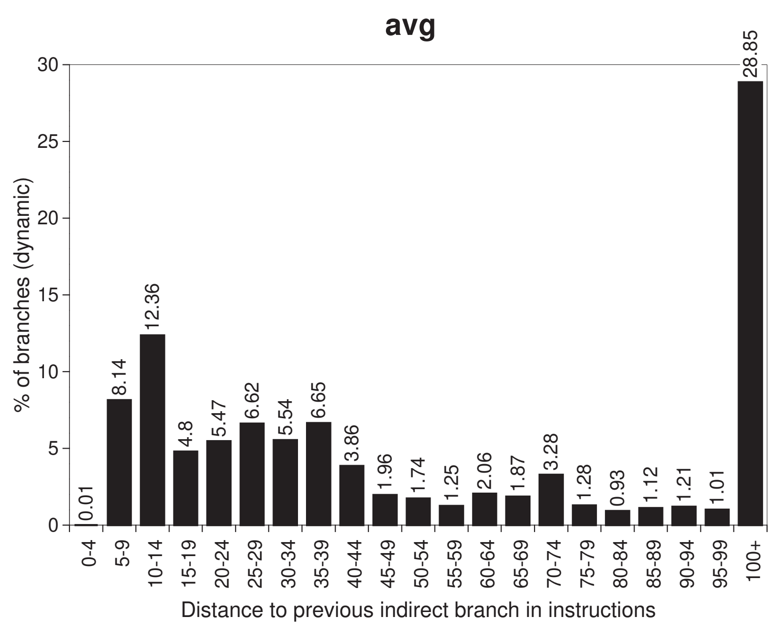


Figure 45. Average execution distance between indirect branches

Figure 46 shows the average distance between executions of the same branch. Here the instruction distance is usually large enough to allow a predictor to update its prediction table in time for the next branch execution.

Branch frequency histograms for individual benchmarks can be found in Appendix B.

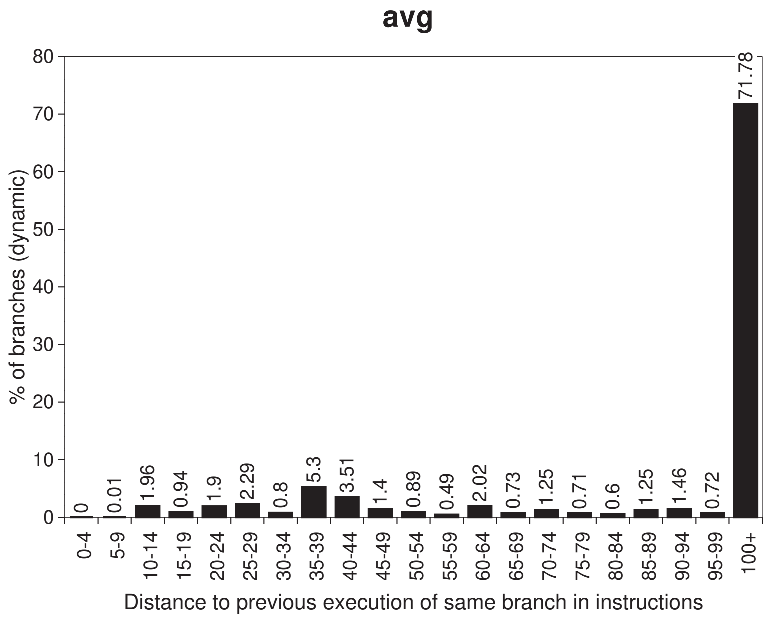


Figure 46. Average distance between executions of the same branch

7.4 Experimental setup

The current experimental setup is shown in Figure 47. To explore a part of the predictor design space, we first decide the range of parameters (if necessary, the branch predictor simulator must be extended in C++). Subsequently, a dedicated perl script generates all the simulation jobs. These jobs are distributed over a collection of workstations by a processor farmer shell script. After they are executed (the main computation cost), specific misprediction rate measurements are extracted from the result files and organized into tables.

7.5 Problem statement

In the next two sections, we look at hardware techniques to reduce the average overhead of indirect branches by reducing branch misprediction rates. Indirectly, this will reduce the overhead of polymorphic calls by avoiding the branch penalty associated with mispredicted branches. We aim to answer the following questions:

- For a given branch prediction technique, what is the limit of prediction accuracy reachable in the absence of hardware constraints?

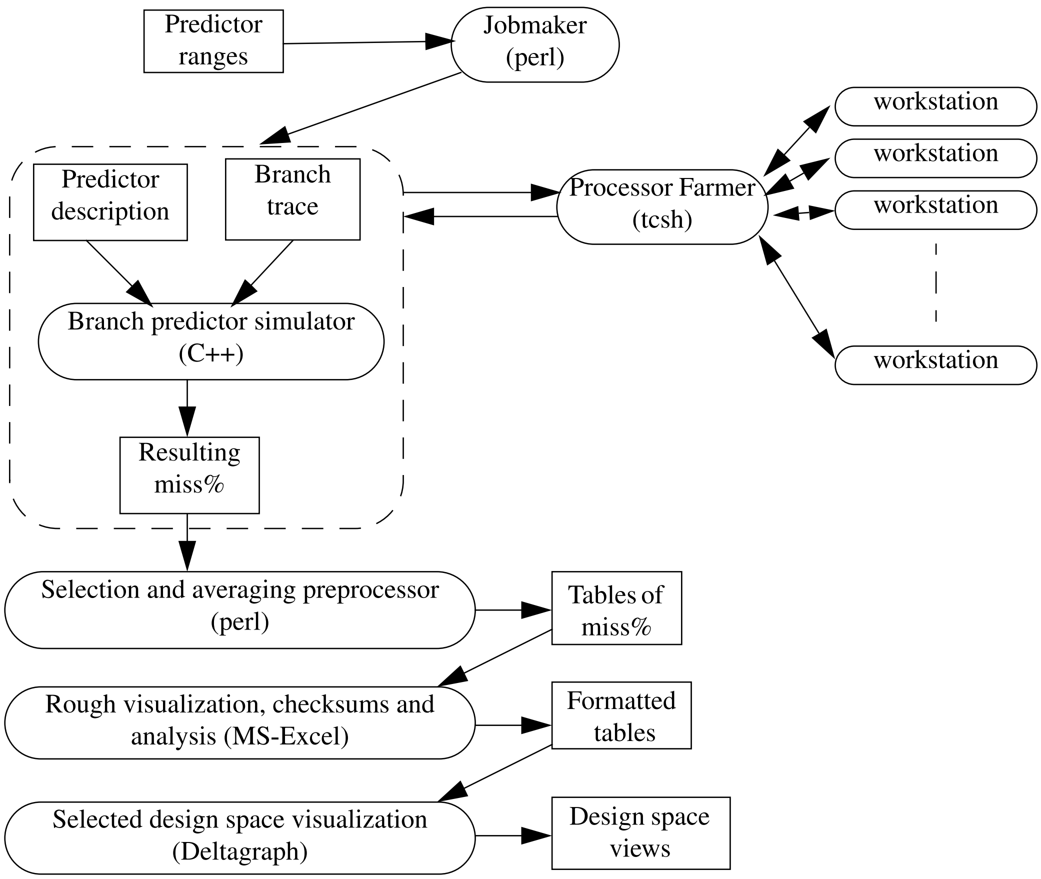


Figure 47. Experimental setup

- In the presence of hardware constraints such as limited table size and limited associativity, how close can a technique approach this ideal?
- Given a particular hardware budget, how do we maximize prediction accuracy?
- Given a particular desired prediction accuracy, how can we minimize its cost?

Our aim is to reach adequate (+90%) prediction accuracy for predictor architectures that use a limited, practical transistor budget (64Kbit tables with simple logic).

8 Basic indirect branch predictors

"Many existing branch prediction schemes are capable of accurately predicting the direction of conditional branches. However, these techniques are ineffective in predicting the targets of indirect jumps...",

Po-Yung Chang, Eric Hao, Yale Patt [25]

We investigate a wide range of two-level predictors dedicated exclusively to indirect branches. We first study the intrinsic predictability of indirect branches by ignoring any hardware constraints on memory size or logic complexity. Then we progressively introduce hardware constraints and minimize the loss of predictor performance at each step. Thus, we initially assume unconstrained, fully associative tables and full 32-bit addresses (unless indicated otherwise).

8.1 Branch target buffer

Current processors use a branch target buffer (BTB) to predict indirect branches (see Section 6.2.1 for more detail). The predictor uses the branch address as a key into a table (the BTB) which stores the last target address of the branch (Figure 48).

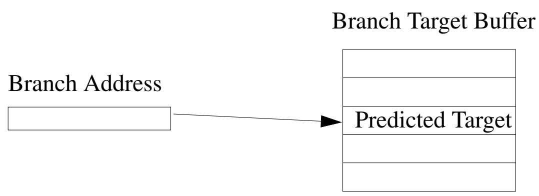


Figure 48. Branch target Buffer

8.1.1 2-bit counter update rule

We simulated two variants: "BTB" is a standard BTB which updates its target address after each branch execution. "BTB-2bc" is a BTB with two-bit counters which updates its target only after two consecutive mispredictions[1]. BTB-2bc predictors perform better in virtually all cases, with an average of 24.9% misprediction rate, compared to 28.1% for a standard BTB. Polymorphic branches occasionally switch their target but are often dominated by one most frequent target, a situation observed in object-oriented programs [6][36]. But even with two-bit counters BTB accuracy is quite poor, ranging from average misprediction ratios of 20% in OO programs to

[1] In conditional branch predictors, the latter strategy is implemented with a two-bit saturating counter (2bc), hence the name. For an indirect branch, one bit suffices to indicate whether the entry had a miss the last time it was consulted.

37% for C programs. Infrequent indirect branches (AVG-200) are less predictable, with a misprediction average of 38% vs. 10% for the programs in AVG-100.

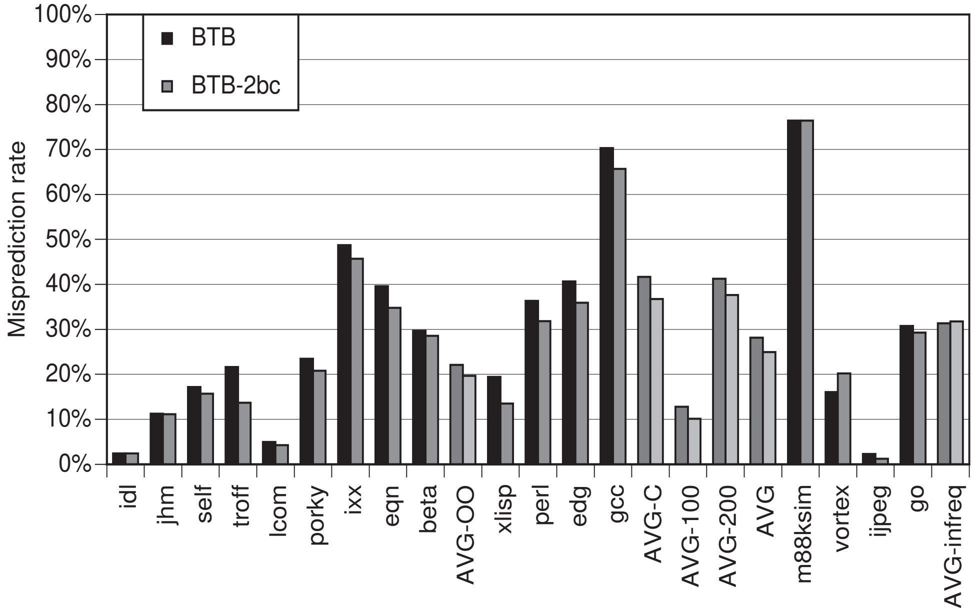


Figure 49. Indirect branch misprediction rates for an unconstrained BTB

8.2 Two-level predictor

Two-level predictors improve prediction accuracy by keeping information from previous branch executions in a history buffer. Combined with the branch address, this history pattern is used as a key into the History Table which contains the predicted target addresses. As in BTBs, the entries can be updated on every miss or after two consecutive misses (2-bit counters). We tested every predictor in this section with both variants, and always saw a slight improvement with 2-bit counters. I.e., ignoring a stand-alone miss when updating seems to be a good strategy in general. Thus, we will only show 2-bit counter results in the rest of the paper.

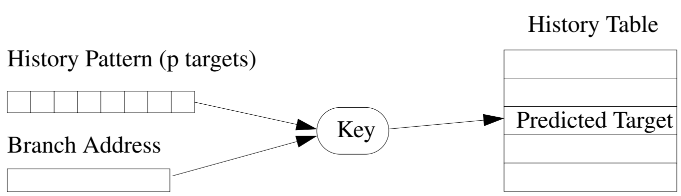


Figure 50. Two-level branch prediction

For conditional branches, a branch history of length p consists of the taken/not-taken bits of the p most recently executed branches [135]. In contrast, most indirect branches are unconditional, and thus keeping a history of taken/not-taken bits would be ineffective. Instead, the history must consist of previous target addresses or bits thereof. Such a path-based history could also be used to predict conditional branches, but since taken/not-taken bits succinctly summarize the target addresses of a conditional branch, conditional branch predictors usually do not employ target address histories (but see [105]).

8.2.1 First level: history pattern

Branch predictors can use one or more history buffers. A *global history* uses a single history buffer (correlation branch prediction), and all branches are predicted using the outcomes of the p most recently executed branches. In contrast, a *per-address history* keeps a separate history for each branch, so that branches do not influence each other's prediction. Finally, *per-set history* prediction forms a compromise by using a separate history for each set of branches, where a set may be determined by the branch opcode, a compiler-assigned branch class, or a particular address range (see Yeh and Patt's comprehensive study [135]).

To investigate the impact of global vs. local histories, we simulated per-set histories where a set contains all branches in a memory region of size 2^s bytes, i.e., all branches with the same values in bits s..31 fall into the same set (Figure 51). With this parametrization, a global history corre-

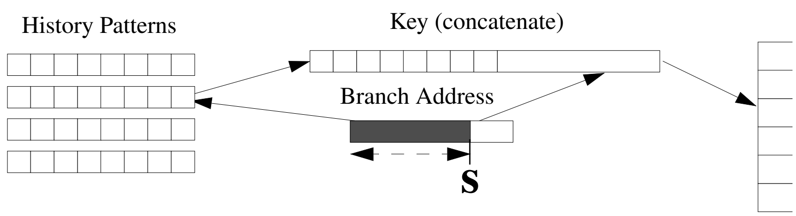


Figure 51. History pattern sharing

sponds to s=31, and per-branch histories correspond to s=2. Using the results of the exhaustive run on a limited benchmark suite, we obtained good initial values for the other parameters (path length p=8, per-branch pattern entries in the history table). The results are shown in Figure 52.

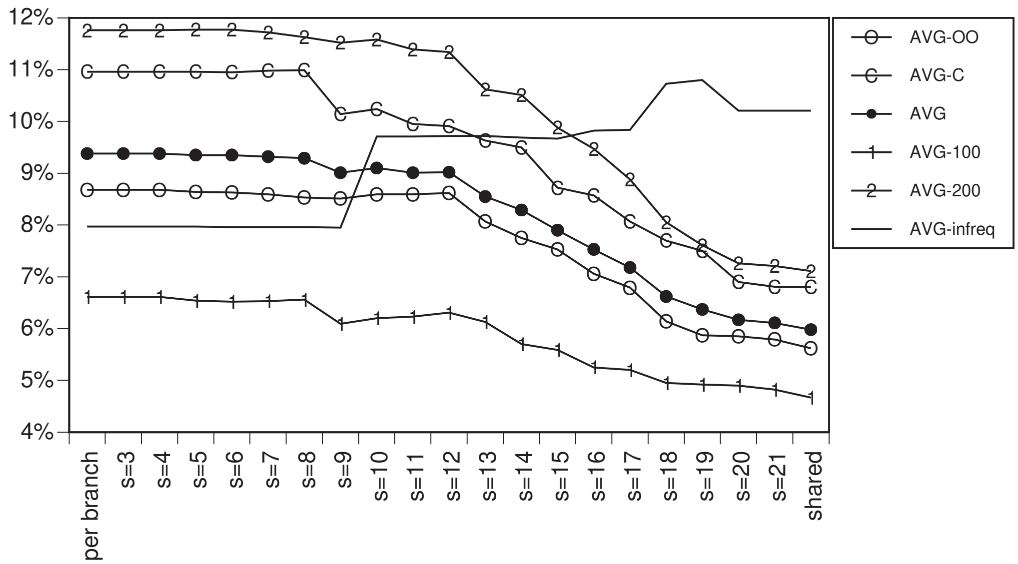


Figure 52. Influence of history sharing for path length p=8, per-branch entries

In general, a global history outperforms local histories for all benchmark groups except `AVG-infreq`. *AVG* declines from 9.4% with per-address paths down to 6.0% with a global path. The OO programs benefit most from sharing paths, with misprediction rates falling from 8.7% to 5.6%. This result indicates a substantial correlation between different branches (i.e., inter-branch correlation) in our benchmark suite, a correlation not limited by code distance. This result is analogous to the results for conditional branches in Yeh and Patt's study [135], where a global predictor generally performs better than a per-address scheme.

The C benchmarks show a pronounced dip for s=9 (i.e., if branches within 512-byte code regions share paths). On closer observation, the dip is caused by `xlisp` where only three indirect branches are responsible for 95% of the dynamic indirect branch executions. For `xlisp`, moving from s=8 to s=9 reduces mispredictions by a factor of three. Similarly, at s=10 `go`'s misprediction ratio jumps from 26% to 33% (`go` is dominated by two indirect branches), which causes *AVG-infreq* to jump at s=10.

The programs in *AVG-infreq* (which execute indirect branches very infrequently) are the only ones benefiting from per-address histories (*AVG-infreq*). Apparently, the targets of different branches do not correlate well with each other since they occur very far apart. Since these programs use indirect branches only sparingly, we can safely ignore their different behavior when designing branch predictors.

8.2.2 Second level: history table sharing

A two-level predictor uses the history pattern to index into a history table that stores predicted targets. Again, we have a choice of per-branch, per-set, or global prediction. We simulated per-set tables that grouped all branches with identical address bits *h*..31 into the same set (see Figure 53). Thus, *h*=2 implies per-branch history tables (each branch has its own history table)

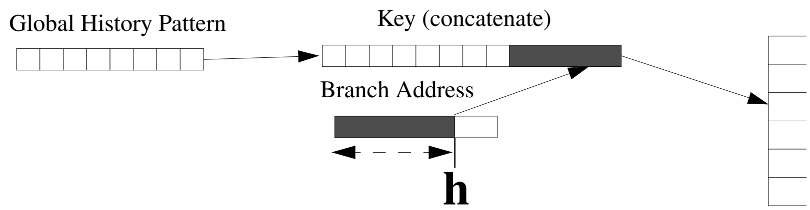


Figure 53. History Table sharing

and *h*=31 implies a single shared history table (i.e., all branches with the same history share the same prediction). Figure 54 shows that the branch address matters: The misprediction average

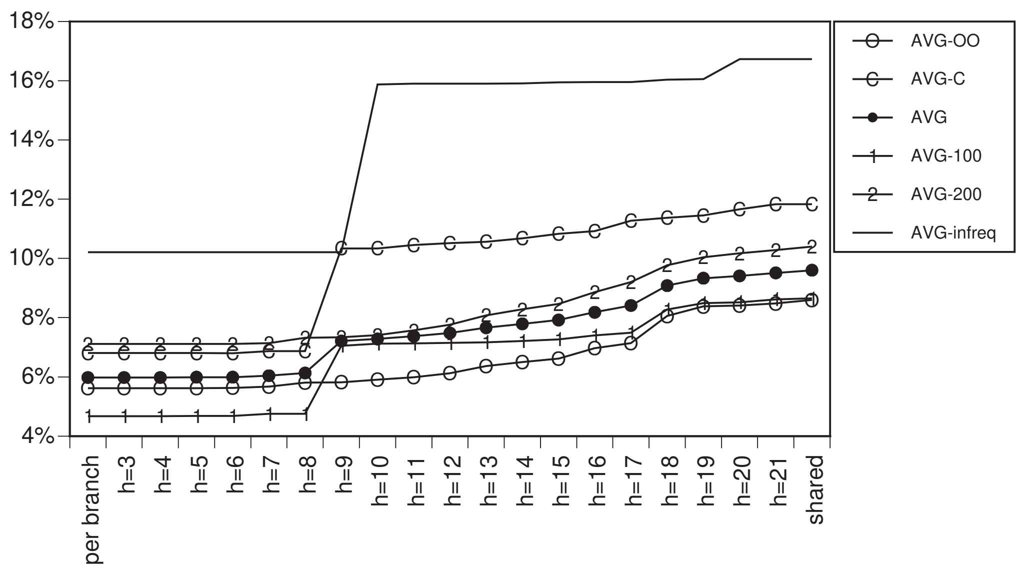


Figure 54. Influence of history table sharing for path length 8 with a global history pattern.

for all benchmarks increases from 6.0% for per-address history tables to 9.6% for a globally shared history table, the rate of the OO programs increases from 5.6% to 8.6%, and that of the C benchmarks rises from 6.8% to 11.8%. (Again, *xlisp* changes dramatically at *h*=9, causing a sharp increase for some averages.) Therefore, we will only consider per-address tables (*h*=2) in subsequent experiments.

8.2.3 Path length

The history pattern consists of target addresses of recently executed branches. The history buffer is shared (global), so all indirect branches influence each other's history. Concatenation with the branch address results in the key used to access the history table. The path length *p* determines the number of branch targets in the history pattern. In theory, longer paths are better

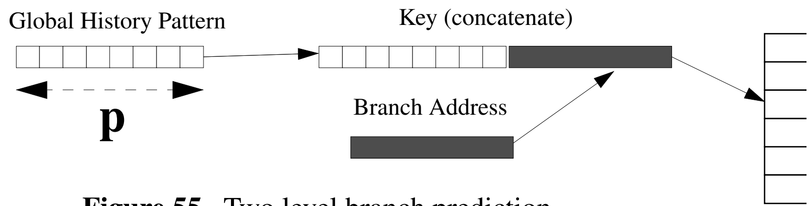


Figure 55. Two level branch prediction

since a predictor cannot capture regularities in branch behavior with a period longer than *p*. Shorter paths have the advantage that they adapt more quickly to new phases in the branch behavior. A long path captures more regularities, but the number of different patterns mapping to a given target is larger, so it takes longer to fill in the table. This long "warm-up"-time for long patterns can prevent the predictor from taking advantage of longer term correlations before the program behavior changes again. We studied path lengths up to 18 target addresses in order to investigate both trends and see where they combine for the best prediction rate.

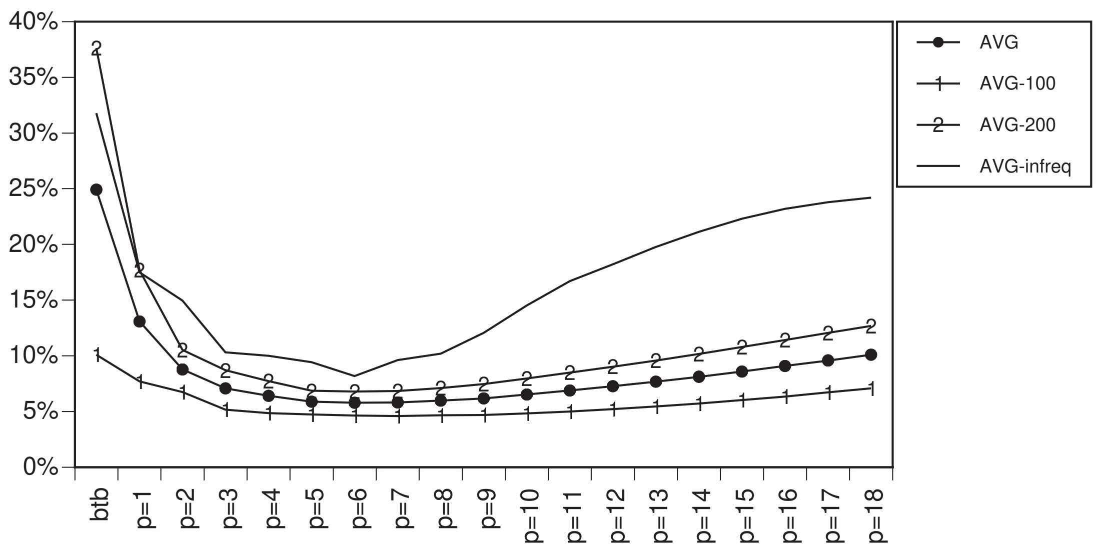


Figure 56. Misprediction rates per path length (global history, per-address table entries)

Figure 56 shows the impact of the history path length on the misprediction rate for all path lengths from 0 to 18. (A path length of 0 reduces the two-level predictor to a BTB predictor since the key pattern consists of the branch address only.) The average misprediction rate drops quickly from 24.9% for a BTB to 7.8% for p=3 and then slowly reaches a minimum of 5.8% at path length 6. Then the misprediction rate starts to rise again and keeps rising for larger path lengths up to the limit of our testing range at p=18. All benchmark suites follow this pattern, although programs with infrequent branches show uniformly higher misprediction rates.

This result indicates that most regularities in the indirect branch traces have a relatively short period. In other words, a predictable indirect branch execution is usually correlated with the execution of less than three branches before it. Increasing the path length captures some longer term correlations, but at path length six cold-start misses begin to negate the advantage of a longer history. At this point, adding an extra branch target to the path may still allow longer-term correlations to be exploited, but on the other hand it will take the branch predictor longer to learn a full new pattern association for every branch that changes its behavior due to a phase transition in the program. A hybrid branch predictor combining both short and long path components should be able to adapt quickly to phase changes while still exploiting longer-term correlations; we experiment with such hybrid predictors in Section 9.

8.3 History buffers

8.3.1 Trace information

We explored a few other choices for the history pattern elements. In the first variant we used both branch address and target, and in the second we included targets of conditional branches in the history. Both resulted in inferior prediction accuracy[1] for any pattern length p (see [46]).

8.3.2 History pattern compression

The global history pattern is a very long bit pattern. For p=8, it consists of 8 * 32 = 256 bits, and concatenation with the branch address results in a total of 288 bits. The information content of this bit pattern is quite low: the number of different patterns that occur during program execution is much smaller than 2^{288}. Since a tag in an associative table includes most of the pattern, long patterns inflate the size of the predictor table. We need to compress the pattern for each path length into a short bit pattern, ideally without compromising prediction accuracy. As a first step towards smaller history patterns, we will only consider path lengths up to size p=12, since longer path lengths result in higher misprediction rates (as seen in Figure 56)

8.3.2.1 Target pattern projection

A straightforward approach for history pattern compression is to select a limited number of bits from each target and concatenate these partial addresses into the history pattern. We explored a

[1] This result is often misunderstood. It does not say that conditional branches are irrelevant to indirect branch prediction. It says that, given a the choice of tracing the last n indirect branches or the last n conditional branches, the indirect branches predict the next indirect branch better. For some individual branches this may be wrong (as suggested by Kalamatianos and Kaeli [81], who dynamically choose a pure indirect or a mixed conditional/indirect trace for each branch), but over the whole benchmark suite, this trend holds.

number of choices by using a range [*a..A*] of the address bits. We varied *a* from 2 to 10, and *A* from *a* to *a*+(*b*-1), where *b* is the largest number of bits that still allows the history pattern to fit within a total of 24 bits (i.e. $b * p <= 24$). Starting with bit *a*=2 worked best on average, and thus we will not show data for other values of a.

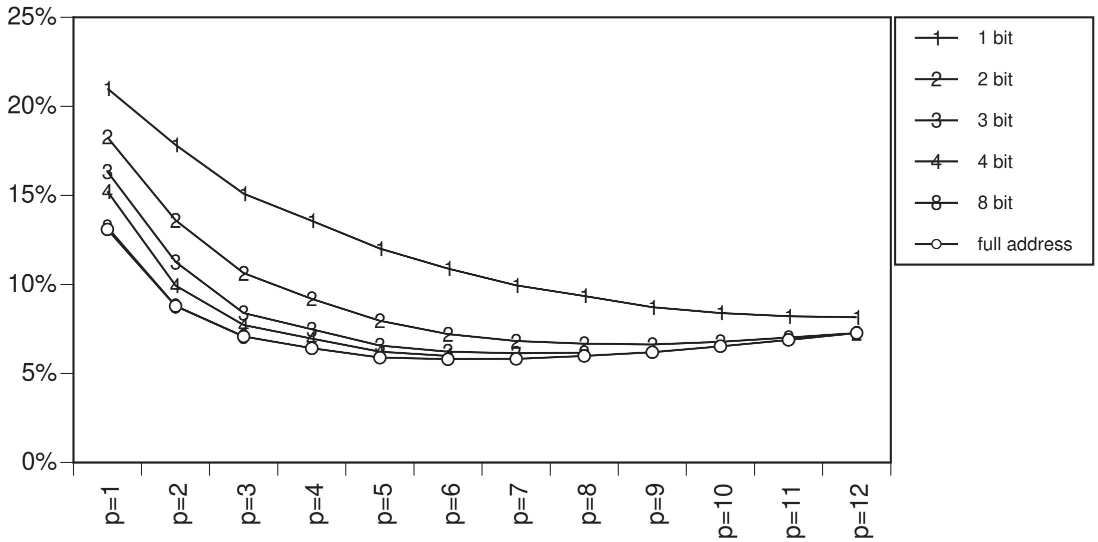


Figure 57. Limited Precision misprediction rates.
AVG for 1, 2, 3, 4, 8 bit and full target addresses.
(low order bits starting from bit 2)

Figure 57 shows the misprediction ratios resulting from the selection of bits [2..2+(*b*-1)], for *b* values of 1,2,3,4 and 8, as well as the misprediction rate for full-precision addresses. The curve for *b*=8 almost completely overlaps with the full-address curve, indicating that 8 bits are enough even for short path lengths. For decreasing address precision, shorter path lengths suffer most. For example, for path length p=10, 2 bits achieve a misprediction rate of 6.77% vs. 6.53% for full addresses, while for path length p=3, the miss ratio decreases from 10.6% (2 bits) to 7.1% (full addresses). A total bit length of 24 bits suffices for the history pattern to approach the full-address performance for all path lengths. Thus, in the rest of the paper we always choose the largest number *b* of bits from each address that keeps $b * p <= 24$. For example, for path length 2 we choose 12 bits for each history entry, and for path length 6 we choose 4.

We also tried two other schemes for target address compression:

- Fold the new target address into the desired number of *b* bits by dividing it into chunks of *b* bits and xor-ing them all together.
- Shift the history pattern *b* bits to the left and xor with the complete new target address.

These variants were intended to use more information of the target address but did not reliably result in better prediction rates and were sometimes even worse. Since they require more logic than the bit selection discussed above, we decided to drop them from further tests[1].

8.3.2.2 Address folding

As mentioned in section 8.2.2, omitting the branch address reduces the performance of a two-level predictor (for p=8, the misprediction rate increased from 6.0% to 9.6%). However, concatenating the branch address with the history pattern results in a key of 24 + 30 = 54 bits. In analogy with the *Gshare* predictor used in conditional branch prediction [24], we can reduce the number of bits in the key pattern to 30 by xor-ing the branch address with the history pattern. Table 18 shows the misprediction rate averages for both alternatives. Compared to the increase

Operation	p=0	p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=8	p=9	p=10	p=11	p=12
Xor	24.91	13.58	8.84	7.09	6.49	6.27	6.01	6.18	6.19	7.44	7.34	7.49	7.67
Concat	24.91	13.08	8.78	7.08	6.48	6.22	5.99	6.13	6.16	6.62	6.77	7.02	7.27
difference	0.00	0.50	0.06	0.01	0.01	0.05	0.02	0.05	0.03	0.82	0.57	0.47	0.40

Table 18. Concatenation versus Xor of history pattern with branch address (AVG)

in misprediction rate due to limited table size and associativity in the next section, the reduction of the key pattern from 54 to 30 bits by xor causes a very small increase in misprediction rate. For example, for path length 8, misprediction rate increases by 0.03%, from 6.16% to 6.19% Since this operation reduces the table space used for tag bits by more than half, we use this scheme in the remainder of this study.

8.4 History tables

In this section we introduce limited table sizes and limited associativity in order to obtain practical indirect branch predictors.

8.4.1 Capacity misses

Limited tables introduce a new source of branch misses: capacity misses. When the table is too small to store the history patterns of all branches in its working set, some patterns will be evicted from the table, resulting in capacity misses.

Longer path lengths generate more patterns for a given set of branches. For example, *ixx* generates 203 different patterns for path length *p*=0, 402 for *p*=1, 865 for *p*=2, 1469 for *p*=3, and ends

[1] Since the difference in prediction accuracy between partial bit-selected addresses and full-precision addresses is already very small, there is not much to be gained by sophisticated hashing schemes. We measured a difference of 0.2%, when going from a fullprecision path length 6 to a 4-bit path length 6 predictor. One could hope for positive interference, but this is much less likely to occur in indirect branch prediction than in conditional branch prediction, since the predicted value is a 32bit address, not just a bit (which is 50% likely to positively interfere on a miss).

up with 9403 patterns for p=12. Though not all patterns are used more than once (some only occur once in the warm-up phase), for longer path lengths capacity misses will occur fairly soon. A predictor with a longer path length may be more accurate than a predictor of shorter path length for an unlimited table, but the capacity misses caused by a small table size can affect the longer path length predictor enough to negate this advantage.

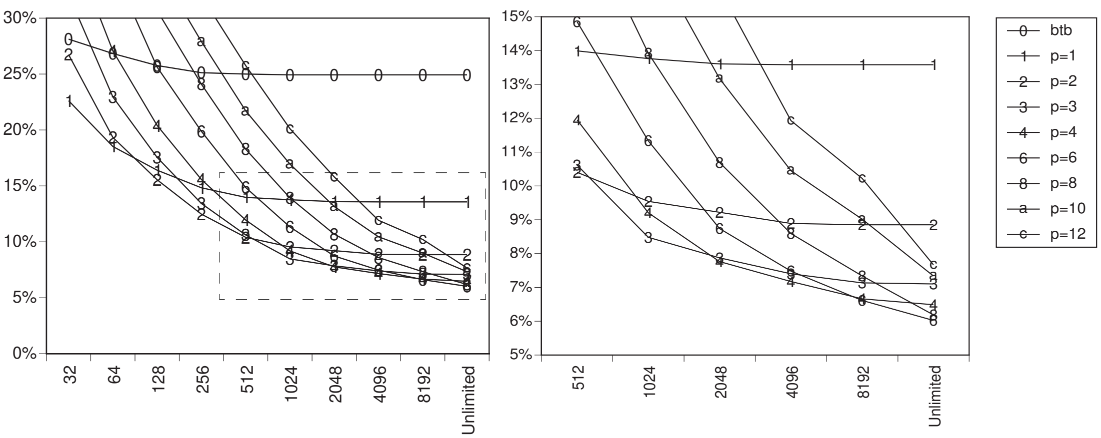


Figure 58. AVG of limited size fully-associative misprediction ratios
LRU replacement policy
(cut-out section enlarged in right graph)

To estimate the effect of capacity misses we simulate fully-associative tables with LRU replacement policy. Figure 58 shows the average misprediction rate for various fully-associative tables for predictors with path length p=0-4,6,8,10 and 12. The misprediction rate of some path lengths reaches its minimum in the explored range. For p=0 (BTB), the miss rate decreases with increasing table size and reaches its minimum at 256 entries. Since there are no capacity misses left, increasing the table size beyond this point will not lower the miss rate for p=0. Increasing the path lengths pushes this point out to 1024 entries (p=1), 2048 entries (p=2), and 8192 entries (p=3 and p=4). Longer path lengths never completely recover from capacity misses in the explored range. A longer path's ability to detect longer-term regularities can pay off, although the best predictor for each table size is still affected by capacity misses. For instance, p=2 wins at table size 256 with a misprediction rate of 12.5%, 3.6% of which is due to capacity misses. For size 1024, p=3 takes over with a misprediction rate of 8.5%, with 1.4% due to capacity misses. For a 8192-entry table, p=6 (which achieved the lowest misprediction rate for an unlimited table) has a misprediction rate of 6.6%, with 0.6% due to capacity misses.

8.4.2 Conflict misses

In practice, a fully-associative LRU table of sufficient size requires too much logic to implement in hardware, and thus we will explore limited-associative tables in this section.

Limited associativity means that part of the key pattern is used as an index into a table to access a limited set of entries. Each entry in the set has a tag that is checked for equality to the rest of the key pattern. The index part of the key determines how a working set of branch patterns is spread out over the sets, and how many patterns share the same set. For instance, if one only used the high-order 8 bits of the branch address as index in a BTB of 256 sets, most of the patterns would have to share the same set. This can cause conflict misses; these are similar to capacity misses, but it is the capacity of the set instead of the table that is the limiting factor. Conflict misses can be reduced without changing the total size of the table by increasing the associativity or by choosing a different index scheme, so that different patterns share the same sets.We start out choosing the lower order bits of the key pattern as index. In a two-level predictor, this part contains the lower order branch address bits, xor-ed with the target address bits of the recent targets in the history pattern (see section 8.3.2).

We test 1, 2 and 4-way associativity, and tagless tables, which is like 1-way associativity but without tags. Where a one-way associative table will register a miss if the search pattern is not in the table, a tagless table will simply return the target corresponding to the index part of the pattern. We compare misprediction rates for equal table sizes, i.e. a table with 256 sets of one entry each (1-way associative) is compared to a table with 64 sets of four entries each (4-way associative).

We tested all table sizes of the previous section, but will show only selected examples for this analysis to reduce the amount of cluttering in the graphs. Figure 59 (a) shows the misprediction rate of different associativities for a 4096-entry table, for all path lengths.

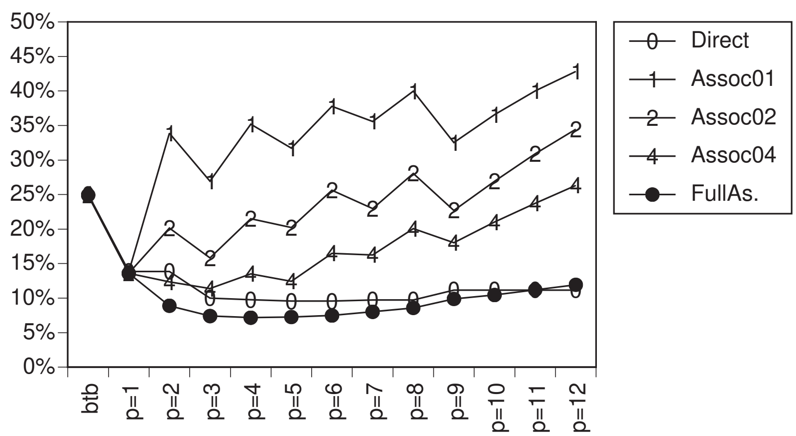


Figure 59. Misprediction rates using concatenation (4K entries)

8.4.2.1 Interleaving

The saw-tooth curve for associativities 1, 2 and 4 indicates that there is something wrong with the way the history pattern is assembled from the target address bits. In particular, for associativity one, the misprediction rate of a p=2 predictor is much higher than a p=1 predictor.

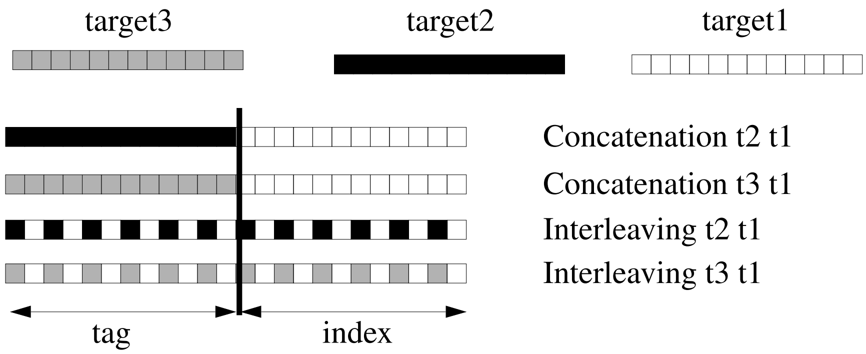


Figure 60. Concatenation and interleaving of target address bits for path length 2 for a 4096-entry table

Figure 60 shows an example for p=2. Since the index part of the pattern is identical for target sequence t2t1 and t3t1, both paths will occupy the same set in the table. The predictor assigns sets in the same way as a predictor of path length one. If the two patterns alternate often, the path length two predictor will incur frequent conflict misses with a one-way associative table and not return a prediction, while the path length one predictor will return the predicted target address. To a lesser degree, the same effect applies to larger path lengths and higher associativities[1], explaining the saw-toothed lines for concatenation in Figure 59. Interleaving remedies this problem by ensuring that the index part of a pattern contains the lower order bits of all target addresses, rather than all bits of a subset of the target addresses. When the target bits are interleaved, target sequences t2t1 and t3t1 will likely differ in the index part of the pattern and will therefore not interfere with each other.

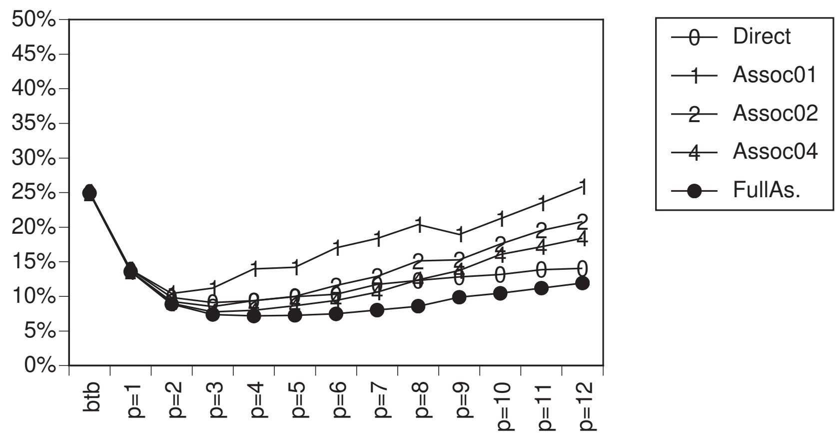


Figure 61. Misprediction rates using reverse interleaving (4K-entries)

[1] Also note that since concatenation places the oldest targets completely in the tag, they are invisible to a tagless table. A path length 12 pattern, with two bits per target in a predictor with a tagless, 4096-entry table will use only the 6 most recent targets, so its effective path length is only 6.

Interleaving of target bits is effective because it spreads patterns over more different sets than concatenation. For example, interleaving increases table utilization for *ixx* from 50% to 79% for a 1024 entry, one-way associative table for path length four. Figure 61 shows that interleaving dramatically improves predictor performance compared to concatenation.

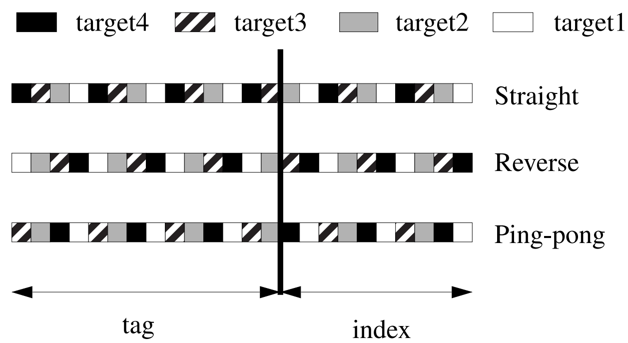


Figure 62. Interleaving schemes for path length 4 with a table of 1024 sets (10-bit index)

We experimented with three variants of interleaving schemes. Figure 62 shows the interleaving schemes for path length 4 and index length 10. The index part of the pattern contains low order bits from all targets, but two targets are more precisely represented with three bits, and two contribute only their two lower order bits. Straight interleaving represents the most recent targets with higher precision (target 1 and 2), while reverse interleaving represents the older targets most precise (target 3 and 4). Ping-pong interleaving represents both the oldest and youngest target more precise (1 and 4). Suppose the current branch depends only on the address of target4, and some of the possibilities are equal in their two lower order bits. With straight interleaving, the two patterns will conflict. With reverse interleaving, they will use entries in different sets.

We found that reverse interleaving performs slightly better on average than the two other schemes. For shorter path lengths, the order does not make much difference since the index part of the pattern contains many bits from every target. For longer path lengths the difference in precision becomes more important. Reverse interleaving gives longer path length predictors the opportunity to use more exact information from older targets, which is their main advantage compared to shorter path lengths. In the remainder of the paper we use reverse interleaving.

8.4.2.2 Associativity

Figure 63 shows that for any given table size and path length, higher associativity results in lower misprediction rates. The only exception is the tagless table, which obtains a lower misprediction rate than a four-way associative table for path length 7 to 12. This effect is caused by positive interference. Since these longer path lengths generate a larger set of distinct patterns, conflict misses occur frequently even in four-way associative tables. The tagless table returns its stored target as a prediction even though it may belong to a different pattern, while the associative table registers a miss. Since many patterns map to a small number of targets, the prediction is better than random so that a tagless table can outperform the associative table. Even where tagless tables do worse than two- or four-way associative tables, the difference in miss rate remains relatively small. Since associative tables require tags and tag checking logic, the hardware implementation of a tagless table is smaller and faster than its associative counterpart, so that it may be the preferable choice under many circumstances.

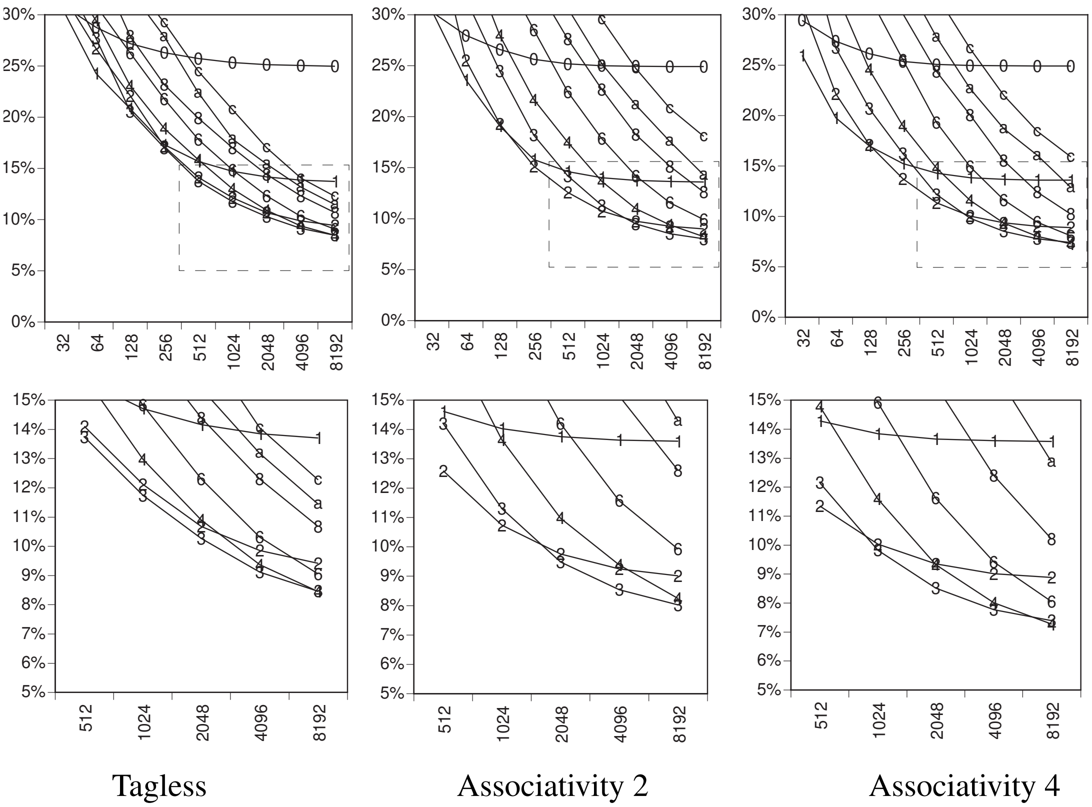


Figure 63. AVG misprediction rates for various table sizes and associativity
Numbers indicate path length, a = 10, c = 12
Bottom graphs enlarge cut-out section

Figure 63 shows the AVG misprediction rates for practical associativities. The best predictor for a given table size changes depending on associativity. For tagless tables, p=3 is best for table sizes 128 to 8192. For 2-way associative tables, p=1 wins for size 128, then p=2 is best for sizes 256 to 1024, after which p=3 performs better. For 4-way associativity, the best predictor for every size up to 1024 is the same as for a fully-associative table (see Figure 58). Then p=3 remains the best choice up to table size 4096. At size 8192, p=4 has a slight edge. P=6 retains too many conflict misses even for large table sizes and therefore loses its status as best practical predictor. Limited table size and associativity prevent the predictor from taking full advantage of the longer-term regularity detection capability of longer path length predictors (however, see the next section). Table A-1 in the appendix shows the exact misprediction rates for the best predictors for all table sizes, and Table A-2 contains their path lengths.

8.5 Summary

We have explored a wide range of two-level indirect branch predictors, starting with unconstrained predictors with full-precision addresses and unlimited hardware resources. For a suite of large C++ and C programs totalling more than half a million lines of source code, the best unconstrained predictor achieved a misprediction rate of 5.8%, indicating that indirect branches are intrinsically predictable even though current hardware predictors (BTBs) do not predict them well. An exhaustive search of the design space established that a global history and per-address predictors perform best.

Subsequent experiments introduced resource constraints in order to evaluate whether realistic predictors could approach this performance with a limited hardware budget. Introducing limited-precision addresses (for a history buffer of 24 bits) increased the misprediction rate to 6.0%. Limiting table size (thus causing capacity misses) resulted in a further increase to a 8.5% misprediction rate for a 1K-entry table and 6.6% for a 8K-entry table. Restricting table associativity resulted in 11.7% and 8.5% misprediction rates for 1K and 8K tagless tables, respectively. Four-way associative tables of the same sizes reduce the misprediction rates to 9.8% and 7.3%, respectively. In comparison, an infinite-size fully-associative branch target buffer achieves a best-case misprediction rate of 24.9%. In other words, two-level prediction improves prediction accuracy by more than a factor three.

We also explored a variety of alternatives that resulted in inferior performance. In particular:

- Per-address or per-set history buffers perform worse than a global, shared history buffer.
- Updating targets on every miss lowers the performance, compared to updating only after two consecutive misses.
- Including conditional branch targets in the history pattern lowers prediction performance by pushing the more relevant indirect branch information out of the history buffer.
- Using bits other than the lower-order bits of target addresses results in lower performance.
- For limited-associative tables, the index part of the key pattern should contain bits from as many targets as possible, i.e., interleaving of target address bits performs better than concatenation.

The difference in performance between a BTB and the best practical two-level predictor becomes significant only for history tables larger than 64 entries. As the hardware budget allows larger history tables to be implemented, the path length of the best predictor grows.

9 Hybrid indirect branch predictors

"I'll get by with a little help from my friends,
I'll get high with a little help from my friends,
Going to try with a little help from my friends.",
The Beatles [11]

As discussed in the previous section, predictors with short path lengths adapt more quickly when the program goes through a phase change because it doesn't take much time for a short history to fill up. Longer path length predictors are capable of detecting longer-term correlations but take longer to adapt and suffer more from table size limitations because a larger pattern set is mapped to the same number of targets. In this section we combine basic predictors into a hybrid predictor in order to obtain the advantages of both.

9.1 Hybrid prediction

When constructing a hybrid predictor, one has two separate but related problems to solve:

- Component selection: which component predictors work well together?
- Metaprediction: given more than one target prediction for a branch, which one do we use?

9.1.1 Components

Ideally, a hybrid predictor's components do not overlap. Each one should specialize on a subset of all branches which it predicts well. If components cover the same area of expertise, resources are wasted, since double work is performed. In this study, we use component predictors that differ in the path length of the history buffer, and in the size of the history table they employ. Each component is an instance on the global history two-level predictor class studied in the previous section. It uses only indirect branch targets in the history. We call such a component a *monopredictor.*

This monopredictor serves as the basic building block for more advanced predictors. Figure 64 shows the setup for path length 8. A 24-bit global history buffer stores three bits of the most recent eight targets[1]. The branch address is xor-ed with this pattern, giving a 24-bit key pattern. This pattern is used to access a history table storing the most recently observed target address of each pattern. Longer path lengths use fewer bits of each target address, so that the length of the pattern remains constant. Unless mentioned otherwise, we use 4-way associative history

[1] Using reverse interleaving, the most accurate projection mechanism tested in Section 8.3

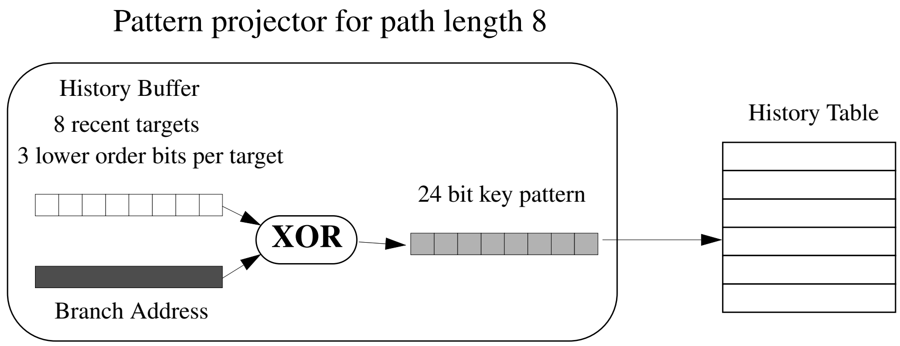


Figure 64. Basic component predictor.

For each path length between 0 and 12, the pattern projector constructs a key of 24 bits, which is then used to access a history table that stores target addresses.

tables[1], and table entries are updated only after two consecutive mispredictions. We will refer to this setup as a *monopredictor* of path length p.

9.1.2 Meta prediction

A hybrid branch predictor combines two or more component predictors that each predict a target for the current branch. The hybrid predictor therefore needs a selection mechanism to predict which of the predictors is likely to be most accurate. We call this prediction of prediction accuracy *metaprediction*. In the literature, at least two metaprediction mechanism are explored. A Branch Predictor Selection Table (BPST), proposed my McFarling [98], associates a two-bit counter with each branch to keep track which of two component predictors is more accurate. After resolving a branch, the counter is updated to reflect the relative accuracy of the two components. Alternatively, branches can be partitioned into different classes based on run-time or compile-time information, and each class is associated with the component predictor best suited to handle it, as proposed by Chang, Hao and Patt [23].

The hybrid predictors presented in this chapter differ mainly by their meta prediction technique. Component predictor path lengths are tuned by varying their separate path lengths until the prediction accuracy, averaged over the benchmark suite (AVG), is maximized.

[1] We show tags in figures only where they are required for correct operation of the predictor, as in the cascaded predictor. However, in the simulations we measure all setups with 4-way associative (tagged) tables in order to remove the noise of extensive conflict misses.

9.2 Branch classification

A classifying hybrid predictor, first explored by Chang, Hao and Patt [23] for conditional branch prediction, assigns to each branch a branch class. All branches that belong to a class use the same component predictor. Classification therefore implements a kind of off-line metaprediction. Before the program is run, it is determined for each branch which component delivers a target prediction. This means that only one component executes for every branch, allowing components to share resources such as a prediction table.

In our experiments, component predictors differ only by global history path length. While every component predictor's history pattern is updated by all branches, only one is selected for prediction based on the branch class. The classifying predictor can use a shared history table for all component predictors (as shown in Figure 65) or several separate tables.

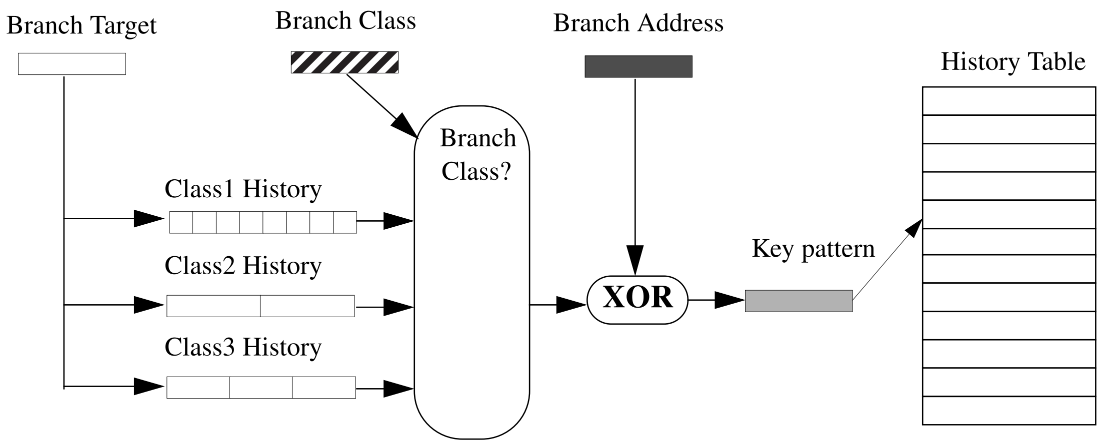


Figure 65. Classifying hybrid predictor with shared history table

First, we study classification based on a branch's source-level origin (indirect call, virtual call, or switch statement). Then we classify branches based on the number of different targets encountered in a program run (the *arity* of a branch).

9.2.1 Opcode-based classification

In a first experiment, we classified branches into three classes: switch, virtual call, and indirect call. The GNU compiler translates switch statements with more than seven cases into an indirect jump; no other C/C++ constructs use an indirect jump[1], and thus this class is easy to recognize. To recognize *virtual function calls*, we used EEL [89] to look for the characteristic five-instruction sequence generated by gcc and marked the corresponding call instruction accord-

[1] They use jump and link (JMPL) instead of jump (JMP), in order to return to the call point.

ingly. All other indirect calls were classified as indirect calls (i.e., calls through function pointers).[1]

			OO			C		
Class	Source structure	Detection	dyn%	stat%	#targets	dyn%	stat%	#targets
switch	switch statements, >7 cases	by opcode (JMP in SPARC)	22.5	2.6	12.7	31.4	35.6	4.9
virtual	virtual function calls in C++	with EEL (5-instr. sequence)	61.2	69.4	2.1	N/A	N/A	N/A
indirect	all other indirect branches	by opcode (JMP and link)	16.3	28.0	2.2	68.6	64.4	5.1

Table 19. Indirect branch classes with dynamic and static frequency, and number of targets

For each branch class, Table 19 shows the average dynamic frequency, static frequency, and average number of targets. Not surprisingly, virtual function calls dominate in OO programs (61% of all branches) whereas indirect calls dominate in C programs (69%). The branch classes differ substantially; in particular, switch branches have a much higher number of targets than the other branches. We hope to exploit this difference by fitting a monopredictor to each class and combining these in a hybrid predictor.

We first determined the path length that results in the lowest misprediction rate for each branch class. The best path length varies according to table size and associativity, since longer paths require larger tables to be effective (see Section 8). Figure 66 shows AVG misprediction rates per branch class and path length for a 1K-entry, 4-way associative table.

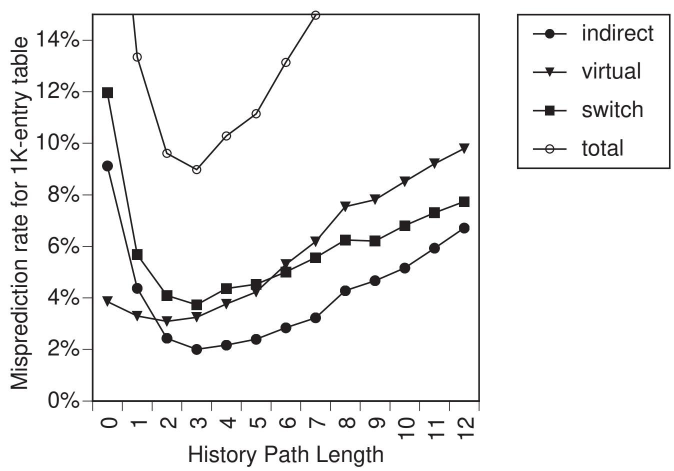


Figure 66. Misprediction rates per opcode-based branch class
for a 1K-entry 4-way associative history table
path lengths 0 (BTB) to 12

Switch branches behave similarly to *indirect* branches even though they have more targets; both classes reach minimal prediction rates at path length three. *Virtual* branches differ from the

[1] Obviously, the distinction between virtual and indirect in hardware would require the introduction of a new opcode or special register usage conventions.

other classes in that their ideal path length is slightly shorter, i.e. they correlate with more recently executed branches.

After finding the best path length for each branch class separately, we calculated the misprediction rate of a classifying predictor using separate tables for each class. For a given total table size T, one of the component predictors uses a table of T/2 entries, the other two components use T/4 entries[1]. From the three possible configurations we selected the one with the lowest miss rate. We then used the best path lengths for separate component predictors of size T, 2T and 4T, to run a simulation of a shared-table classifying predictor of size T.

Table 20 and Figure 67 show the resulting predictor performances in comparison with a non-classifying monopredictor of equal size. Path lengths for the indirect and switch branch class are nearly identical over all table sizes. Virtual branches are better predicted by shorter path lengths.

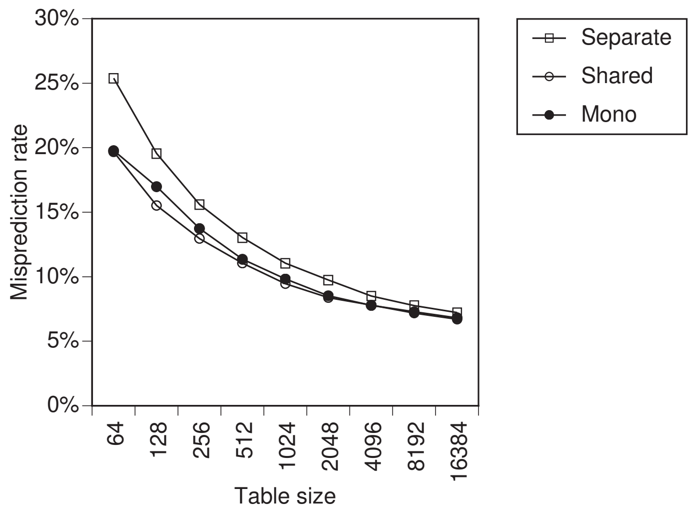


Figure 67. Misprediction rates for opcode-based classifying predictors

Clearly, classifying predictors with separate tables don't perform well, having uniformly higher misprediction rates than a monopredictors. These predictors do not utilize their table space efficiently since the relative frequency of the different branch classes varies widely across benchmarks. For instance, C programs do not have virtual branches, so at least a quarter of the total table space is unused. A shared-table classifying predictor performs better, obtaining a uniformly lower misprediction rate than a monopredictor of equivalent size. This result demonstrates that per-class path length determination does pay off, although the gain is quite small so that the higher hardware cost of opcode-based prediction does not seem justified.

---

[1] In these and later experiments, the total number of table entries is a power of two, in order to make point-to-point comparisons with two-level monopredictors.

Table size	Separate Hybrid			Shared Hybrid		Mono	
	Class with T/2	P I.V.S	miss %	P I.V.S	miss %	P	miss %
64	Indirect	2.0.2	25.4	2.0.2	19.7	1	19.8
128	Indirect	2.0.2	19.5	2.0.2	15.5	1	17.0
256	Switch	2.0.2	15.6	2.0.2	13.0	2	13.7
512	Indirect	2.0.2	13.0	3.1.2	11.0	2	11.3
1024	Indirect	3.1.2	11.0	3.2.3	9.5	3	9.8
2048	Switch	3.1.3	9.7	3.2.3	8.4	3	8.5
4096	Switch	3.2.3	8.5	4.2.3	7.8	3	7.8
8192	Indirect	4.2.3	7.8	4.3.4	7.2	4	7.3
16384	Switch	4.3.4	7.2	5.3.5	6.7	5	6.8

Table 20. Misprediction rates for separate and shared opcode-based classifying predictors. Also shown are the branch class with the largest table (for classifying predictors with separate tables), and the best path length combinations (indirect/virtual/switch) for opcode-based classifying hybrid predictors.

9.2.2 Arity-based classification

In a second experiment, we classified branches according to the number of different targets encountered in a program run, or branch *arity*. The *arity* of a branch can be determined in a profiling run, or estimated by source code analysis. Annotation of this arity in the branch opcode would allow a classifying hybrid predictor to function in the same manner as the opcode-based classifying predictor shown earlier.

After some experimentation, we chose to form three classes (one target, two targets, and more than two targets). Branches with only one target (*monomorphic* branches) constitute 67% of all branches but are executed only 34% of the time. 18% of all branches jump to two targets, for 17% of all branch executions. Branches with three or more targets constitute 15% of all branches but are executed 49% of the time.

Figure 68 shows AVG misprediction rates per branch class and path length. Monomorphic branches are perfectly predicted by a monopredictor without history, i.e., a BTB. Longer path lengths increase the number of mispredictions since every different path leading to the branch causes an extra cold-start miss. Also, each monomorphic branch may occupy multiple table entries, thus increasing capacity misses. Branches with two targets have an optimal path length of two and cause few mispredictions. The bulk of mispredictions comes from branches with three or more targets, with optimal path length three.

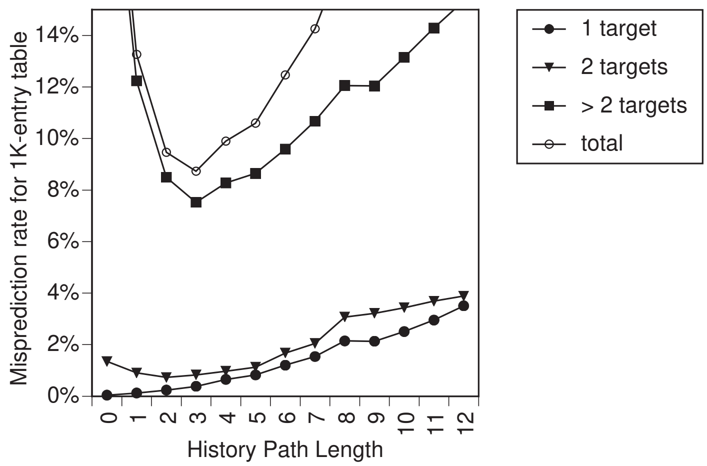


Figure 68. Misprediction rates per arity-based branch class for a 1K-entry 4-way associative history table path lengths 0 (BTB) to 12

Table 21 shows the best arity-based predictor per table size for both separate and shared table predictors. The best path length for monomorphic branches is always zero, but path lengths for the other branches increase with table size. In a component predictor with separate tables, the largest table always goes to polymorphic branches, and the best path length for these branches is nearly always the same as the best path length for a monopredictor.

Table size	Separate Hybrid			Shared Hybrid		Mono	
	Class T/2	P 1.2.>2	miss%	P 1.2.>2	miss%	P	miss%
64	> 2 targets	0.0.1	23.3	0.0.2	18.8	1	19.8
128	> 2 targets	0.0.2	17.3	0.0.2	14.7	1	17.0
256	> 2 targets	0.0.2	13.5	0.0.2	12.1	2	13.7
512	> 2 targets	0.1.2	11.3	0.1.2	10.5	2	11.3
1024	> 2 targets	0.2.3	9.7	0.2.3	8.9	3	9.8
2048	> 2 targets	0.2.3	8.4	0.2.3	8.0	3	8.5
4096	> 2 targets	0.2.3	7.8	0.2.4	7.4	3	7.8
8192	> 2 targets	0.2.4	7.3	0.2.4	6.9	4	7.3
16384	> 2 targets	0.3.5	6.9	0.3.5	6.5	5	6.8

Table 21. Misprediction rates for separate and shared arity-based classifying predictors.

Also shown are the branch class with the largest table (for classifying predictors with separate tables), and the best path length combinations (indirect/virtual/switch) for opcode-based classifying hybrid predictors.

Figure 69 illustrates that the misprediction rate of a separate component hybrid is nearly the same as that of a monopredictor. These predictors benefit from shorter path lengths which generate fewer different patterns per branch, so that fewer table entries are used for branches of arity 1 and 2. On the other hand, polymorphic branches are restricted to use half of the total table size. Both effects appear to nearly cancel each other out, as shown in Figure 69. With a shared table, the latter restriction is removed, and the misprediction rate is always lower than that of a monopredictor.

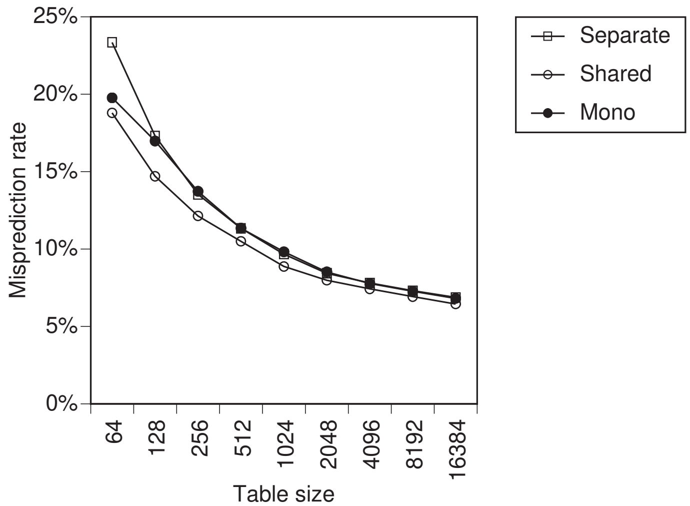


Figure 69. Misprediction rates for arity-based classifying predictors

Although an arity-based classifying hybrid predictor performs significantly better than a monopredictor, especially for smaller tables, it has significant practical disadvantages:

- Branch instructions need to be annotated with an arity counter, requiring an instruction set extension.
- Arity must be determined by profiling, which may not remain accurate over different program runs, or program analysis, which may not be precise enough to lower the overall misprediction rate.

For these reasons, arity-based classification may not be practical. However, its performance shows that classification on the number of targets is more promising than opcode-based classification. In particular, arity-based classification helps reduce the load on a path-based predictor by removing monomorphic branches. This effect significantly reduces capacity misses: for example, for *jhm* the 256-entry predictor of a separate-table classifying predictor of size 512 experiences a table miss rate of only 0.9% versus 2.3% for a 256-entry monopredictor. To exploit this effect, we will use a form of dynamic classification in Section 9.4. Dynamic classification is more adaptive and does not require changes to the instruction set architecture

9.2.3 Discussion

Classifying predictors match branches to predictor components in a round-about way: first the branch is assigned to a class, and then a predictor component is tuned (by varying path length) to a branch class. Much depends on the uniformity of branches within a class. If classes are too coarsely defined, as in the opcode classification experiment, then there is no class-specific behavior to take advantage of. In the arity-based classification experiment, the per-class behavior is much more distinct, allowing a closer match between a branch and a predictor component. However, some branches may still be misclassified. A duomorphic branch may jump to the same target most of the time[1], and could therefore benefit from classification into the monomorphic class. The classification mechanism could take this into account by classifying branches according to the number of times they undergo a target change. Many such alternative classification criteria can be conceived. However, a branch whose behavior (and thus its classification) varies according to program input, will be misclassified in some program runs. Ideally, a branch should be classified according to its current behavior, i.e. if the behavior varies between program runs or between program phases in the same run, then the branch should be re-classified. Dynamic classification at run time can deliver more accurate metaprediction (and thus prediction), because it uses more up to date information.

In the next section, we cut out the classification step and match a branch directly to a component, based on its run-time behavior.

[1] For instance, if it tests for exceptions or end conditions in a loop

9.3 Dual-path hybrid prediction

In this section we combine two monopredictors with different path length into a hybrid predictor that adjusts its meta prediction dynamically. A given branch is always predicted by *both* components. The success of a component in predicting the branch in the past then determines whether its target prediction will be chosen for the current branch.

9.3.1 Meta prediction

We attach a “confidence” counter to each table entry to keep track of the number of times the table entry predicted the correct target. The counter is a n-bit saturating counter which tracks the success rate over the last 2^{n-1} times the entry was consulted. (Replacing an entry resets the counter to zero). The hybrid predictor selects the target with the highest confidence value; ties are resolved using a fixed ordering (we test different orders in the next section).

This metaprediction scheme is a natural generalization of the McFarling Branch Predictor Selection Table [98]. A BTB (path length 0) per-branch confidence information, like a McFarling table. A two-level predictor with path length n stores a confidence for each *path* of length n rather than each branch, so it is more fine-grained. A branch has a confidence attached to each path of length n that leads up to it. This could lead to higher prediction accuracy, since some paths to a branch may be very predictable, some not, and the misses of one path could penalize the confidence in another if they shared a confidence counter. From an implementation point of view, per-entry counters add two bits to all tables, but do not require an extra, separate table. Looking up the pattern and its confidence requires only one table access. A McFarling table requires two table accesses: the per-branch confidence table and the per-path prediction table.

We tested 1,2,3 and 4-bit confidence counters for all configurations in the next section. Although the performance difference between 2,3 and 4 bit counters was small, 2-bit counters usually performed best and are used for all results shown.

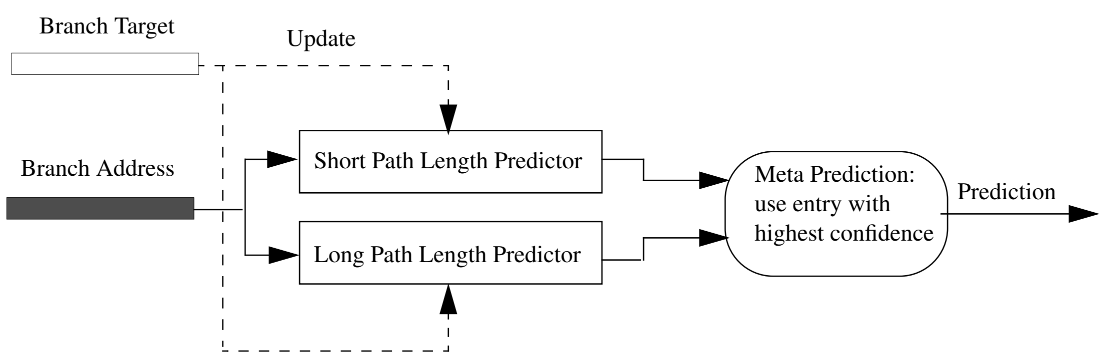


Figure 70. Dual-path hybrid predictor scheme

9.3.2 Component predictors

We simulate hybrid predictors with two component predictors of equal table size and associativity but different path lengths[1]. The component table sizes vary from 32 entries to 16K entries, and we simulate all combinations of path lengths in the range 0..12.

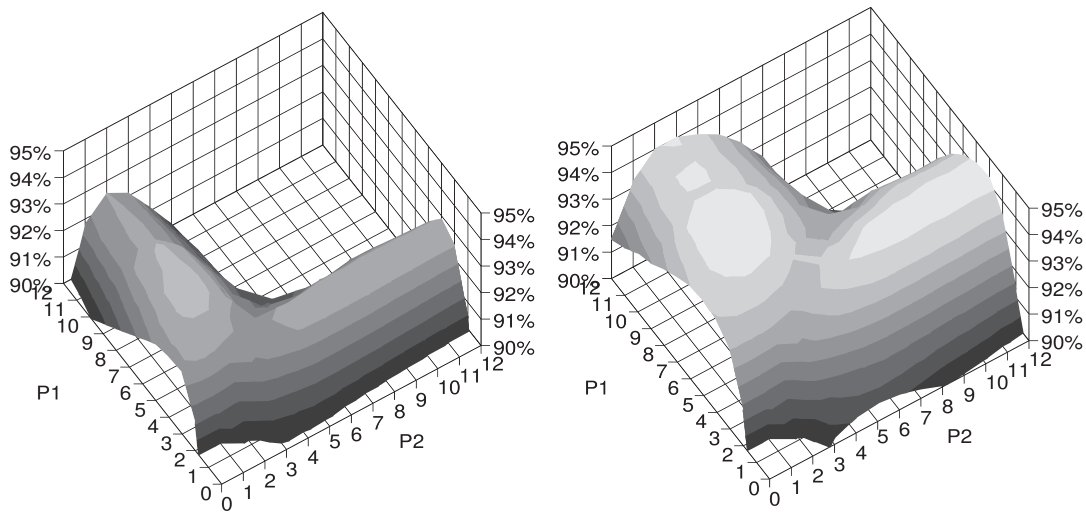


Figure 71. Prediction hit rates for dual-path hybrid predictors, all path length combinations. 4-way associative tables with 2-bit confidence counters and component table size of 2048(left) and 8192(right) entries. P1 is the path length of the first component predictor, P2 of the second. When P1 = P2 (the diagonal), the hit rate for a non-hybrid predictor of twice the component size is shown.

Figure 71 shows the AVG hit ratios for two component table sizes with representative behavior (2048 and 8192); more details are given in Table 22. The best hit rates are obtained by the combination of a short path length predictor (*p*=1..3) with a longer path length predictor (*p*=5..12). Since the curve is fairly symmetrical with respect to the diagonal, it appears that the order of the predictors (which is used to break ties in component predictor selection) does not matter much. For smaller tables, the curve is sharper and peaks at shorter path lengths, i.e., it the choice of the short path length component is more important, and very short path lengths do much better.

[1] Chronologically, this study was performed before the branch classification study. Therefore we still test different associativities. As this section shows, a hybrid predictor suffers from conflict misses to the same extent as its component predictors do. In later studies we only consider 4-way associative tables in order to remove most of the noise of conflict misses, and to cut down on the number of different configurations to be run through our simulation engine.

9.3.3 Results

Figure 72 shows the misprediction rates of the best non-hybrid and hybrid predictors for each table size and associativity. We compare predictors based on total table size, i.e., we treat a hybrid predictor with two component predictors of size N as a predictor of size 2N and compare it against the non-hybrid predictor of that size. In all but one case (64 entry, associativity 4), hybrid predictors obtain lower misprediction rates than equal-sized non-hybrid predictors, even though each component separately suffers more from capacity and conflict misses than the non-hybrid predictor. For smaller table sizes (between 64 and 512 entries), the effect of increased associativity remains stronger than that of hybridization. For example, a non-hybrid 4-way associative table of size 256 achieves a lower misprediction rate than a hybrid predictor with two 2-way associative components of size 128 each. For larger table sizes (between 1K and 32K entries), a hybrid predictor with 2-way associative components performs better than a non-hybrid 4-way associative predictor of the same size. For 2- and 4-way associative non-hybrid predictors with tables larger than 2K entries, the prediction rate improves more by changing to a hybrid predictor than by doubling the total table size. For tables larger than 4K entries, a 4-way associative hybrid predictor outperforms even a fully-associative table of the same size.

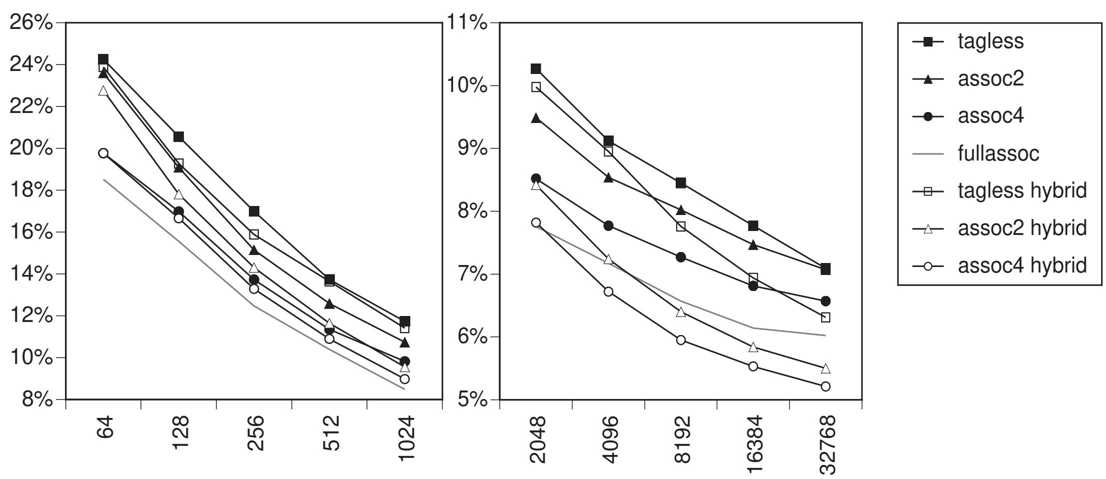


Figure 72. Misprediction rates per table size, associativity and predictor type

Best predictor (choice of path length) is chosen according to AVG
Tagless, two-way, four-way and fully associative tables.
Hybrid and non-hybrid versions
Table size given in total number of entries

Table 22 shows the misprediction rate of the best predictor for each table size as well as the component path lengths for which the misprediction rate was achieved. The trend towards longer path lengths with increasing table size is very pronounced; clearly, long paths are ineffective for small predictor tables. We also show the misprediction rate and path length per class for the shared-table arity-classifying predictor from Section 9.2.2 (with table associativity four), the most accurate classifying predictor. For total table entry sizes of 1K entries or smaller, classifying prediction wins, while for larger tables, dual-path hybrid prediction performs better.

size	tagless		assoc2		assoc4		Arity-classification	
	miss%	p1.p2	miss%	p1.p2	miss%	p1.p2	miss%	P 1.2.>2
64	23.89%	0.2	22.76%	1.0	19.77%	1[a]	18.80	0.0.2
128	19.28%	1.4	17.81%	1.4	16.66%	2.0	14.70%	0.0.2
256	15.89%	1.3	14.31%	2.1	13.29%	2.0	12.14%	0.0.2
512	13.64%	3.1	11.65%	3.1	10.90%	3.1	10.50%	0.1.2
1024	11.42%	3.1	9.56%	3.1	8.98%	3.1	8.87%	0.2.3
2048	9.98%	3.1	8.42%	4.1	7.82%	5.1	7.98%	0.2.3
4096	8.95%	3.7	7.24%	5.2	6.72%	6.2	7.42%	0.2.4
8192	7.76%	3.7	6.40%	6.2	5.95%	6.2	6.92%	0.2.4
16384	6.94%	3.9	5.84%	7.2	5.53%	7.2	6.45%	0.3.5
32768	6.31%	3.9	5.50%	7.2	5.21%	8.2	6.28%	0.3.5

Table 22. Misprediction rates and path lengths for dual-path hybrid predictors

[a] A non-hybrid predictor outperforms all hybrid predictors in this case.

A dual-path hybrid predictor stores two different path length target predictions for each branch, while a classifying predictor uses only one path length for each branch, reducing the amount of storage required (reducing the number of capacity misses). For small tables, this effect seems more important than the capability of a dual-path hybrid predictor to adapt its metaprediction to individual branches. In the next section, we will study a hybrid predictor that combines both these aspects: it employs dynamic metaprediction, but does not require a branch to be predicted by all the predictor components.

9.4 Cascaded prediction

A *cascaded predictor* classifies branches dynamically by observing their performance on a simple first-stage predictor. Only when this predictor fails to predict a branch correctly is a more powerful second-stage predictor permitted to store predictions for newly encountered history patterns of that branch. By preventing easily predicted branches to occupy history table space in the second-stage predictor, the first-stage predictor functions as a *filter*. By filtering out easily predicted branches, the first-stage prevents them from overloading the second-stage table, thereby increasing its effective capacity and overall prediction performance. As a result, the second-stage predictor's history table space is used to predict only those branches that actually need more sophisticated and costly prediction.

9.4.1 Metaprediction

Figure 73 shows the prediction and update scheme for a cascaded predictor. If both predictors have a prediction, the second-stage predictor takes precedence. If the second stage has no prediction (a table miss), then the first stage prediction is used. Therefore, the second-stage predictor's history table must have tagged entries, so that table misses can be detected.

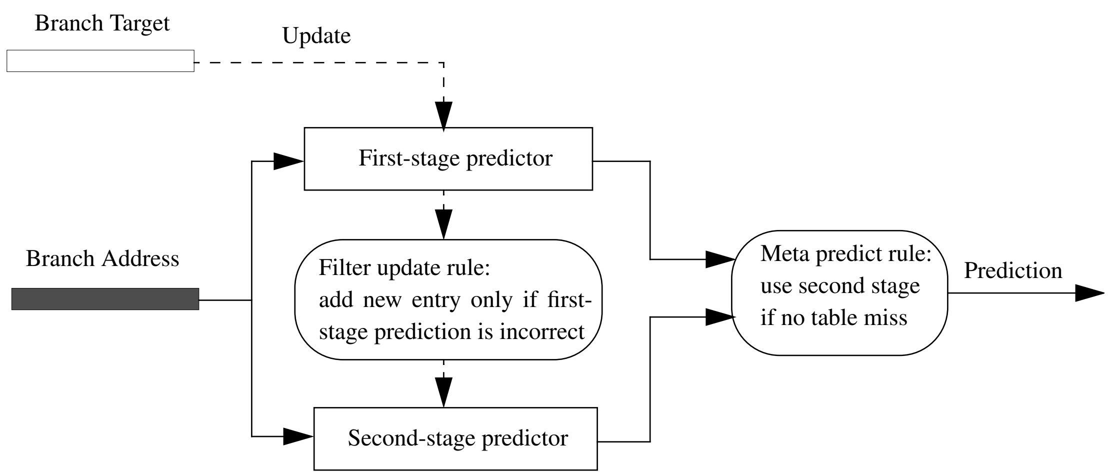


Figure 73. Cascaded predictor scheme

This metaprediction rule by itself already increases prediction accuracy compared to dual-path hybrid prediction, as shown in later sections. If the first stage is a short path length monopredictor, and the second a long path monopredictor, then it always seems to pay to trust the longer one. In a sense, the longer path match is a more specialized case of the short path match. We still need the short path length predictor while the long path length predictor is warming up, since the latter suffers more from cold start misses. In a cascaded predictor, the short path predictor also serves a second purpose: it functions as a filter for table *updates*.

9.4.1.1 Table update filtering

If the first-stage predictor predicts a branch correctly, we do not allow the second-stage predictor to create a *new* entry for the branch. Only if the first stage predictor misses do we enter the pattern and its target in the second stage table. Old entries in the second stage are always updated. Since they have been entered before, there was at least one case in which the first stage mispredicted the branch. In contrast, the first-stage predictor is always allowed to update its table.

Update filtering has a precursor in conditional branch prediction: Chang, Evers and Patt propose to use the BTB to detect heavily biased branches [22]: a saturating counter keeps track of the number of times that a branch goes into the biased direction. If the counter overflows, then history patterns for the branch are no longer updated in a separate Pattern History Table (PHT). The difference with the filtering rule we employ (explained in more detail in the next section) is subtle: we assume a branch is biased (monomorphic) until misprediction in a BTB proves otherwise, Chang, Evers and Patt assume it is non-biased until proven otherwise, i.e. until enough successful predictions occur in the BTB to overflow the counter. For heavily biased branches, that always go to the same target, our scheme avoids the storage of new patterns from the beginning[1]. Using a bias counter does not prevent the storage of new patterns until the counter overflows. After the bias is detected, the two filtering rules behave the same way.

9.4.2 Cascaded prediction with a BTB as first stage

To simplify our analysis, we first perform experiments in which the first stage is a BTB (0 path length monopredictor), and vary its size as well as the path length and type of the second stage component. This allows us to precisely determine the optimal table update rule, which is then employed in the more general experiments of Section 9.4.3.

A BTB does not employ history. It stores at most one entry per branch, which suffices for monomorphic branches. Non-monomorphic branches are better predicted by a longer path length predictor, as shown in section 9.2.2, and such branches will advance to the second stage of the predictor cascade.

We examined two variants of cascading predictors. Predictors with a *strict filter* only allow branches into the second-stage predictor if the first-stage predictor mispredicts (but not if it misses). In other words, branches only advance to the second stage if they are provably non-monomorphic. In contrast, a *leaky filter* also allows new second-stage entries on first-stage misses. Thus, the second-stage table may contain entries for monomorphic branches, but these are likely to be evicted by patterns from branches that actually need two-level prediction.

[1] Except for the very first pattern occurring for a branch, if leaky filtering is employed.

9.4.2.1 Strict filters

Successful prediction in the filter classifies a branch as monomorphic (if only temporarily). If a branch misses in the filter predictor, nothing conclusive is known: every branch incurs a compulsory table miss in the filter, even if it is purely monomorphic. To prevent cold-start misses of the filter to pass through to the second-stage predictor, *strict filtering* disallows new entry insertion in the second-stage predictor on a miss (but not on a mispredict). To implement this strict filter design, the filter's table must be tagged.

Figure 74 shows the misprediction rates for three selected second-stage predictors (each with optimal path length for its size). We also show the misprediction rate of a cascaded predictor without filtering. Even without filtering, the first-stage predictor reduces overall misprediction rates compared to the stand-alone predictor (shown as filter size 0) by providing an educated guess in the case of a table miss in the second-stage predictor. In other words, a cascaded predictor consisting of a BTB and a path-based predictor reduces cold-start misses even without filtering. We call this a *staged* predictor (see Section 9.4.3)

Strict filters do not perform well for small filter table sizes. For 16 entries or less, overall misprediction rates are even higher than that of the stand-alone predictor. Essentially, a strict filter predictor recognizes branches as polymorphic only if the branch remains in the filter table long enough to incur a target change (and thus a misprediction). But with small filters, many polymorphic branches are displaced from the filter before they mispredict, and thus they never enter the second stage. When the filter becomes large enough to avoid most capacity misses, this effect disappears and filtering starts to pay off. At 256 entries, strict filtering performs as intended, i.e., it prevents monomorphic branches from overloading the second-stage predictor, resulting in lower misprediction rates than those of a non-filtering cascaded predictor.

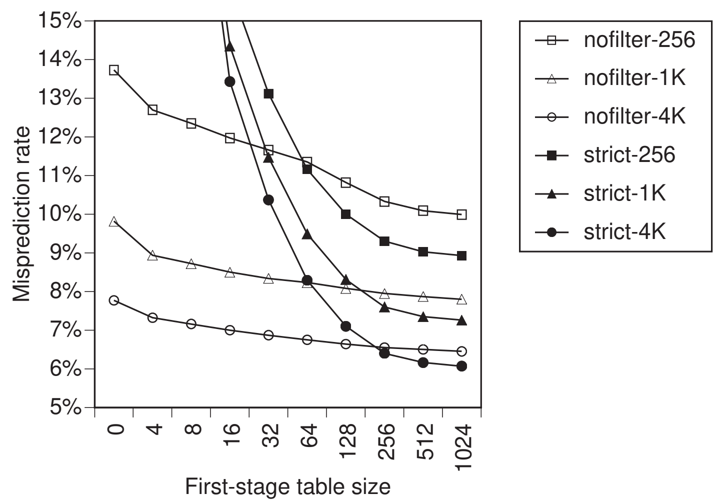


Figure 74. Mispredication rates for a cascaded predictor with strict filter

Second-stage predictor table size is 256 1K and 4K entries; both predictor tables are 4-way associative. Also shown is a cascaded predictor without filtering.

9.4.2.2 Leaky filters

The sensitivity of strict filters to capacity misses is a serious flaw; the performance of a filtered predictor should remain at least as good as that of a stand-alone second-stage predictor. To prevent filter capacity miss problems, a *leaky filter* inserts an entry into both predictors upon a first-stage table miss. That is, only correctly predicted branches are stopped by the filter. Thus, every branch is introduced at least once into the second-stage predictor, but filtering still occurs for later executions of the same branch: as long as the branch remains in the filter table and doesn't mispredict, no further second-stage entries will be permitted. If the load on the second-stage predictor table is high, the cold-start entries for monomorphic branches will eventually be displaced by entries for polymorphic branches. Leaky filters are cheaper to implement than strict filters: since a misprediction and a table miss is treated the same way (new entry in second-stage table), the filtering predictor can use a tagless table[1]. The second-stage predictor still needs tags in order to recognize a table miss.

Figure 75 shows the performance of leaky filters. Even for very small filters, the filtering effect is pronounced and improves misprediction rates compared to a non-filtering cascaded predictor. For example, a 32-entry BTB filter improves the misprediction rate of a 256-entry monopredictor from 11.7% to 10.7%. Filtering still helps even with very large (4K) predictors, reducing

[1] For easy comparison, all experiments are performed with 4-way associative, tagged tables.

mispredictions by about 0.5%. For large filters of 256 entries or more, the leaky filter's misprediction rate is only slightly better than that of a strict filter (not shown in the figure).

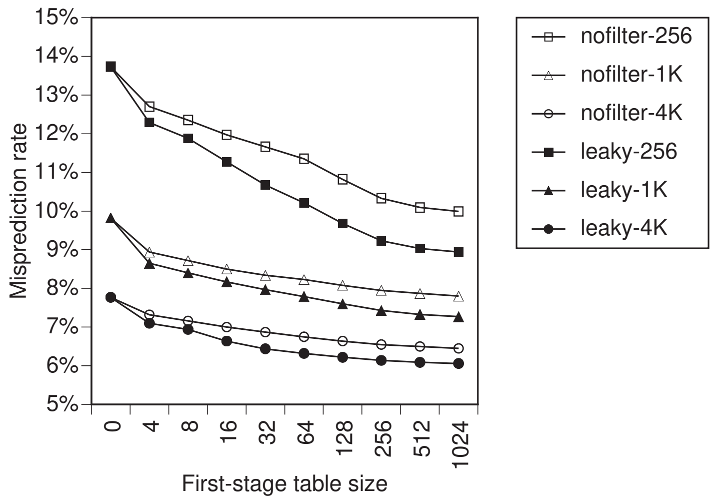


Figure 75. Misprediction rates for a cascaded predictor with leaky filter

Second-stage predictor table size is 256, 1K and 4K entries; both predictor tables are 4-way associative. Also shown is a cascaded predictor without filtering.

9.4.2.3 Results

Second stage table size	mono	Best P per filter table size								mono	Miss% per filter table size								hybrid
		4	8	16	32	64	128	256	512		4	8	16	32	64	128	256	512	
64	1	2	2	2	2	2	2	2	2	19.8	18.6	17.4	15.8	14.7	13.7	12.7	12.0	11.7	19.8
128	1	2	2	2	2	3	3	3	3	17.0	14.8	14.1	13.3	12.7	11.9	11.1	10.5	10.3	16.7
256	2	2	2	3	3	3	3	3	3	13.7	12.3	11.9	11.3	10.7	10.2	9.7	9.2	9.0	13.3
512	2	3	3	3	3	3	3	3	3	11.3	10.2	9.7	9.3	8.9	8.7	8.4	8.1	8.0	10.9
1024	3	3	3	3	4	4	4	5	5	9.8	8.6	8.4	8.2	8.0	7.8	7.6	7.4	7.3	9.0
2048	3	4	4	4	5	6	6	6	6	8.5	7.8	7.6	7.4	7.2	7.0	6.9	6.7	6.7	7.8
4096	3	4	4	6	6	6	6	6	6	7.8	7.1	6.9	6.6	6.4	6.3	6.2	6.1	6.1	6.7
8192	4	5	6	6	6	6	6	6	6	7.3	6.5	6.3	6.1	6.0	5.9	5.8	5.8	5.7	6.0
16384	5	6	6	6	6	6	8	8	8	6.8	6.2	6.0	5.8	5.7	5.7	5.6	5.6	5.5	5.5

Table 23. Path length and misprediction rate for second-stage monopredictors

For comparison, the table also shows the best monopredictor ("mono") and the best dual-path hybrid predictor ("hybrid") of equivalent size.

Table 23 shows the best path length and misprediction rate for cascaded predictors with second stage monopredictors. Even very small BTB filters increase the effective capacity of the second-stage table, allowing it to accommodate longer paths without incurring extensive capacity misses. For example, a 16-entry filter reduces the misprediction rate for all second-stage sizes to below that of a monopredictor with twice the number of entries and uses the path length of a monopredictor that is four times larger. A 64-entry filter lowers the misprediction rate below that of a monopredictor four times as large, and for all table sizes smaller than 1K entries, beyond that of a dual-path hybrid predictor of twice the size.

We also studied cascaded predictors that use dual-path hybrid predictors in the second stage, anticipating that filtering would again reduce second-stage misses and allow longer path lengths. For each filtered hybrid of table size T, we simulated the best path length couples of non-cascaded dual-path hybrid predictors of section 9.3 with table sizes T, 2T and 4T.

Second level table size	hybrid	Best P per filter table size								hybrid	Miss% per filter table size							
		4	8	16	32	64	128	256	512		4	8	16	32	64	128	256	512
64	1.1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	19.8	20.6	19.3	18.3	17.5	15.8	14.7	13.9	13.6
128	2.0	2.0	3.1	3.1	3.1	3.1	3.1	3.1	3.1	16.7	15.5	14.9	13.9	13.1	12.4	11.4	10.7	10.4
256	2.0	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1	13.3	12.3	11.8	11.3	10.9	10.5	9.9	9.3	9.1
512	3.1	3.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1	10.9	10.0	9.6	9.3	9.0	8.8	8.5	8.1	8.0
1024	3.1	5.1	5.1	6.2	6.2	6.2	6.2	6.2	6.2	9.0	8.2	8.0	7.7	7.5	7.3	7.1	6.9	6.8
2048	5.1	6.2	6.2	6.2	6.2	6.2	6.2	6.2	6.2	7.8	7.1	6.9	6.7	6.5	6.5	6.4	6.3	6.2
4096	6.2	6.2	7.2	7.2	7.2	7.2	7.2	7.2	7.2	6.7	6.3	6.1	6.0	5.9	5.9	5.8	5.8	5.7
8192	6.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	6.0	5.6	5.5	5.5	5.4	5.4	5.4	5.3	5.3
16384	7.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	8.2	5.5	5.3	5.2	5.2	5.2	5.1	5.1	5.1	5.1

Table 24. Path length and misprediction rate for second-stage dual-path hybrid predictors

For comparison, the table also shows the best the best dual-path hybrid predictor ("hybrid") of equivalent size.

The resulting improvements were equally pronounced (Table 24). Again, filtering reduces the misprediction rate and increases the best path length choices for each table size. Whereas non-cascaded predictors need at least 1024 table entries to achieve misprediction rates below 9%, a filtered dual-path hybrid attains this threshold with only 544 entries (32-entry filter BTB plus 512-entry dual-path hybrid). Adding an 8-entry filter to a 1K-entry hybrid predictor lowers its misprediction rate from 9.0% to 8.2%. A 128-entry filter reduces this to 7.5%. Across all table sizes, cascaded prediction reduces table size by roughly a factor of two.

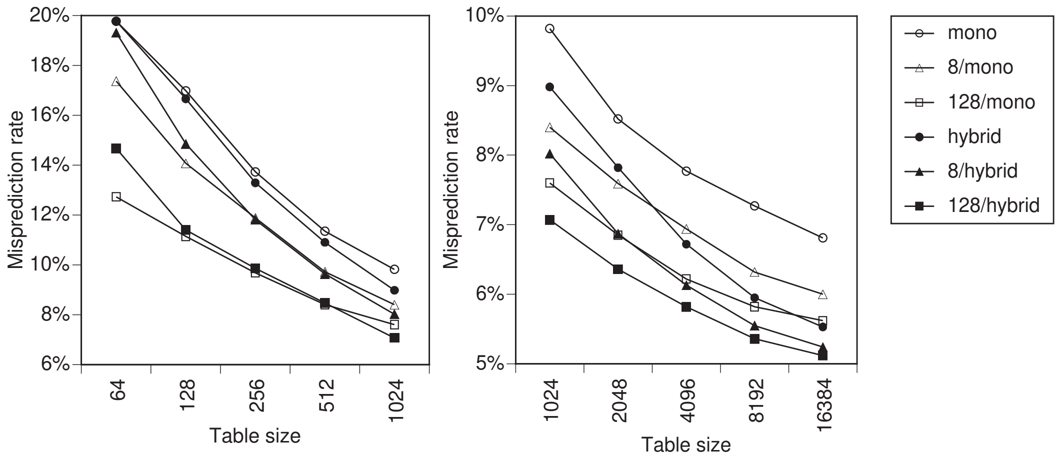


Figure 76. AVG misprediction rates without filters and with 8 and 128-entry filters. Second stage predictors are mono and dual-path length hybrid predictors We use different scales for small and large table sizes to increase visibility.

Figure 76 shows AVG misprediction rates for selected prediction schemes. For all schemes and table sizes, filtering reduces misprediction rates. For tables of 512 entries or less, the resulting misprediction rate of filtered monopredictors is equal or lower than that of filtered dual-path hybrid predictors. This is due to a filter's cold-start miss reduction (it usually has a zero path length prediction if the second stage has not yet encountered the longer path). A dual-path hybrid predictor does not get extra benefit from this effect at small table sizes, since its shorter component's path length is already close or equal to zero. Most of the misprediction rate reduction of small hybrid predictors is therefore due solely to the filter's capacity miss reduction. Mono predictors, with path lengths of two or three, benefit both from capacity miss reduction and cold-start miss reduction. The resulting misprediction rates end up being fairly similar for mono and hybrid predictors. However, for second-stage tables of 1K entries and larger, the filtered hybrid predictor's misprediction rate is substantially lower than that of a filtered monopredictor at all filter sizes. Since the short component path length is two in most of these cases, a filter's cold-start miss reduction also benefits a dual-path length hybrid predictor. At the high end of the table size range (+8K), conflict and capacity misses become less frequent, and the benefit of filtering starts to diminish.

Figures 77,78 and 79 show misprediction rates for the s*elf*, *edg* and *gcc* benchmarks[1]. The former two programs have the largest number of active branches in the benchmark suite. The reduction in misprediction rate is higher than the reductions on AVG of Figure 76, which shows that the benefit of filtering is especially pronounced for large programs.

[1] Best mono predictor path lengths are determined separately for each benchmark. Hybrid predictor path lengths are picked from the AVG path length choices.

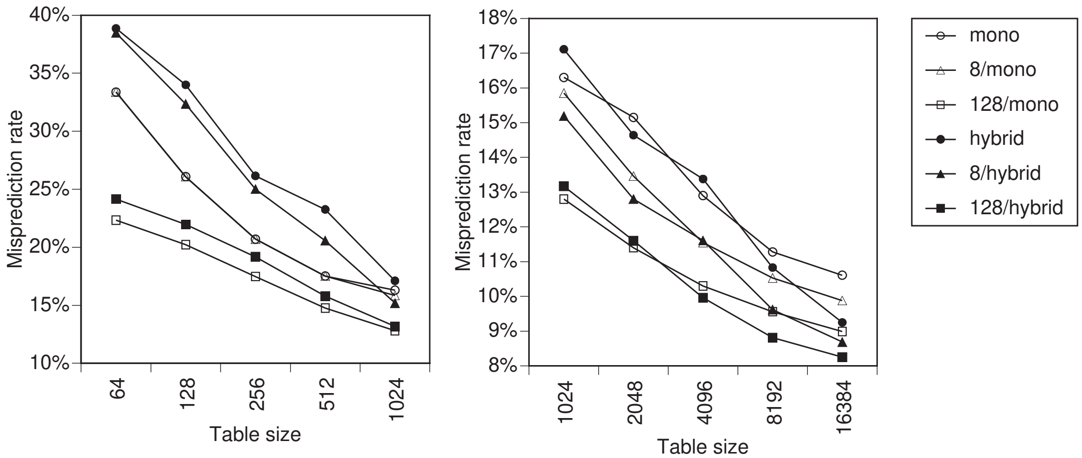


Figure 77. *Self* misprediction rates without filters and with 8 and 128-entry filters

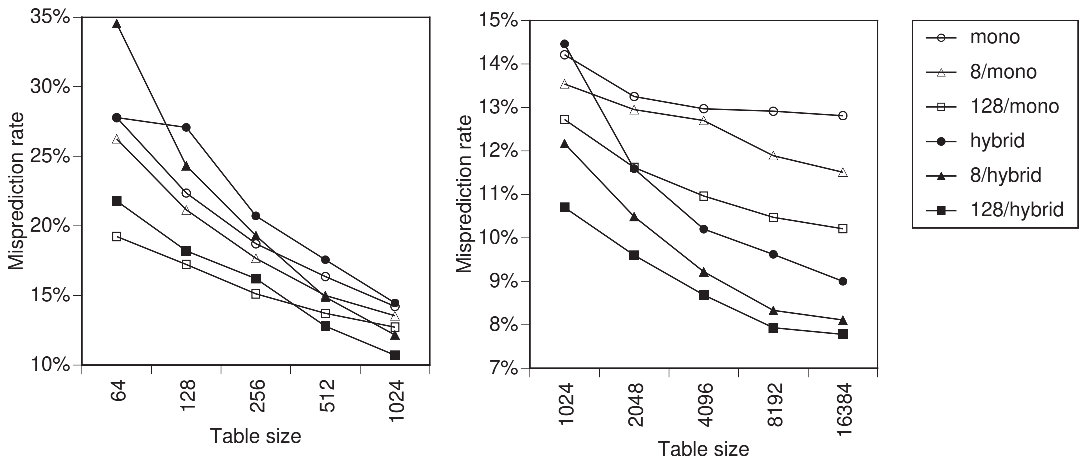


Figure 78. *Edg* misprediction rates without filters and with 8 and 128-entry filters

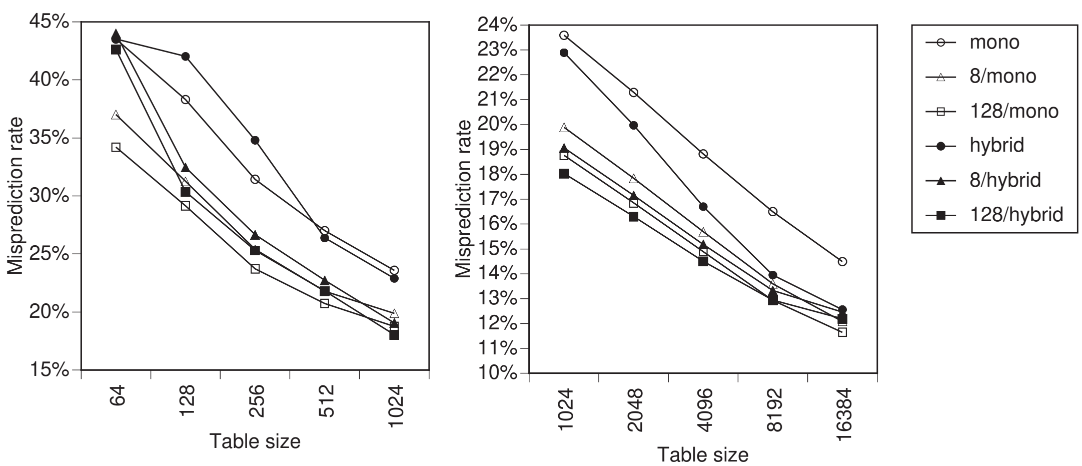


Figure 79. *Gcc* misprediction rates without filters and with 8 and 128-entry filters

9.4.3 Multi-stage cascaded prediction

In the previous section we determined the optimal metaprediction and update filtering rule for a cascaded predictor, on the simplified case where the first stage predictor is a history-less (0 path length) BTB. Here we investigate the accuracy of a natural generalization of this two-stage predictor by allowing any type of predictor in the first stage and any number of stages. First, we use the maximum number of stages, and unlimited, fully associative tables for each stage, to determine the limit of prediction accuracy reachable by this architecture. Then we test two and three stage predictors for a wide range of table sizes, to investigate how close we can get to the limit with a practical architecture.

9.4.3.1 Predictor components

Figure 80 shows representative examples of the predictors used in this study. The simplest architecture is a *branch target buffer* (BTB). A selection of bits from the branch address serves as a key pattern into a predictor table, which stores the last target observed for this branch.

A *two-level* predictor extends the BTB scheme by taking bits from the last p branch targets preceding the execution of the current branch and xor-ing these bits with the branch address. The parameter p is the *path length* of the two-level predictor.

A *cascaded* predictor consists of several stages, each containing a two-level predictor with its own history buffer and predictor table. Successive stages use increasing path lengths (in preliminary experiments we observed the consistently inferior accuracy of decreasing path lengths). The use of separate tables allows all stages to predict in parallel. In a final step, the predictor chooses the prediction from the last stage that did not encounter a table miss. This ensures that its target prediction is based on the longest available path history.

A cascaded predictor saves table space by using a *leaky filter* update rule (see Section 9.4.2): a new history pattern enters a long path length stage only if none of the shorter stages predicted the branch correctly. This rule prevents easily predicted branches from occupying table space in an expensive, long path length stage. For example, branches with only a single target are perfectly predicted by a BTB, after the initial compulsory miss, so they do not need a long history pattern. As a result, the longer path length stage encounters fewer capacity misses, improving overall prediction accuracy.

We also measure the accuracy of a cascaded predictor without filtering. We call this a *staged* predictor. Staged predictors, even without filtering, improve prediction accuracy compared to a two-level predictor because they reduce cold start misses. Longer path length two-level predictors are more accurate than short path length predictors, but they need a longer time to reach that potential since they store more patterns per branch. In a staged predictor, the early stages predict many branches accurately while the later stages are warming up. This also reduces the *effect* of capacity misses in later stages, since an earlier stage is likely to have a (less accurate) prediction available in the case of a late stage capacity miss.

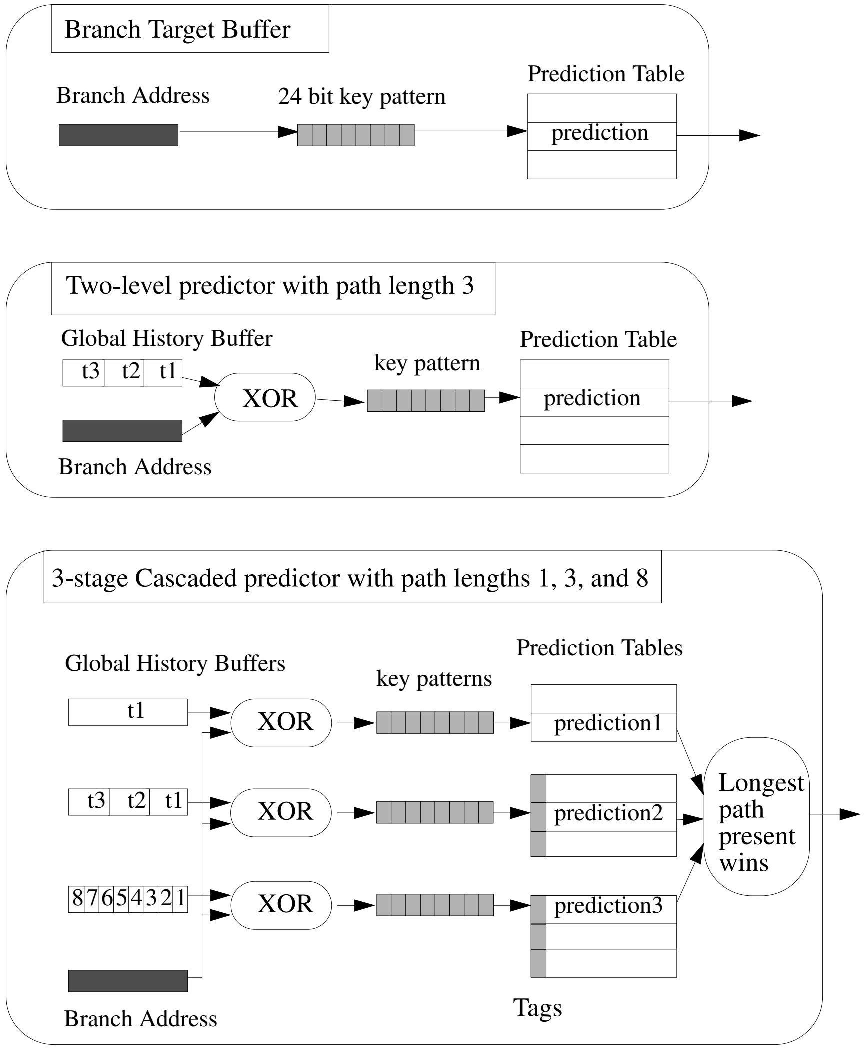


Figure 80. Examples of a branch target buffer, two-level predictor and cascaded predictor.

A staged predictor looks the same as a cascaded predictor, but has a different update rule (every stage is updated, where a cascaded predictor prevents insertion of new patterns in later stages if an earlier stage predicts a branch correctly).

In the next section we investigate predictor accuracy under ideal circumstances, in the absence of table interference (conflict misses) and capacity misses.

9.4.3.2 Ideal predictors

We use the term *ideal* for a predictor scheme with an unlimited, fully associative predictor table. The misprediction rate measured for such a predictor is free from the noise of conflict and capacity misses. This allows us to compare the intrinsic strength of different prediction schemes, and provides us with a limit beyond which prediction cannot be improved merely by spending more transistors on predictor tables.

In one respect, the predictors studied in this section are not ideal: they have limited 24-bit history buffers. History buffers must be kept small because they determine the cost of a table entry through the size of the tag. A small buffer represents each target with few bits, and this can cause pattern interference, reducing prediction accuracy. However, a 24-bit history buffer suffices for near-ideal accuracy. Beyond path length 12, a 24-bit history buffer causes extensive pattern interference because it stores only one bit for some targets. Therefore we do not use path lengths longer than 12. This restriction does not substantially reduce the potential accuracy, as we found during a preliminary experiment with a 30-bit buffer and path lengths up to 15.

Ideal cascaded predictors have a full complement of stages. For example, an ideal cascaded predictor of path length 6 has 7 stages, consisting of ideal two-level predictors with path lengths 0 (a BTB) up to 6. Similarly, and ideal *staged* predictor also has the maximum number of stages, but does not employ pattern filtering.

Terminology	Description
Unlimited ideal two-level of path length P	Two-level predictor with an unlimited history buffer, storing a concatenation of full precision addresses of the P most recent targets, with an unlimited, fully associative predictor table
Ideal two-level of path length P	Two-level predictor with 24-bit history buffer, storing the xor of (24 div P) bits of the P most recent targets, with an unlimited, fully associative predictor table
Ideal branch target buffer (BTB)	Ideal two-level predictor of path length 0
Ideal staged of path length P	A non-filtering staged predictor, with P+1 stages, consisting of ideal two-level predictors of path length 0,1,..,P
Ideal cascaded of path length P	An ideal staged predictor of path length P, with filtering of new patterns

Table 25. Ideal predictor terminology

Figure 81 shows misprediction rates for ideal two-level predictors, cascaded predictors and staged predictors for path lengths from 0 to 12. Two-level prediction reaches a minimum misprediction rate of 6.0% at path length 6. Longer path lengths show increasing misprediction rates, because cold start misses start to negate the advantage of capturing longer-term correlations. The loss of accuracy due to pattern interference is small, as shown by the accuracy of an ideal two-level predictor with unlimited buffer size (see Section 8). Where path lengths are integer factors of the buffer size, enabling full buffer use, interference is negligible (path lengths 0 to 4, 6, 8 and 12). Only for path lengths that use less than the full history buffer does the misprediction rate increase noticeably. For example, path length 9 in a 24 -bit history buffer uses only two bits per target, for a total of 18 bits[1]. The difference in accuracy, represented by

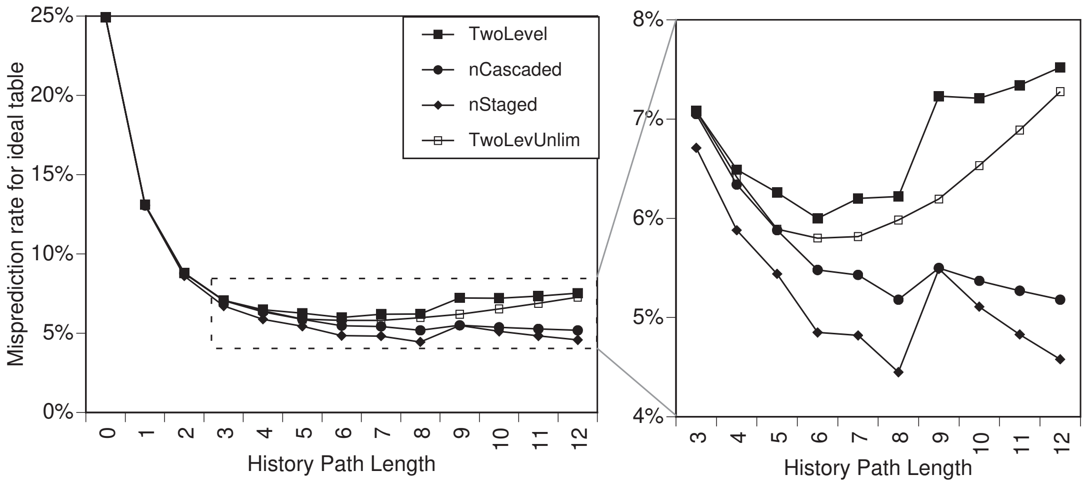


Figure 81. Ideal two-level, fully cascaded and fully staged predictor misprediction rates
For path lengths 0 to 12 (cutout enlarged in right graph)

the "bump" in the curves, is strictly due to pattern interference. The curves for cascaded prediction and staged prediction show similar bumps, the most prominent at path length 9.

An ideal cascaded predictor is better than ideal two-level prediction at all path lengths. The short path length stages in a cascaded (and a staged) predictor immediately start predicting many branches accurately while the later stages are warming up, as explained in Section 9.4.3.1. This gives it an advantage over a two-level predictor of the same path length even in the absence of capacity misses.

Staged prediction reaches lower misprediction rates than cascaded prediction at all path lengths. This is to be expected, since a cascaded predictor uses filtering and therefore stores strictly less information than a staged predictor. Cascaded prediction merely economizes on the number of table entries required by staged prediction, and this has no beneficial effect for ideal predictors with unlimited tables.There simply are no capacity misses to reduce.

However, the reduction of table entry cost is dramatic, as shown in Figure 82. The graph shows the total number of table entries occupied in a two-level, cascaded and staged predictor. As the path lengths grows, a staged predictor's size grows exponentially, while a cascaded and two-level predictor show nearly linear growth. At path length 12, a staged predictor occupies 173325 entries, five times as much as a cascaded predictor (34572) and a two-level predictor (34076). The curves of cascaded and two-level prediction overlap almost completely, suggesting some systematic effect.We currently can not explain this similarity, but suspect that the exact reason for the close resemblance is to be found in information theory. In terms of

[1] It is of course possible to fill the unused bit space with bits from a strict subset of the target addresses. This opens up a variety of choices as to which targets should be represented with 1 bit extra accuracy. In the reverse interleaved address projection employed, this choice is unclear. To simplify our analysis, we left these bits unused.

branch prediction schemes, it means that a two-level predictor table cost is in the same ball park as a fully cascaded predictor. However, the latter eliminates cold start misses, while preserving the capacity to capture longer term correlations. In the next section we measure these benefits in the context of practical predictors.

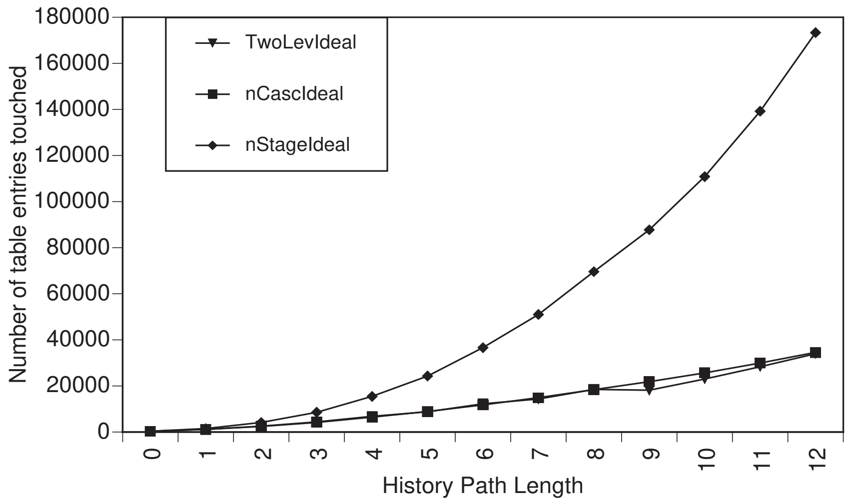


Figure 82. Number of patterns stored by an ideal two-level, cascaded and staged predictor

9.4.3.3 Practical predictors

In this section we study practical cascaded predictor architectures with limited, 4-way associative tables and a small number of stages. Our aim is to reduce the number of table entries required to attain a given prediction accuracy. We also want to find out how close we can get to the prediction accuracy of the hypothetical ideal predictors of the previous section.

9.4.3.3.1 Practical predictor tables

The main cost of predictor architectures lies in the amount of on-chip memory required to store predictions. In the previous section we saw that the total number of patterns generated by an ideal predictor grows as its path length increases. For example, a two-level predictor reaches a minimal misprediction rate of 5.8% at path length 6, by storing13210 pattern/target associations (averaged over the AVG benchmarks). Given a table entry size of about 60 bits (24-bit tag, 1-bit update counter, and 32-bit target address), the resulting data structure takes up about 800 K bits of memory, barely within the capability of current processor technology. We want to reduce this memory cost while keeping misprediction rates as low as possible.

Reducing the size of a predictor table generates *capacity* misses: a pattern/target association, though it was stored, and would have correctly predicted the target of a branch, was evicted from the table by a pattern/target of a more recently executed branch. For smaller tables, the path length must be shortened to prevent extensive capacity misses. However, shorter path length predictors are less accurate. For every given table size, there is some path length which forms the optimal compromise between these opposing effects. We use simulation to determine this optimal path length, and the resulting misprediction rate, for table sizes from 32 to 32K entries.

One further limitation is necessary to allow limited size predictor tables to be implemented in silicon: a limited associativity (see Section 9). We use tables with associativity four, as usual.

9.4.3.3.2 Practical multi-stage predictors

A staged predictor must split up a given total table entry budget and allocate some part of it to each stage. Although it is conceivable to use one global predictor table for all stages, this would require an expensive multi-access table. Therefore each stage uses a separate single-access table. For a given total table entry budget, we have the option of using many stages with small tables, or few stages with large tables. Which one is better?

For a cascaded predictor, there are two opposite effects at work. On the one hand, partitioning table space over a large number of stages increases the likelihood that a benchmark's working set overloads the capacity of any given stage, resulting in extensive capacity misses. This also reduces the efficiency of pattern filtering, since only correct predictions prevent patterns from occupying space in later stages. On the other hand, a large number of stages allows filtering to occur at a stage with a shorter path length, reducing the total number of patterns stored. For our benchmark suite, a small number of stages uses a given table entry budget more effectively than a full complement of stages. For example, a cascaded predictor with 9 stages, from path length 0 up to 8, with 128 entries per stage (total budget of 1152 entries) achieves a misprediction rate of 7.7%, while a 2-stage cascaded predictor with 512 entries per stage (total budget of 1024), and path lengths 1 and 8, reaches 6.8%.

We therefore focus on predictors with a small number of stages. From an analysis point of view, their superior accuracy is a windfall, since a small number of stages substantially reduces the number of different path length combinations to be simulated[1]. For 2-stage predictors without filtering (staged), we test all path length combinations P1P2, with 0 <= P1 < P2 <= 12, for each total table entry budget that is a power of 2 between 32 and 32K entries. Each stage uses half of the total table size.

We also determine optimal path length combinations for cascaded predictors. For the 2-stage cascaded predictor, we test the same range of path lengths as for the 2-stage non-filtering predictor. However, we also simulate table entry budgets halfway between powers of 2 (in order to compare with 3-stage predictors with same-size stages). In that case, two third of the entries is allocated to the last stage.

[1] An truly exhaustive study would require simulation of approximately #stages ^ (#tablesizes * #historylengths). For the range of table sizes, history lengths and number of stages explored in this study, this would lead to 15 ^ (10 * 15) ~= 2.6E176 different combinations, far too many to explore within the given time frame.

For the 3-stage predictor we test all path length combinations with 0 <= P1 < P2 < P3 <= 12, with P1 <= 4, and P2 <= 10, to reduce the number of combinations without removing good candidates. We simulate the same range of total table entry budgets as for the 2-stage cascaded predictor by using stage sizes X+X+X and X+X+2X, with X equal to a power of 2. Table 26 summarizes the terminology.

Terminology	Description
Two-level of size T	Two-level predictor with a 4-way associative predictor table of size T
2-staged of size T	A non-filtering staged predictor, using 2 two-level predictors of size T/2
2-staged cascaded of size T	A 2-staged predictor of size T, with pattern filtering. If T is not a power of 2, the stages have size T/3 and 2T/3
3-staged cascaded of size T	A 3-staged predictor with pattern filtering, using 3 two-level predictors of size T/3. If T is a power of 2, the stages have size T/4,T/4, and T/2

Table 26. Practical predictor configurations (path length combinations are tuned to each size T)

9.4.3.4 Results

Figure 83 shows misprediction rates that result from using the best path length combinations for each predictor configuration. Exact numbers with path length combinations can be found in Table 28. For comparison we also show the misprediction rate of a BTB and the best ideal two-level, cascaded and staged predictors of the previous section (shown as dotted lines parallel to the x-axis since table size is not a factor).

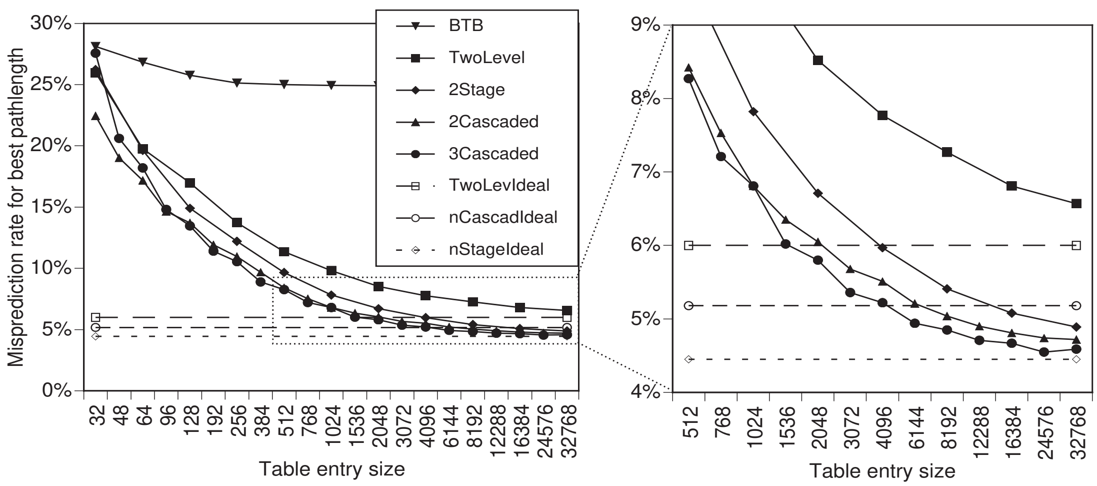


Figure 83. Misprediction rates for practical two-level, staged and cascaded predictors

Cascaded predictors perform better than two-level predictors at all table sizes in the explored range. In other words, any given misprediction rate is bought at a much lower cost. For instance, a 3-stage predictor of size 512 outperforms a two-level predictor of size 2K. At a fairly modest budget of 1.5K entries, a 3-staged cascaded predictor attains the same misprediction rate as an ideal two-level predictor with an unlimited table.

3-stage cascaded predictors consistently outperform 2-stage predictors, and 2-stage cascaded predictors outperform 2-stage (non-filtering) predictors. For limited budgets, cascaded prediction wins over staged prediction because pattern filtering reduces capacity misses, allowing a cascaded predictor to use longer path lengths for any given table size.

2-and 3-staged predictors seem to get asymptotically close to the accuracy of an ideal staged predictor for large, but still practical table budgets. A small number of stages clearly suffices to get almost arbitrarily close to ideal accuracy. A 3-stage cascaded predictor, at 4K table entries and higher, even improves upon the accuracy of an ideal cascaded predictor with a full complement of stages. This surprising result indicates that a small number of stages improves prediction accuracy. The label "ideal" for a fully staged cascaded predictor thus seems to be a misnomer. We are not entirely sure why this is the case. Filtering seems to be responsible for the reduced accuracy, since a non-filtering ideal staged predictor still outperforms all other predictors. A small number of stages does not filter out as many patterns, and therefore it may represent an intermediate between these extremes, at large table budgets[1], resulting in an intermediate misprediction rate.

Do 3-stage cascaded predictors form the best possible configuration for a limited budget? We think not. Dividing the table entry budget over 8 stages gives worse performance than a 2-stage cascaded predictor, as shown above, so the optimal number of stages may lie somewhere in between 3 and 8, most likely at the lower end. However, the benefit of extra stages is not likely to improve performance by a large factor. The difference between 2 and 3 stage cascaded prediction is already fairly small, allowing a 3-stage predictor of size 3X to reach the same or better performance than a 2-stage predictor of size 4X, as soon as X is larger than 512. The added complexity of extra stages may not be worth the effort.

We did not show staged predictors in which stages differ in size by a large degree, although we did simulate 2-and 3stage predictors with the later stage(s) as much as a factor 32 larger than the earlier one(s). The results indicate that predictors with a table budget spread out evenly among (a small number of) stages perform best. This effect is already noticeable for the 3-stage predictors shown here, since the curve in Figure 83 shows slight bumps for those configurations with total table budgets that are powers of two. In other words, a 3-stage predictor with stages of size X+X+X performs relatively better, in accuracy per table entry, than a 3-stage predictor with stage sizes X+X+2X.

[1] Positive interference could also be responsible. However, the tables are tagged, so this can only be pattern interference.

9.4.3.5 Detailed data

Table 27 shows misprediction rates and occupied table entries for ideal predictors with unlimited tables, for various path lengths.

	Table entries				Misprediction rates				
Path length	BTB	Twolevel	Cascaded	Staged	BTB	Twolevel	Cascaded	Staged	Unlimited Twoleve
0	356	356	356	356	24.9	24.9	24.9	24.9	24.9
1	356	1195	1191	1551	24.9	13.1	13.1	13.1	13.1
2	356	2617	2495	4169	24.9	8.8	8.8	8.6	8.8
3	356	4477	4266	8645	24.9	7.1	7.0	6.7	7.1
4	356	6863	6469	15508	24.9	6.5	6.3	5.9	6.4
5	356	8838	8940	24346	24.9	6.3	5.9	5.4	5.9
6	356	12324	11838	36670	24.9	6.0	5.5	4.8	5.8
7	356	14379	14935	51048	24.9	6.2	5.4	4.8	5.8
8	356	18580	18430	69629	24.9	6.2	5.2	4.5	6.0
9	356	18160	21914	87788	24.9	7.2	5.5	5.5	6.2
10	356	23061	25770	110849	24.9	7.2	5.4	5.1	6.5
11	356	28400	30003	139249	24.9	7.3	5.3	4.8	6.9
12	356	34076	34572	173325	24.9	7.5	5.2	4.6	7.3

Table 27. Total number of entries and misprediction rates for ideal BTB, two-level, fully cascaded and fully staged predictors, and for an ideal two-level predictor with unlimited history buffer

Table 28 shows misprediction rates of practical predictors and the best path length combination at various table budgets.

	Misprediction rate in%					Tuned path length combinations			
Total table size	BTB	Twolevel	2stage	2cascaded	3cascaded	Twolevel	2stage	2cascaded	3cascaded
32	28.1	26.0	26.2	22.4	27.6	1	0.2	0.2	0.1.2
48				19.0	20.6			0.2	0.2.3
64	26.8	19.8	19.6	17.2	18.2	1	0.2	0.2	0.1.3
96				14.7	14.8			0.2	0.2.5
128	25.8	17.0	14.9	13.7	13.5	1	0.2	0.2	0.1.3
192				11.9	11.4			0.3	0.2.8
256	25.1	13.7	12.2	11.0	10.6	2	0.2	1.6	0.1.5
384				9.7	8.9			0.3	0.2.8
512	25.0	11.3	9.7	8.4	8.3	2	1.5	1.8	0.2.8
768				7.5	7.2			1.6	0.2.8
1024	24.9	9.8	7.8	6.8	6.8	3	1.6	1.8	0.2.8
1536				6.4	6.0			1.6	1.4.12
2048	24.9	8.5	6.7	6.0	5.8	3	1.6	1.8	1.3.12
3072				5.7	5.4			1.8	1.4.12
4096	24.9	7.8	6.0	5.5	5.2	3	2.8	1.8	1.4.12
6144				5.2	4.9			1.8	1.4.12
8192	24.9	7.3	5.4	5.0	4.8	4	2.8	2.12	1.4.12
12288				4.9	4.7			2.12	1.6.12
16384	24.9	6.8	5.1	4.8	4.7	5	2.8	2.12	1.6.12
24576				4.7	4.6			2.12	3.4.12
32768	24.9	6.6	4.9	4.7	4.6	5	2.8	2.12	1.6.12

Table 28. Misprediction rates of a BTB, Twolevel, 2-stage, 2-stage cascaded and 3-stage cascaded predictor and their best path length combinations (a BTB's path length is 0), for various table entry budgets.

Table 29 shows misprediction rates per benchmark for selected predictor schemes. Headings represent predictor schemes. For example, *8Staged* is an 8-staged nonfiltering staged predictor, *3Casc.P0.2.8* is a 3-staged cascaded predictor with path lengths 0, 2, and 8 (the best combination for a total of 1K-entries, in three 4-way associative tables). *Oracle12* is a 12-stage nonfiltering staged predictor with a perfect metaprediction rule: instead of picking the component prediction with the longest path length, the *correct* prediction is chosen, if any component predicts the branch correctly. *3CascPerBench* is the misprediction rate reached by a 3-staged cascaded predictor (1K-entry in this case), which is tuned specifically to each benchmark (instead of choosing path lengths which minimize AVG misprediction rates).

Name	instr. / indirect	cond. / indirect	active branches		Ideal					1.5K 4-way assoc		1K-4way assoc				
			99 %	100 %	Oracle12	8Staged	8Cascaded	P6TwoLevelIdeal	BTB	3Casc.P1.4.12	2Casc.P1.6	3CascPerBench	3Casc.P0.2.8	2Casc.P1.8	TwoLevel.P3	BTB
idl	47	6	70	543	0.2	0.3	0.4	0.4	2.4	0.3	0.4	0.3	0.4	0.4	0.8	2.4
jhm	47	5	34	155	2.1	8.1	8.6	8.7	11.1	8.7	9.7	8.6	9.9	9.6	11.3	11.1
self	56	7	848	1855	3.1	7.4	8.0	10.1	15.7	12.2	12.9	12.2	12.4	13.9	18.6	16.3
xlisp	69	11	4	13	0.3	1.4	1.6	1.4	13.5	1.9	1.7	1.5	1.8	2.0	2.2	13.5
troff	90	13	61	161	3.9	6.7	6.8	7.1	13.7	7.7	7.3	7.2	7.9	7.8	7.8	13.7
lcom	97	10	87	328	0.4	0.8	0.9	1.4	4.2	1.2	1.2	1.3	1.3	1.4	2.2	4.2
AVG-100	68	9	184	509	1.7	4.1	4.4	4.8	10.1	5.3	5.5	5.2	5.6	5.9	7.2	10.2
perl	113	17	7	24	0.2	0.2	0.2	0.5	31.8	0.2	0.2	0.2	0.2	0.2	0.3	31.8
porky	138	19	89	285	1.6	4.2	4.3	4.5	20.8	4.6	4.7	5.0	6.2	5.0	7.9	20.8
ixx	139	18	91	203	1.5	3.5	4.0	5.6	45.7	3.3	4.5	3.3	4.2	4.0	9.0	45.7
edg	149	23	186	350	3.4	7.2	8.1	11.9	35.9	8.9	9.6	9.4	9.6	9.4	16.3	35.9
eqn	159	25	58	114	2.8	8.5	10.8	12.5	34.8	12.8	13.1	14.4	14.8	14.9	21.9	34.8
gcc	176	31	95	166	2.6	8.2	11.9	11.7	65.7	14.8	15.1	15.3	17.3	17.2	23.6	65.7
beta	188	23	135	376	0.6	1.3	1.5	2.3	28.6	1.7	2.1	2.6	2.6	2.6	5.6	28.6
AVG-200	152	22	94	217	1.8	4.7	5.8	7.0	37.6	6.6	7.1	7.2	7.8	7.6	12.1	37.6
AVG	113	16	136	352	1.7	4.5	5.2	6.0	24.9	6.0	6.4	6.3	6.8	6.8	9.8	25.0
AVG-OO	107	14	164	447	1.8	4.5	5.0	5.8	19.7	5.8	6.2	6.1	6.6	6.6	9.5	19.7
AVG-C	127	21	73	138	1.6	4.3	5.5	6.3	36.7	6.5	6.7	6.6	7.2	7.2	10.6	36.7
m88ksim	1827	233	5	17	1.0	5.7	6.7	3.1	76.4	3.0	3.0	2.0	7.7	5.8	14.4	76.4
vortex	3480	525	10	37	2.4	4.3	6.3	9.9	20.2	7.7	9.2	4.5	6.4	4.6	7.1	20.2
ijpeg	5770	441	7	60	0.2	0.2	0.2	0.6	1.3	0.3	0.3	0.2	0.2	0.3	0.5	1.3
go	56355	7123	5	14	4.7	23.2	23.8	22.8	29.2	21.7	22.0	20.6	21.9	23.0	21.6	29.2
AVG-infreq	16858	2081	7	32	2.1	8.4	9.3	9.1	31.8	8.2	8.6	6.8	9.1	8.4	10.9	31.8

Table 29. Misprediction rates per benchmark for selected predictor schemes

9.4.4 Conclusions

We have studied a new hybrid predictor architecture, the cascaded predictor, on a trace of purely indirect branches. We first tested different table update rules and measure prediction accuracy on the special case of a two-stage cascaded predictor with a BTB as first stage. Then we studied multi-stage cascaded prediction. Cascaded prediction delivers superior accuracy even in the absence of resource constraints, by exceeding the accuracy reached by any other predictor scheme previously tested on these traces. In the context of limited table entry budgets, cascaded prediction also provides superior accuracy, this time by reducing the cost of two-level prediction by a factor of four or more.

More specifically:

- Ideal cascaded prediction with unlimited, fully associative tables reaches a hit rate of 94.8%. Ideal staged prediction, without pattern filtering, reaches 95.5%. We believe this accuracy is close to the limit of predictability, using a pure indirect branch history, of the indirect branches in our benchmark suite.

- Cascaded predictors with a small number of stages closely approach this limit when using large but practical table entry budgets. In particular, a 4K entry, 3-stage cascaded predictor attains 94.8% accuracy.

- At every table entry budget from 32 to 32K entries, multi-staged cascaded prediction delivers accuracy superior to two-level prediction. In particular, a 512-entry three-stage cascaded predictor reaches 92% accuracy, reducing table size by a factor of four compared to a two-level predictor. At 6K entries, a 3-stage cascaded predictor with tuned path lengths reaches 95% prediction accuracy, higher than the 94% accuracy achieved by a hypothetical two-level predictor with an unlimited, fully associative predictor table.

We believe that cascaded prediction can also improve conditional branch prediction and load value prediction, because these applications suffer equally from cold start and capacity misses, and because recent related work [52] (see Section 10) shows that a similar architecture delivers superior accuracy on conditional branches. It seems to be an idea whose time has come.

9.5 Summary

We used two-level predictors of different path length as components in a hybrid predictor. We studied three classes of hybrid predictors, whose main distinctive feature is the way they decide which component is used for a particular branch, and if there are several target predictions, which one to trust (metaprediction):

- *Classifying predictors*, first proposed for conditional branches by Chang, Hao and Patt [23], assign a class to each branch according to compile-time or profile-based criteria. We assigned to each branch class a separate component predictor, with a path length tuned to the branch class. *Opcode-based* classification, where an indirect branch is classified as a switch, virtual or indirect branch, does not perform well. *Arity-based* classification, where an indirect branch is classified according to the number of different targets (one, two, or more than two), performs well, but requires a change in the instruction set architecture. The best of the tested predictors achieves 91% prediction accuracy for 1K table entries.

- *Dual-path hybrid predictors* predict each branch with both components and use the target prediction with the highest confidence counter. This dynamically updated counter keeps track of the number of successes among the most recent predictions of a table entry. The fine-grained metaprediction implemented by per-path confidence counters increases prediction accuracy compared to classifying predictors, but this only pays off with large enough table entry budgets. Since every branch is always updated in both components, dual-path predictors need more table entries for the same working set of branches. After tuning, the best predictor achieves 91% misprediction rate for 1K total table entries.

- *Cascaded predictors*, a new prediction architecture, use several stages of two-level predictors of increasing path length, and give precedence to the longest path length prediction available. This already improves prediction accuracy compared to dual-path hybrid predictors with confidence counters. In addition, cascaded prediction does not require that every branch is predicted by all components. If a short path length component predicts a branch correctly, the later stages are prevented from storing the longer path length target prediction. This filtering of new paths reduces capacity misses. The best combination with a BTB as first stage resulted in 92% prediction accuracy for 1K total table entries. A cascaded predictor reduces the cost of a two-level predictor by a factor four, for similar misprediction rates. At 6K entries, a 3-stage cascaded predictor with tuned path lengths reaches 95% prediction accuracy, higher than the 94% accuracy achieved by a hypothetical two-level predictor with an unlimited, fully associative predictor table.

Cascaded prediction works so well because the first-stage predictor reduces the load on the longer path length predictors. Generally speaking, longer branch histories require larger tables. In principle, this effect should occur for any application of path-based predictors where the dynamic frequency of easily predicted cases is high. In particular, conditional branch prediction or load value prediction could behave in qualitatively the same way. Therefore, we believe that cascaded predictors might also perform well in those areas. Of course, only empirical work can confirm this hypothesis, and thus we are planning to explore these questions in future work.

10 Related work

"Our field is still in its embryonic stage. It's great that we haven't been around for 2000 years. We are still at a stage where very, very important results occur in front of our eyes."
Michael O.Rabin [115]

In this section we discuss related work not touched in other chapters.

10.1 Software techniques

10.1.1 Code rewriting optimization

Dispatch overhead can also be reduced by eliminating dispatches (rather than just making them fast). When a branch is provably monomorphic, its target procedure can be inlined. This optimization not only removes the direct overhead of lookup and parameter passing, but also enables classic optimizations such as common subexpression elimination. Many standard optimizations such as interprocedural analysis require a static call graph to work well, and many intraprocedural optimizations are ineffective for the small function bodies present in object-oriented programs. Thus the presence of dynamic dispatch hinders optimization, and consequently, the resulting program will run more slowly. Concrete type inference [106, 130,2, 110, 3] or link-time optimizations [10,58] can determine the concrete receiver types of calls, possibly eliminating dynamic dispatch for many sends.

When a call site has the same target for a sufficiently long time, a dynamic compilation system can detect this through type feedback [70], and recompile code with inlining in much the same way as a static compiler. The SELF-93 system inlines 95% of all dispatches[71] with compiler optimizations such as customization [16] and type feedback. Hölzle and Ungar [70] estimate that the resulting speedup in SELF is five times higher than the direct cost of the eliminated dispatches. Given the dispatch overheads reported here, this ratio suggests significant optimization opportunities for C++ programs. Preliminary results from an optimizing C++ compiler confirm this assumption [6].

However, these techniques are not applicable in all cases. Type-inference requires the whole program to be present at compile time, an unrealistic assumption in the typical environment of dynamically linked libraries. Type-feedback requires dynamic recompilation, or profiling, one of which has a cost at run time, the other requiring most of the program to be tested in a separate phase. The implementation effort associated with either method is typically much larger than that of implementing an efficient dispatch technique in software, an important consideration to practitioners. Even when either of these techniques is applicable, part of the program still needs dynamic dispatch, since an object-oriented program typically contains real run-time polymorphism.

10.1.2 Message dispatch techniques

Rose [112] analyzes dispatch performance for a number of table-based techniques, assuming a RISC architecture and a scalar processor. The analysis included both dispatch and tag checking code sequences. The study considers some architecture-related performance aspects such as the limited range of immediates in instructions. Other studies have analyzed the performance of one or two dispatch sequences. For example, Ungar [127] analyzes the performance of IC, LC, and no caching on SOAR, a RISC-processor designed to run Smalltalk. In [40], we analyze algorithmic issues of a number of dispatch techniques for dynamically-typed languages, but without taking processor architecture into account. Hölzle et al. [70] compare IC and PIC for the SELF system running on a scalar SPARC processor. Milton and Schmidt [102] compare the performance of VTBL-like techniques for Sather. None of these studies takes superscalar processors into account.

Calder et al. [14] discuss branch misprediction penalties for indirect function calls in C++. Their measurements of several C++ programs indicate that inline caching might be effective for many C++ programs (although measurements by Garrett et al. [59] are somewhat less optimistic). For each call site the address of the most frequently called function is determined from execution profiles. Calder et al. propose to improve performance with “if-conversion,” an inline cache with a statically determined target. For their suite of programs (which differs from ours), they measured an average BTB hit ratio of 91%, assuming an infinite BTB. In comparison, the hit ratios we observed were much lower, with a median hit ratio of only 76% for the standard benchmarks. Grove et al. [61] also report more polymorphic C++ programs than Calder. BTB hit ratios vary substantially between individual programs (see Figure 42), and the difference between benchmark suites should therefore not come as a surprise.

The efficiency of message dispatch has long been a concern to implementors of dynamically-typed, pure languages like Smalltalk where dispatches are more frequent since these languages model even basic types like integers or arrays as objects. Dispatch consumed a significant fraction of execution time in early Smalltalk implementations (often 30% or more, even in interpreted systems). Hash tables reduced this overhead to around 5% [30]; however, 5% of a relatively slow interpreter still is a lot of time. The introduction of *inline caching* [35,126] dramatically diminished this overhead. Polymorphic inline caches (PICs), extend the technique to cache multiple targets per call site. For SELF-93 which uses inline caching and polymorphic inline caching [68], Hölzle and Ungar [72] report an average dispatch overhead of 10-15% on a scalar SPARCstation-2 processor, almost half of which (6.4%) is for inlined tag tests implementing generic integer arithmetic. (This figure also includes other inlined type tests, not just dispatched calls.) Given the large differences in languages, implementation techniques, and experimental setup, used, it is difficult to compare these results with those presented here.

Srinivasan and Sweeney[117] measure the number of dispatch instructions in C++ applications, but do not calculate the relative dispatch overhead or consider superscalar issue.

10.1.3 Multiple dispatch techniques

We have studied single dispatch only. Cecil [18] and CLOS [83], among other languages, support multiple-argument polymorphism directly in the form of multi-method dispatch. Ingalls [75] shows how to implement multiple dispatch with a sequence of single dispatches, but such an implementation is unlikely to be optimal (see Section 2.3.3).

Kiczales and Rodriguez [83], describe a multiple dispatch technique where each generic function has a corresponding hash table, mapping the combination of dynamic classes of specialized arguments to a target method.

Dussud [51] uses a tree of hash tables per generic function, each mapping the dynamic class of a specialized argument to a nested subtree of has tables that test the remaining arguments.

Chen and Turau model the multiple dispatch function as a collection of Deterministic Finite Automata (DFA's) [26],[27]. In a thorough analysis, the authors outline an algorithm to construct an automaton *DFAm* for each message *m*, so that dispatch is achieved in *n* transitions, with *n* the number of arguments. Each argument type in turn decides the transition to be executed. The algorithm ensures a minimal number of states for each automaton, so that space requirements are minimized. However, the actual implementation of *DFAm* is not treated. In [26], the suggested implementation for each transition is "a hash table or a binary tree". We do not believe that this is sufficiently fast for a practical system. However, if space is at a premium, this method is very promising. A fast implementation of multiple dispatch stores the DFA as an n-dimensional table, indexed by the n argument types. As in single dispatch, this table is too large to be stored in memory. In the Cecil system, for example, there are 1008 types, so a 3-argument message would have a table with $1008^3 \sim 10^9$ entries. If a memory address takes four bytes, about four gigabytes would be necessary just to store the table of this one message. Since there are on average four definitions for each message, the potential redundancy in this table is enormous.

Amiel, Gruber and Simon compress the aforementioned n-dimensional table by sharing rows between types [7]. If, for instance type *t* and type *u* in the first argument lead to exactly the same dispatch code, for all combinations of types of the other argument, then *t* and *u* are mapped to the same index for the first argument, prior to the actual access into the compressed table.This mapping is implemented by an argument array for each argument of each message. For instance *ARGm1[t] = ARGm1[u]*, for the above case. The argument array has as many entries as there are types in the system. The memory overhead of this technique is difficult to estimate without a quantitative study. The argument arrays occupy a substantial amount of memory, but the space requirements depend mainly on the amount of sharing that can be accomplished on the n-dimensional table. This quantity depends on the content of the table, which is determined by the actual message definitions and thus can not be estimated without looking at real samples.

We propose a multiple dispatch technique that takes advantage of row displacement compression (two-argument cascaded dispatch) [44]. For two-argument dispatch, the two-dimensional table, indexed by the first two argument types, can be mapped into the memory space used by single dispatch tables. Since the number of messages is much larger than the number of types in any given system (in Cecil there is a difference of a factor four), the width of such a table will likely fit into the row displacement master array. Empty space in this table can be reclaimed by fitting in single dispatch tables. For three or more arguments, the entries in the two-dimensional table refer to a dispatch table for the third and subsequent arguments. Since most of the messages in Cecil dispatch over two or fewer arguments, this technique offers efficient dispatch for the majority of cases (see Table 30). we analyze dispatch sequences, for argument array

#Args	#Messages	Messages%
0	671	16.16%
1	3159	76.07%
2	271	6.53%
3	44	1.06%
4	6	0.14%
5	1	0.02%
6	1	0.02%

Table 30. Multiple-argument dispatch in the Cecil system

dispatch and two-argument cascaded dispatch, using the methodology of Section 5. According to this analysis, two-argument cascaded dispatch outperforms argument-array dispatch only for two-argument messages. For three or more arguments, argument-array dispatch wins because the argument table loads are not dependent.

Pang, Holst, Leontiev, and Szafron [108] apply row displacement on multi-dimensional dispatch tables generated by multiple dispatch. They show on the Cecil class library that multiple row displacement is faster and uses less space than the aforementioned techniques.

Chambers and Chen [21] build a Directed Acyclic Graph (DAG) for each multiple dispatch, and prune unreachable paths from this DAG using static information such as type declarations and class hierarchy analysis. In [20], they unify a number of the above techniques (see the discussion in Section 2.3.4), and demonstrate, on a variety of mature class libraries, that a combined approach commonly delivers superior performance to techniques used in isolation.

10.1.4 Dispatch table compression

Besides the techniques presented in Section 3.3, two other methods for compressing dispatch tables are reported on in the literature.

Huang and Chen propose two-way coloring [74], which applies the selector coloring principle to both selectors and classes, and also shares occupied entries. This technique needs to check at run time both the actual class and selector for a "message not understood" error. To our knowledge, it has not been tested on real class libraries.

Thorup presents sparse arrays [122], used to implement Objective-C, which find a method through two indirections. The storage of empty entries is reduced by dividing a dispatch table into chunks of constant size, and not storing those that are completely empty. No fill rates or other comparable data is reported, so we can not compare the memory requirements with our technique. However, the dispatch speed of sparse arrays is lower because of the double memory indirection.

Compression of sparse tables is an old problem. Dencker et. al., in [34], list a number of techniques for the compression of parser tables, which contain large numbers of default entries. For this application, accessing a table entry is time critical, changing the table is not. Therefore the requirements are identical to those of dispatch tables. However, of the techniques presented, only row displacement and selector coloring perform retrieval in constant time. The other techniques, like run length encoding, require search within the compressed table.

We used a genetic algorithm to find near-optimal column and row numbers for arbitrary sparse tables [41]. This experiment showed that better encodings exist than the heuristics employed in class-based row displacement [40]. However, the time required to find these encodings was prohibitive in the context of dispatch table compression (several hours on a Cray Y-MP).

10.2 Hardware techniques

10.2.1 Indirect branch prediction

Lee and Smith [89] describe several forms of BTBs. Jacobson et al. [77] study efficient ways to implement path-based history schemes and observe that BTB hit rates increase substantially when using a global path history. Their Correlated Task Target Buffer (CTTB), unconstrained and fully associative, reached misprediction rates of 18% and 15% for *gcc* and *xlisp* with path length 7; our study found misprediction rates of 12% and 1.5% for p=7. The different results can be explained by several factors: different benchmark version (SPEC92 vs. SPEC95), inputs, and radically different architectures (e.g., the multiscalar processor's history information will likely omit some branches in the immediate past). Finally, Jacobson et al. include conditional branches in the path histories, which is probably responsible for the difference in `xlisp`.

Chang et al. [25] explore a limited range of two-level predictors for indirect branches and simulate the resulting speedups of selected SPECint95 programs for a superscalar processor. The misprediction rate of a BTB-2bc is reduced by half to 30.9% for `gcc` with a Pattern History Tagless Target Cache with configuration gshare(9). This predictor XORs a global 9-bit history of taken/non taken bits from conditional branches with the branch address, and uses the result as a key into a globally shared, tagless 512-entry history table. In the present study, a comparable non-hybrid predictor (p=3, tagless 512-entry) reaches a misprediction ratio of 31.5% for *gcc*, the best non-hybrid predictor (p=2, four-way associative 512-entry) has 28.1% misprediction rate, and the best dual-path hybrid predictor (p1=3, p2=1, four-way associative 512-entry) reaches 26.4%. These comparisons should be regarded with caution, since the two experiments differed in architectures (HPS vs. SPARC), compilers, and benchmark inputs.

Emer and Gloy [54] describe several single-level indirect branch predictors based on combinations of the values of PC, SP, register number, and target address, and evaluate their performance on a subset of the SPECint95 programs. For these programs, the best predictor shown achieved a misprediction ratio of 30%, although the authors allude to a better predictor that achieves 15%.

Calder and Grunwald proposed the two-bit counter update rule for BTB target addresses [14] and showed that it improved the prediction rate of a suite of C++ programs.

Stark, Evers and Patt [118], determine the best path length for each branch in the program by profiling, and communicate this path length to the hardware through the ISA. This is a refinement of branch classification, with 32 branch classes, each corresponding to a particular path length. For a tagless table of 128 entries, a misprediction rate of 27.7% is achieved for gcc. This technique represents the most fine-grained profile-based classification of indirect branches we know of.

Kalamatianos and Kaeli [81] apply PPM prediction (see Section 10.2.2.2) to indirect branches, demonstrating excellent accuracy. The PPM predictor shortens a history pattern bit by bit, and looks it up in successively smaller stages. Each stage is half the size of its predecessor. The bits correspond to branch targets, so this scheme tests ever shorter path lengths. This resembles the prediction rule of a cascaded predictor. Cascaded prediction differs from PPM prediction because a cascaded predictor employs pattern filtering and uses a separate history buffer for each stage.The number of stages is also independent from pattern length, and each stage can use any table size. Although Kalamatianos and Kaeli's study [81] demonstrates slightly better prediction performance on some of the benchmarks measured here, at least part of the improvement is due to dynamic classification of indirect branches into two classes: a class that correlates best with a history buffer which stores both conditional and indirect branch targets, and one that correlates best with only indirect branch targets. In this study we use a purely indirect branch target trace.

10.2.2 Prediction architectures

This study builds on previous work in conditional branch prediction. Many alternative implementations were conceived by adapting existing conditional branch prediction architectures to indirect branches. The literature on conditional branch prediction is extensive. We refer to Uht et al. [123] for a recent general overview, and limit the discussion to work which is closely related to this study.

10.2.2.1 Basic prediction

Yeh and Patt first proposed two-level branch prediction [134] and presented a thorough classification [135] of two-level predictors, which influenced the work discussed in Section 8.2.

Nair [105] introduced path-based branch correlation for conditional branches and showed that a path-based predictor with two-bit partial addresses attained prediction rates similar to a pattern-based predictor with taken/not taken bits (for similar hardware budgets).

Talcott's disseration work [120] influenced much of this study, and the Powersim simulator which he used to do cycle-level simulation was adapted to the SPARC architecture and used in the experiments of Section 6.

Juan, Sanjeevan and Navarro [80] extend the two-level predictor with a table that measures prediction accuracy and adapts the path length of a two-level predictor dynamically in order to improve prediction accuracy over individual programs in a benchmark suite. Although this proposal is an improvement over a statically chosen path length, it still seems too coarse-grained for optimal performance. Hybrid predictors, as discussed in Section 9, allow individual branches or even history patterns, rather than individual programs, to settle on a path length which best predicts the branch.

10.2.2.2 Hybrid prediction

McFarling first proposed to combine basic predictors into a hybrid predictor [98]. See Section 9.1.2 for a detailed discussion. Chang, Hao and Patt's study [24] and Evers, Chang and Patt's [56] present more recent results of hybrid conditional branch prediction.

Chang, Hao and Patt [23] first proposed branch classification for conditional branches. Conditional branches are divided in six classes, corresponding to the frequency with which a branch is taken in a profiling run, with boundary values 5%, 10%, 50%, 90%, 95% and 100%. The authors present a GAs predictor with multiple branch history length and shared history table, which is similar to the classifying shared-table hybrid predictor used in this study. A dynamically classifying predictor uses a fully associative branch address cache (BAC), consisting of 2-bit saturating counters that indicate the component predictors which best predicts a given branch. Since we use a simple predictor both to predict indirect branches and to classify them as "hard-to-predict", a BAC is unnecessary in a dynamically classifying cascaded predictor. By combining profile-guided classification for mostly monomorphic branches with dynamic classification for mixed-direction branches (between 10% and 90% taken), prediction accuracy of 96.4% is achieved for conditional branches of the SPECint92 benchmarks suite.

Stark, Evers and Patt [118] determine a path length per branch by profiling each branch in a separate profiling run. Their technique is an instance of profile based classification which is much more fine-grained than the classifications used in Section 9.2 (see Section 10.2.1 for more detail).

Cheng, Coffey and Mudge [28] propose Partial Prefix Matching (PPM) prediction, based on a compression technique, for conditional branch prediction. They show that a PPM predictor performs better than a two-level predictor for a similar hardware budget. Since a PPM predictor predicts for the longest pattern for which a prediction is available (choosing progressively shorter path lengths until a prediction is found), a cascaded predictor mimics this behavior. However, PPM prediction has a predetermined table size for all its components (exponentially decreasing in size), reserves a component for each history bit length, and does not employ filtering (similar to the ideal staged predictor discussed in Section 9.4.3.2).

Federovsky, Feder and Weis [57] also adapt a compression technique, Context Tree Weighting (CWT), to conditional branch prediction. They demonstrate that the technique delivers excellent prediction accuracy under the assumption of unconstrained hardware resources. They do not assess prediction performance under limited hardware budgets.

Eden and Mudge [52] proposed the YAGS architecture for conditional branch prediction. A YAGS predictor uses two kinds of predictor tables. A direct-mapped table (the choice table) stores the dominant direction of a branch using a 2-bit counter. Two tagged tables (direction tables) store a prediction for a pattern that represents history as taken/non taken bits. One of these is used for branches that are mostly taken, the other for branches that mostly not taken. Prediction in a direction table takes precedence over the prediction in the choice table, and patterns enter a direction table only if the choice table mispredicts. This scheme resembles a 2-stage cascaded predictor with a direct-mapped BTB as first stage. The main difference is that the second stage of a YAGS predictor consists of two separate tables. However, the authors agreed that this is not a requirement. A YAGS predictor shows better prediction accuracy than other conditional branch predictor schemes. We believe this is evidence that cascaded prediction is also likely to perform well on conditional branches.

Table update filtering, as used in a cascaded predictor, has a precursor in conditional branch prediction: Chang, Evers and Patt [22] study a conditional branch predictor which uses a BTB to filter out easily predicted branches. Their predictor inhibits the history table update if the BTB's confidence counter is at maximum (e.g., "strongly taken"). See Section 9.4.1.1 for a more detailed discussion and comparison with the filtering rule of cascaded predicition.

11 Future work and open problems

"Speed is exchangeable for almost anything. Any computer can emulate any other at some speed",
Burton J. Smith [115]

"Sometimes it is good that some things are impossible. I am happy there are many things that nobody can do to me."
Leonid Levin [115]

Much work remains to be done in every field we explored in the course of this study. The design space of software techniques is by no means exhausted. Both hardware and compiler techniques continue to evolve at a rapid pace, making continuous evaluation of dispatch cost necessary. The population of programs that run on the majority of processors also changes, as different trends emerge in the market place. Adaptivity, both in software and hardware, and across the software/hardware interface, seems to provide a general solution to many efficiency problems, but it has barely been explored. Finally, the hardware techniques explored for indirect branch prediction may have a wider application domains.

11.1 Software techniques

11.1.1 Inline caching with fast subtype tests

It is entirely feasible to replace the equality test in the inline cache method prefix (see Section 3.2.2) with a subtype test. Recent advances in fast subtype testing, explored by Krall, Vitek and Horspool [85], suggest that such a test can be implemented in only a few assembly instructions. The advantage of subtype testing is that the current receiver class does not need to be identical to the last observed class, but can be a descendant of a common super class, provided that the implementation is not overridden along the way. The inline cache may thus achieve a higher hit rate. An disadvantage of such a scheme is the requirement that the subtype relationship can be trusted. Whenever the class hierarchy changes, inline caches must be checked for correctness. A standard inline cache is more responsive, since a changed class can be assigned a new identity, so that old cache information results in a miss and causes an update.

11.1.2 Global caching with history

A global cache (see Section 3.2.1) is like a BTB: for every class/selector combination, exactly one target is stored. If the key into the cache were augmented with a history of previous targets, much higher hitrates may be achieved (a global cache is less accurate than an inline cache). Since global caches are easy to implement, they are the preferred form of optimization in many mid-level systems. With little extra effort, a costly dispatch table search may be avoided more often.

11.2 The software-hardware interface

11.2.1 Measuring cycle cost of more software techniques on superscalar processors

We only measured virtual function table dispatch on a range of superscalar processor designs. To evaluate other table-based techniques, in particular row-displacement dispatch, and various forms of inline caching, these techniques should be implemented in full in the context of a real programming environment. Hybrid techniques, that use inline caching but resort to a more efficient table-based technique in the miss case, can reduce dispatch cost further on processors that lack indirect branch prediction. The SUIF project [84], for instance, is an appropriate research vehicle for this work.

11.2.2 Benchmarking Java

With the growing popularity of Java, processors are likely to see more code generated by a Java compiler. Java programs are different from C++ programs. Although the all-virtual versions of C++ programs in Section 6 capture part of the behavior of their Java counterparts by forcing the Java default (non-final) on member function declarations, it would be nevertheless expedient to directly measure Java applets.

11.2.3 Influence of code-rewriting techniques on indirect branch population

Since code-rewriting techniques are capable of removing monomorphic indirect branches from the executable, they can change the nature of the indirect branch stream that the micro-architecture must predict. With statically predicted branches removed from the indirect branch stream, capacity misses will likely diminish, while the remaining indirect branches will be, on average, harder to predict accurately. Hölzle [71], shows that inlining, by removing near-monomorphic calls, increased the dynamic average number of targets (arity) of the remaining calls by more than a factor two. This may take some load off the first stage of a cascaded predictor, since fewer easily predicted branches reach the hardware, resulting in overall reduction of dispatch cost.

Some code-rewriting techniques are capable or over-optimization. For instance, a recently proposed technique for Java dispatch, by Zendra, Colnet and Collin [136], replaces all polymorphic calls by a tree of nested conditional branches. Whether this is a win depends on whether the processor's indirect or conditional branch prediction architecture is best suited to capture the regularity of target patterns of the branch. It may well be the case that this kind of optimization reduces performance, as in the case where an inline cache uses an inferior form of prediction, assuming that the hardware does not predict indirect branches at all.

11.2.4 Feedback from predictor hardware to code-rewriting techniques

Type-feedback, in combination with dynamic recompilation of a program's "hot spot's" has proven a very effective tool for program optimization in a dynamic environment [71]. Two statistics are gathered by the run-time system: the types that occur as receivers of a particular message call site and the frequency of invocation of each particular message implementation. The latter tells the dynamic (JIT) compilation system which methods are executed frequently enough so that it is worth paying a run-time recompilation cost. The former tells the optimizer which polymorphic calls are, at least temporarily, dominated by one or only a few targets, so that these can be inlined, enabling classic compiler optimizations and further reducing type tests within the inlined code. The overhead, compared to an off-line profiling optimizer (assuming the program's behavior is relatively constant) consists of the actual compilation cost, occurring at run time, and the cost of updating type feedback information and invocation counters. Type feedback can be supported by hardware.

Branch prediction hardware, by its very nature, stores statistical information about branch targets in its history table. Suppose we collect, from the history table, all targets that occur for a particular call site (an indirect branch address[1]). This gives us, in the case of a BTB, the target that occurred most recently[2] for recently executed call sites. If the BTB is sampled regularly, frequency information can be obtained from a) the presence of a target in the table, b) it's place (assuming LRU replacement) in the associativity set to which it belongs. Branch prediction hardware can therefore not only speed up non-inlined dispatch sequences, it can also, at little extra cost, suggest opportunities for inlining to a dynamic compiler. It can therefore free the run-time system from the burden of continually updating invocation counters and polymorphic inline caches (which speed up dispatch and provide type-feedback in SELF). We only need to augment the architecture with an instruction to block-dump the BTB in memory. Periodical branchmaps can be processed by measuring the difference with the previous one. The size of this delta can signal state changes in the program, and the persistent entries can designate inlining opportunities to the dynamic compiler. This may prove a useful framework to adaptively tune the recompilation strategy, for instance by increasing the sampling frequency for more reactive recompilation, or lowering it for settled systems, to reduce the run-time overhead of BTB sampling. The effectiveness of hardware-supported type-feedback, or target feedback, is still an open problem.

Merten et al. [99] propose a separate hardware device for detecting program hot spots. Their Branch Behavior Buffer (BBB) closely resembles a BTB, but is augmented with counters to detect the frequency of branch execution. In future work, they envision to take advantage of binary reoptimization opportunities enabled by using the information in the BBB.

[1] This has to be stored as a separate address, or, with only slight loss of prediction accuracy, instead of the tag in an associative table. A tag comparison becomes a comparison with the branch address. In early experiments we disambiguated entries this way, and misprediction rates were fairly similar to that of tags checking for identity of a the key pattern (history pattern XOR address).

[2] With the adjustment that 2-bit counters filter out targets that occur in isolation.

11.3 Exploring different applications for hardware-based prediction

The new prediction architectures in this study may well prove advantageous in other application domains than indirect branch prediction, such as:

- Conditional branch prediction: the Pentium II conditional branch target predictor (512 entries with for each a 4-bit local history), arguably the currently most popular predictor in practice, is a two-level predictor. It seems likely that cascaded prediction with leaky filtering will reduce cold-start and capacity misses as it does for indirect branches, since many conditional branches behave monomorphically. Chang, Evers and Patt [22] demonstrate the potential of table update filtering on conditional branches, using a different filtering rule (see the discussion in Section 9.4.1.1).

- Value prediction: this is currently an active field. Again, cascaded prediction may prove advantageous, since the value equivalent of a monomorphic branch is a constant. Run time constants occur frequently enough to warrant special treatment, as indicated in Lipasti, Wilkerson, and Shen's seminal paper [93]. They propose a constant verification unit that stores highly predictable and constant values.

- Prefetching: if the history table is augmented to store multiple alternative targets, which serve as hints to the prefetcher as to which code sequences are likely to be needed in the instruction cache, the cache missrate could be reduced. Each target could keep a counter to indicate the likelihood of its occurrence.

- Predict-ahead: when the history table stores the address of the next indirect branch as well as the target address, the predictor can run arbitrarily far ahead of execution. The average length of a precisely predicted sequence of targets equals the reciprocal of the misprediction rate. With an achievable prediction rate of 95%, an average of 20 branches could therefore be predicted ahead of time. This can be useful in all the previous application domains. The Rise mP6 processor [116] uses predict-ahead as part of its fetch unit: the *target* of the *previous* branch (not the *address* of the *current* branch) is used to look up the next target prediction in a BTB. Up to four targets can be predicted ahead of time.

- Compression: the link between compression and prediction has been pointed out multiple times in the literature (see Section 10.2.2.2). Compression of arbitrary traces can be accomplished by storing a trace consisting of the runlength of precisely predicted values and the next mispredicted value, along with a predictor specification. Accurate run-ahead prediction architectures may thus enable the building of fast, economic compression hardware.

11.4 Algorithms

The loss of prediction performance due to hardware constraints is an algorithmic issue. This study explores, for a limited application domain, the performance of indirect branch target predictors, hampered by limited resources and committed to simple prediction logic (so as to remain practical in hardware). These designs only scratch the surface of what is possible when the latency of the operation and the size of its memory resources is not as restricted as in hardware. In less constrained, less time-critical domains than branch prediction, much lower misprediction rates could be achieved.

Since trace prediction has many applications, a rigorous study on the mechanics of limited resource prediction may be both desirable and profitable. Prediction architectures similar to those presented in branch prediction literature may have applications on all kinds of domains that deal with strings. We envision applications in compression, fast string matching (for example in gene sequencing of DNA string representations), and user interface event anticipation.

11.5 Misprediction rate as a program complexity metric

A program with easily predicted branches is a simple program. *Go*, the SPECINT benchmark that implements a strategy for playing Go, is notoriously hard to predict. Part of the reason is that indirect branches occur very infrequently in *go* (once every 50K instructions), so that the indirect branch trace captures only a small part of the overall execution path. 80% prediction accuracy seems fairly good, considering this context. In contrast, we believe that the high misprediction rate of for instance *jhm* (with less than 50 instructions between indirect calls, for the best hybrid: 8%, the highest in the AVG-100 class), is due to the structural complexity of the Java class files it processes. If values, conditional branches and indirect branches are predicted, a program's behavior can be quantified as X,Y, with X the misprediction rate for a particular predictor configuration Y. This may be a practical way to quantify program execution complexity. Ideal predictors, which lack the noise of capacity, conflict and pattern interference misses, could be used to establish equivalence classes of complexity.

12 Conclusions

“We didn’t know what we wanted and how to do it. It just sort of grew....it was a big problem and what we did looks astonishingly clumsy now”, John Backus on the invention of Fortran [115]

Though we don’t claim to have discovered something as fundamental to computer science as a high-level language compiler, we have struggled hard to optimize a high-level language construct: polymorphic calls. We have gathered substantial evidence that software- and hardware-based support for polymorphic call resolution is useful, effective and affordable, and, according to current projections, will become more so in the next decade.

Polymorphic call optimization is useful because polymorphic calls are expensive. We measured a number of industry-size object-oriented programs and found that the most efficient and popular call resolution technique costs up to 30% of execution time. Since these programs are written in C++, where polymorphic calls are not the default case, it is likely that the growing popularity of fully polymorphic languages like Java will expose even more polymorphism to future processors. Near-future processors with high clock frequency need deep instruction pipelines, which are likely to increase the cost of a polymorphic call.

Polymorphic call resolution techniques are effective in reducing the run-time call cost. We studied both software and hardware techniques:

- In the software domain, we have analyzed current dispatch techniques qualitatively and quantitatively. This study showed that inline cache strategies can outperform table-based techniques by predicting indirect branches in software, even though they show bad worst-case performance. However, if hardware-based prediction is present, table-based software techniques win. We have increased the application domain of table-based call resolution by using row displacement compression on dispatch tables. Dynamically typed and multiple inheritance languages can now also use fast table-based dispatch. Row displacement compression is used in at least one industrial strength programming system, JOVE [133], a Java compiler designed to handle very large applications[1].

- In the hardware domain, we have studied dedicated indirect branch prediction architectures. Simple prediction architectures, similar to software-based prediction, can reduce the frequency of mispredicted branches by a factor four (BTB, 256 entries, 25% misprediction). More sophisticated techniques, some of which have been first presented in this work, can reduce misprediction frequency by a factor twenty (Cascaded, 3x2K entries, 5% misprediction). Table-based call resolution thereby becomes nearly as efficient as a procedural call, since only one out of twenty calls incurs a branch penalty.

[1] Though statically typed, the use of interfaces in Java necessitates compression of interface dispatch tables. These incur memory overhead like multiple inheritance dispatch tables, since they allow the merging of dispatch tables from classes unrelated by subclass (implementation) inheritance.

Polymorphic call support is affordable. Table-based dispatch techniques require affordable off-chip memory (4Mbytes for Self). The extra compilation time required to build the tables is negligible compared to total compilation cost (2.5 seconds for Self on a 60 MHz Sparc-20). Hardware indirect branch prediction starts paying off at low on-chip transistor budgets (a 32 entry BTB already predicts more than 70% of all indirect branches), and reaches 90% predictability for reasonable predictor table sizes (64+256 cascaded). In the next decade, uniprocessors may reach one billion transistors, with 48 million transistors dedicated to branch prediction [109]. Our study, and that of others [14][25][81] justifies that some of those transistors be spent on dedicated indirect branch prediction.

We believe that, using techniques as described in this dissertation, polymorphic call overhead can become negligible in all but the most time-critical applications. This enlarges the application domain of object-oriented programming, allowing programmers to use polymorphism as the default, instead of the exception. From our own experience in building software architectures, we consider this a Good Thing.

13 Glossary

*"'Tis but thy name which is my enemy;
Thou art thyself, though not a Montague.
What's a Montague? It is nor hand, nor foot,
Nor arm, nor face, nor any other part
Belonging to a man. O, be some other name!
What's in a name? That which we call a rose
By any other word would smell as sweet",*
Shakespeare [114]

Branch classification. A classifying hybrid predictor assigns to each branch a branch class. All branches that belong to a class use the same component predictor. Section 9.2

BTB. A table that stores the most recent target address of a branch instruction. Section 6.2.1

Cascaded predictor. A cascaded predictor classifies branches dynamically by observing their performance on a simple first-stage predictor. Only when this predictor fails to predict a branch correctly is a more powerful second-stage predictor permitted to store predictions for newly encountered history patterns of that branch. If both stages have a target prediction, the later stage takes precedence. Section 9.4

Dynamic caching. A message dispatch technique that caches the target of a message send at run time. If the class of the receiver object is the different from that of the cached target, a backup message dispatch technique is invoked and the cache entry is replaced with the new target. Section 3.2

Dynamic typing. A typing regime which does not require that variables are annotated with a type at compile time. At run time, a variable can contain any object type. Section 2.3.1

Dual-path hybrid predictor. A hybrid predictor with two components of different path length. Section 9.3

Duomorphic branch. A branch that jumps to two targets during an entire program run. Section 9.2.2

Hybrid predictor. A branch predictor that combines two or more simpler predictors and chooses for each branch which component to trust. Section 9.1

Indirect branch. A branch instruction which transfers control to a target address stored in a machine register. Section 1.3

Indirect branch prediction. The process of predicting the target of an indirect branch instruction, using only its address and possibly history information. Section 8

Inheritance. Incremental construction of classes (object types). A class is defined by extending an existing class, called its super class, by adding or redefining inherited functions. Section 1.2.1

Inline caching. A form of dynamic caching. Instead of storing a receiver class/message selector pair with its resulting target method in a table, a direct call to the target replaces the call to the message dispatch routine. Section 3.2.2

Message dispatch. Polymorphic call resolution in object-oriented languages. The dispatch function takes as arguments the class of the current object (the receiver), and the name of the called message (selector), and maps this pair to the appropriate implementation (method). Section 2.3

Metaprediction. The mechanism used in a hybrid predictor to decide which component prediction to trust (predict which prediction is best). Section 9.1.2

Monomorphic branch. A branch that jumps to only one target during an entire program run. Section 9.2.2

Multiple dispatch. A generalization of message dispatch. The run time target of a message depends on more than one argument, where single dispatch only takes the receiver's type (first argument) into account. Section 2.3.3

Multiple inheritance. An inheritance regime which allows a class to inherit from any number of super classes. Section 2.3.2

Polymorphic call. A polymorphic call looks like a procedural call, but where a procedural call has only one possible target subroutine, a polymorphic call can result in the execution of one of several different subroutines. The choice is typically based on the type of the first argument, and is made at run time. Section 1

Polymorphic call resolution. The action of mapping a polymorphic call to a single target at run time. Section 1.2

Predicate dispatch. A generalization of multiple dispatch. The run time target of a message depends on a predicate expression, which can be an arbitrary side-effect free expression in the underlying programming language. The most powerful form of polymorphism. Section 2.3.4

Single inheritance. An inheritance regime which requires that a class inherits from at most one super class. Section 2.3.2

Staged predictor. A cascaded predictor without filtering. Section 9.4.3

Static caching. see **Table-based dispatch**.

Static typing. A typing regime which requires that all variables in a program are annotated with a type at compile time. At run time, a variable can contain only objects of the declared type and its subtypes. Section 2.3.1

Subclassing. Implementation by incremental modification of classes through inheritance. A class inherits all functions from its super class, but can override them (redefine their implementation) and add new functions. Section 1.2.1.2

Subtyping. Incremental interface definition through inheritance. A subtype must implement at least all functions of the type it is derived from. Section 1.2.1.1

Table-based dispatch. A message dispatch technique that precomputes all valid combinations of class/selector pairs at compile or link time. At run time, dispatch is accomplished by simple table lookup, typically a small number of dependent array accesses. Section 3.3

Appendix A Basic prediction accuracy per benchmark

Table A-1 shows the misprediction rates (in %) per benchmark and averages for a fully associa-

tablesize	btb fullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4	btbfullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4
	AVG							idl						
32	28.11	30.71	32.50	30.28	25.98	22.62	25.98	6.09	6.76	6.39	6.15	7.87	8.06	7.87
64	26.83	24.26	26.30	23.60	19.77	18.53	19.77	2.99	5.20	6.38	5.55	5.33	5.02	5.33
128	25.76	20.56	22.22	19.09	16.98	15.56	16.66	2.45	4.80	4.54	3.73	2.74	3.77	3.64
256	25.13	16.99	18.06	15.15	13.73	12.47	13.29	2.41	3.40	3.13	3.69	2.67	1.23	2.10
512	25.01	13.74	15.92	12.59	11.35	10.40	10.90	2.40	2.05	3.26	2.18	1.49	1.05	1.17
1024	24.93	11.74	13.53	10.74	9.82	8.48	8.98	2.40	1.43	2.38	1.33	0.85	0.64	0.76
2048	24.92	10.27	11.48	9.49	8.52	7.76	7.82	2.40	1.13	1.80	0.82	0.69	0.62	0.46
4096	24.92	9.12	10.43	8.54	7.77	7.17	6.72	2.40	0.95	1.60	0.71	0.63	0.62	0.38
8192	24.92	8.45	9.68	8.02	7.27	6.57	5.95	2.40	0.69	1.01	0.67	0.63	0.44	0.37
16384	24.92	7.77	8.97	7.47	6.81	6.14	5.53	2.40	0.69	0.85	0.65	0.45	0.42	0.36
32768	24.92	7.09	8.46	7.07	6.57	6.02	5.21	2.40	0.55	0.81	0.50	0.45	0.42	0.35
	100-AVG							ijpeg						
32	14.97	18.92	18.07	16.82	16.81	15.86	16.81	1.26	26.62	1.52	1.64	0.82	0.32	0.82
64	13.31	15.73	16.96	14.68	14.29	13.76	14.29	1.26	26.63	51.15	0.31	0.31	0.31	0.31
128	11.72	15.01	13.98	12.33	11.77	13.63	12.35	1.26	0.92	0.56	0.31	0.31	0.39	0.29
256	10.57	12.49	12.01	12.11	11.20	9.85	10.08	1.26	0.69	0.56	0.65	0.39	0.39	0.29
512	10.32	9.84	12.27	9.84	9.09	8.20	8.22	1.26	0.44	0.39	0.39	0.39	0.39	0.37
1024	10.14	8.42	10.77	8.48	7.15	6.37	6.66	1.26	0.45	0.39	0.39	0.46	0.46	0.37
2048	10.11	7.41	9.21	6.97	6.37	6.12	5.96	1.26	0.45	0.39	0.46	0.46	0.51	0.38
4096	10.11	6.69	8.54	6.29	5.77	5.62	5.47	1.26	0.44	0.39	0.46	0.46	0.51	0.47
8192	10.11	6.21	6.91	5.92	5.62	5.26	4.83	1.26	0.50	0.46	0.46	0.51	0.56	0.47
16384	10.11	5.95	6.43	5.69	5.42	4.93	4.54	1.26	0.56	0.46	0.51	0.56	0.62	0.47
32768	10.11	5.56	6.10	5.68	5.26	4.86	4.42	1.26	0.56	0.46	0.56	0.56	0.62	0.48
	200-AVG							ixx						
32	39.38	40.82	44.86	41.81	33.84	28.40	33.84	46.58	33.15	56.21	51.11	30.07	24.30	30.07
64	38.42	31.58	34.30	31.24	24.48	22.61	24.48	45.75	24.47	32.91	24.86	18.10	15.94	18.10
128	37.79	25.32	29.28	24.89	21.44	17.22	20.36	45.70	24.60	27.55	17.93	15.37	12.10	19.02
256	37.61	20.84	23.24	17.77	15.91	14.72	16.04	45.70	18.74	20.58	15.47	12.57	10.34	13.32
512	37.61	17.08	19.05	14.94	13.29	12.28	13.20	45.70	13.88	21.38	13.48	10.63	10.25	9.48
1024	37.61	14.58	15.90	12.67	12.10	10.28	10.98	45.70	12.37	15.32	11.57	9.03	8.21	8.56
2048	37.61	12.73	13.43	11.65	10.36	9.16	9.42	45.70	10.90	12.51	9.82	8.47	6.94	6.06
4096	37.61	11.21	12.05	10.46	9.47	8.51	7.80	45.70	9.34	12.11	9.29	8.29	6.94	5.06
8192	37.61	10.37	12.06	9.82	8.69	7.69	6.91	45.70	8.86	11.28	8.94	7.11	5.86	4.91
16384	37.61	9.32	11.15	9.00	8.00	7.18	6.38	45.70	7.66	10.55	8.04	5.98	5.58	4.70
32768	37.61	8.39	10.48	8.26	7.68	7.01	5.88	45.70	6.79	10.14	6.70	5.94	5.58	4.15
	INFREQ-AVG							lcom						
32	31.78	31.68	32.32	31.88	20.85	17.95	20.85	5.18	12.99	5.80	5.24	4.81	4.45	4.81
64	31.78	27.61	34.80	18.16	17.74	17.54	17.74	4.71	4.67	4.78	4.03	3.84	3.62	3.84
128	31.78	23.16	20.75	17.96	17.54	15.66	21.46	4.46	4.75	3.90	3.61	3.47	3.49	3.72
256	31.78	19.92	18.07	17.52	15.58	14.98	17.03	4.25	3.93	3.43	3.37	3.25	3.27	2.96
512	31.78	15.26	20.34	16.44	15.18	14.97	11.31	4.25	3.25	3.26	2.80	2.60	2.53	2.56
1024	31.78	14.27	19.40	16.09	10.89	10.40	10.43	4.25	2.68	2.77	2.33	2.20	1.88	1.95
2048	31.78	13.54	18.06	12.86	10.67	10.16	10.09	4.25	2.17	2.43	1.85	1.68	1.65	1.69
4096	31.78	12.04	16.48	11.87	10.44	10.06	8.31	4.25	1.83	2.12	1.56	1.42	1.34	1.31
8192	31.78	12.12	14.41	11.76	10.50	10.81	7.84	4.25	1.62	1.71	1.44	1.36	1.34	1.11
16384	31.78	12.66	14.17	11.67	11.22	9.17	8.60	4.25	1.55	1.58	1.37	1.37	1.39	1.04
32768	31.78	11.14	13.98	11.89	11.07	9.10	8.90	4.25	1.43	1.49	1.39	1.35	1.39	1.02
	OO-AVG							m88ksim						

Table A-1. Misprediction rates (per benchmark and averages) for basic predictors.

tablesize	btb fullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4	btbfullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4
32	23.70	25.80	28.17	25.96	22.64	20.02	22.64	76.41	56.87	78.29	76.41	45.33	35.99	45.33
64	22.18	20.43	22.55	18.96	17.18	16.04	17.18	76.41	46.52	49.25	35.99	35.99	35.99	35.99
128	20.87	20.75	18.40	15.33	14.13	15.43	15.47	76.41	43.78	46.40	35.99	35.99	26.55	46.11
256	19.97	17.00	15.48	14.79	13.29	11.95	12.16	76.41	36.17	35.99	29.46	26.55	26.55	32.27
512	19.81	13.37	15.30	11.95	10.61	9.50	10.50	76.41	22.07	32.29	27.54	26.55	26.55	15.38
1024	19.69	11.29	12.63	9.91	9.47	7.93	8.49	76.41	20.16	30.44	27.54	14.40	14.40	13.56
2048	19.67	9.81	10.51	9.14	8.09	7.21	7.50	76.41	19.31	30.43	17.48	14.40	13.40	8.72
4096	19.67	8.67	9.68	8.21	7.38	6.70	6.48	76.41	15.41	30.43	14.58	14.40	13.40	2.17
8192	19.67	8.03	9.32	7.75	6.90	6.31	5.76	76.41	14.39	16.40	14.40	13.40	10.58	2.17
16384	19.67	7.43	8.57	7.21	6.49	5.91	5.35	76.41	14.43	16.40	13.40	10.58	3.07	4.00
32768	19.67	6.82	8.00	6.76	6.33	5.87	5.06	76.41	9.64	16.40	10.58	10.58	3.07	4.00
	C-AVG							perl						
32	34.91	36.72	37.28	35.94	27.18	23.20	27.18	31.80	60.62	31.81	31.80	36.31	22.80	36.31
64	34.54	30.25	34.76	26.10	21.67	20.83	21.67	31.80	45.24	45.27	49.74	22.76	22.75	22.76
128	34.27	21.65	25.78	22.75	20.46	15.76	20.41	31.80	0.37	40.74	36.21	22.72	0.34	0.29
256	34.25	18.43	20.96	16.75	15.16	14.31	16.43	31.80	0.35	22.73	0.33	0.32	0.29	0.26
512	34.25	14.90	18.84	15.23	14.10	13.69	11.56	31.80	0.31	0.33	0.32	0.29	0.28	0.27
1024	34.25	13.50	17.48	14.34	10.75	10.05	10.26	31.80	0.31	0.32	0.30	0.34	0.32	0.26
2048	34.25	12.43	15.87	11.56	10.08	9.57	9.31	31.80	0.30	0.31	0.34	0.33	0.37	0.27
4096	34.25	11.09	14.31	10.58	9.54	9.15	7.79	31.80	0.30	0.30	0.33	0.32	0.37	0.33
8192	34.25	10.75	12.45	10.19	9.31	8.98	7.12	31.80	0.34	0.35	0.33	0.37	0.40	0.32
16384	34.25	10.60	12.02	9.87	9.38	7.91	7.26	31.80	0.38	0.34	0.37	0.40	0.45	0.33
32768	34.25	9.42	11.73	9.82	9.08	7.72	7.22	31.80	0.38	0.33	0.41	0.40	0.45	0.34
	SPEC-AVG							porky						
32	34.05	35.89	34.75	34.29	25.64	21.58	25.64	21.70	28.88	26.38	23.76	22.52	19.41	22.52
64	34.04	30.36	34.64	25.27	20.80	20.39	20.80	21.21	19.74	21.75	17.41	15.30	13.97	15.30
128	34.02	20.11	25.21	22.61	20.19	14.91	19.45	21.08	19.24	17.53	13.97	13.15	9.68	16.11
256	34.02	17.20	20.64	15.99	14.49	13.68	15.82	20.80	14.94	15.96	12.43	10.22	9.12	11.04
512	34.02	13.72	18.12	14.80	13.78	13.48	10.70	20.80	12.28	12.53	10.16	9.01	8.90	9.28
1024	34.02	12.63	16.96	14.22	9.95	9.43	9.67	20.80	10.37	10.03	8.56	7.89	6.97	7.59
2048	34.02	11.72	15.46	11.00	9.49	9.02	8.99	20.80	9.30	8.96	7.78	6.88	5.13	6.99
4096	34.02	10.39	14.23	10.05	9.01	8.64	7.44	20.80	8.39	8.43	7.24	6.26	4.85	5.41
8192	34.02	10.12	11.76	9.71	8.67	8.56	6.76	20.80	7.22	9.13	6.98	5.30	5.05	5.05
16384	34.02	10.22	11.39	9.26	8.79	7.35	7.01	20.80	6.07	7.68	5.64	5.22	4.61	4.77
32768	34.02	8.95	11.13	9.32	8.51	7.13	7.05	20.80	5.37	6.53	5.19	4.99	4.61	4.29
	beta							self						
32	31.85	27.98	36.63	33.76	25.20	16.89	25.20	36.08	49.54	48.36	46.15	45.21	40.81	45.21
64	30.48	21.88	23.13	19.59	14.73	12.22	14.73	32.34	44.08	46.73	39.69	38.86	36.39	38.86
128	29.44	18.65	16.62	14.66	10.77	13.06	13.03	25.07	39.80	39.03	32.54	31.61	36.84	34.00
256	28.57	13.88	12.88	13.04	10.99	10.44	10.17	18.41	33.05	32.08	31.53	29.44	26.91	26.15
512	28.57	10.20	13.75	9.38	7.46	5.51	8.36	16.94	26.80	29.39	23.94	22.18	19.70	23.26
1024	28.57	7.84	10.64	6.04	5.61	3.19	4.92	15.88	22.37	24.27	19.48	18.61	16.62	17.11
2048	28.57	6.29	6.19	5.13	3.77	2.66	3.23	15.68	18.33	19.69	16.81	15.40	16.01	14.64
4096	28.57	4.42	4.78	4.12	3.28	2.63	2.20	15.68	15.49	16.84	14.03	12.90	13.44	13.38
8192	28.57	3.77	4.91	3.62	2.73	2.46	1.99	15.68	14.00	15.43	12.27	11.91	11.54	10.83
16384	28.57	3.49	4.46	2.81	2.51	2.28	1.88	15.68	13.51	13.73	11.55	11.43	10.50	9.25
32768	28.57	2.91	4.18	2.79	2.49	2.28	1.68	15.68	11.87	12.63	11.51	10.72	10.10	8.61
	edg							jhm						
32	40.88	42.59	54.99	47.48	37.97	34.57	37.97	11.45	14.05	12.24	11.81	14.40	14.03	14.40
64	37.97	29.48	35.61	31.89	27.78	23.89	27.78	11.16	12.92	15.14	13.03	12.88	12.49	12.88
128	35.99	32.48	29.80	23.77	22.36	21.72	27.08	11.13	15.12	12.22	11.96	11.76	13.36	11.81
256	35.91	27.09	23.22	22.07	19.86	18.73	20.72	11.13	13.81	11.80	12.98	12.40	11.31	11.35
512	35.91	23.17	23.84	18.18	16.36	15.18	17.57	11.13	12.75	14.64	12.18	11.27	10.59	11.00
1024	35.91	19.63	21.13	15.16	16.34	14.40	14.46	11.13	11.93	13.57	11.18	11.33	9.67	10.35
2048	35.91	17.40	18.73	15.53	14.19	13.46	11.59	11.13	11.59	13.03	12.10	10.92	9.41	9.85

Table A-1. Misprediction rates (per benchmark and averages) for basic predictors.

tablesize	btb fullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4	btbfullassoc	tagless	assoc1	assoc2	assoc4	fullassoc	hybrid assoc4
4096	35.91	15.94	14.82	14.27	13.28	12.75	10.20	11.13	11.08	12.83	11.42	10.22	9.30	9.16
8192	35.91	15.10	17.25	13.58	13.81	11.94	9.62	11.13	11.12	11.89	11.20	10.75	9.44	8.45
16384	35.91	13.24	16.39	14.15	13.52	11.86	9.00	11.13	10.33	11.40	11.38	10.29	8.75	8.40
32768	35.91	12.65	15.97	13.37	13.06	11.86	8.38	11.13	10.14	10.89	10.51	10.21	8.75	8.32
	eqn							troff						
32	36.87	37.65	39.37	37.60	33.44	32.77	33.44	17.50	21.21	22.12	18.05	20.20	19.48	20.20
64	35.82	32.30	32.68	29.38	29.16	28.00	29.16	15.16	18.57	19.47	17.08	16.45	16.70	16.45
128	34.78	38.22	28.85	26.17	25.56	29.27	24.95	13.70	21.56	15.39	13.44	12.72	17.32	12.91
256	34.78	34.08	26.35	27.28	25.97	25.58	21.95	13.70	17.20	13.14	13.27	12.06	9.38	10.43
512	34.78	28.22	26.77	23.17	21.20	18.69	21.04	13.70	10.93	12.69	10.28	9.64	8.29	8.37
1024	34.78	23.41	23.22	19.72	21.93	16.89	18.16	13.70	9.24	11.50	8.96	7.76	7.33	7.00
2048	34.78	20.18	19.82	20.29	17.62	15.27	17.85	13.70	8.39	10.17	7.68	7.37	7.20	6.75
4096	34.78	18.60	18.69	18.02	16.08	13.99	14.68	13.70	7.93	9.71	7.48	7.34	7.20	6.74
8192	34.78	17.22	20.24	17.23	15.01	13.52	12.55	13.70	7.79	8.29	7.41	7.26	7.13	6.54
16384	34.78	15.95	18.79	16.08	13.87	12.56	11.39	13.70	7.59	8.12	7.35	7.25	7.15	6.40
32768	34.78	14.81	17.39	14.74	13.71	12.56	10.61	13.70	7.45	7.91	7.53	7.16	7.15	6.49
	gcc							vortex						
32	65.96	54.84	68.66	67.18	51.35	48.07	51.35	20.19	19.08	20.21	20.22	13.01	10.98	13.01
64	65.89	47.92	48.75	45.84	43.52	41.52	43.52	20.19	13.42	14.88	12.67	11.30	10.71	11.30
128	65.74	43.71	43.90	41.51	40.11	34.36	42.02	20.19	14.86	12.38	12.01	10.69	12.17	14.92
256	65.70	36.81	40.94	33.73	31.43	28.55	34.80	20.19	13.16	12.18	16.78	13.10	12.08	12.78
512	65.70	31.49	34.77	29.89	28.09	27.15	26.38	20.19	12.09	25.94	16.40	12.56	12.07	7.68
1024	65.70	28.12	30.68	27.31	23.59	21.96	22.89	20.19	11.37	24.43	15.16	7.12	6.36	7.63
2048	65.70	24.72	27.50	22.64	21.29	20.31	19.97	20.19	11.29	19.80	11.68	6.89	5.90	7.81
4096	65.70	21.45	25.23	19.97	18.82	18.02	16.70	20.19	11.03	13.60	11.65	6.39	5.90	8.24
8192	65.70	20.04	21.24	18.07	16.50	14.60	13.95	20.19	10.83	19.57	11.65	7.12	10.98	7.50
16384	65.70	18.48	19.83	15.92	14.49	12.93	12.56	20.19	11.92	19.02	11.48	11.49	9.89	8.12
32768	65.70	15.83	18.79	14.64	13.20	11.71	11.72	20.19	11.34	18.34	12.63	11.38	9.89	8.15
	go							xlisp						
32	29.25	24.17	29.25	29.25	24.25	24.52	24.25	13.51	9.00	13.51	13.51	8.39	8.36	8.39
64	29.25	23.87	23.92	23.66	23.38	23.13	23.38	13.51	8.93	9.25	8.71	8.35	8.34	8.35
128	29.25	33.07	23.66	23.52	23.16	23.51	24.52	13.51	4.03	8.82	8.69	8.34	7.03	8.02
256	29.25	29.67	23.57	23.18	22.27	20.89	22.80	13.51	3.53	8.48	7.80	7.36	7.03	7.50
512	29.25	26.42	22.72	21.44	21.21	20.88	21.80	13.51	3.23	10.41	7.66	7.35	7.03	2.98
1024	29.25	25.08	22.34	21.25	21.57	20.39	20.18	13.51	2.90	10.12	7.62	2.17	2.09	2.77
2048	29.25	23.13	21.61	21.84	20.95	20.83	23.44	13.51	2.85	8.17	2.54	2.15	1.81	2.35
4096	29.25	21.29	21.51	20.81	20.52	20.45	22.37	13.51	2.84	8.15	2.53	2.14	1.81	1.83
8192	29.25	22.75	21.19	20.52	20.97	21.10	21.21	13.51	2.01	3.11	2.53	1.81	1.67	1.71
16384	29.25	23.73	20.80	21.31	22.23	23.09	21.81	13.51	2.06	2.90	1.83	1.76	1.37	1.78
32768	29.25	23.00	20.74	23.76	21.75	22.82	22.95	13.51	1.92	2.84	2.62	1.71	1.37	1.74

Table A-1. Misprediction rates (per benchmark and averages) for basic predictors.

tive BTB, two-level predictors with tagless, one-way, two-way, four-way and fully associative tables of given entry size, and a dual pathlength hybrid predictor with 2-bit confidence counters and 4-way associative table, for comparison. The pathlengths for the best predictor and component predictors is given in Table A-2. Note that the misprediction rates of individual benchmarks in Table A-1 may occasionally increase even for a larger table size, since the pathlength is chosen to minimize the AVG misprediction rate, and some benchmarks go against the general trend, especially for small tables of low associativity.

predictor type	32	64	128	256	512	1024	2048	4096	8192	16384	32768
tagless	1	1	3	3	3	3	3	3	4	5	5
assoc2	0	1	1	2	2	2	3	3	3	4	5
assoc4	1	1	1	2	2	3	3	3	4	5	5
fullassoc	1	1	2	2	2	3	4	4	5	6	6
hybrid assoc4	1	1	2.0	2.0	3.1	3.1	5.1	6.2	6.2	7.2	8.2

Table A-2. Path length of best predictor for each associativity. The hybrid predictor has two path lengths, one for each component. When the same-size non-hybrid predictor is better, we only give the single path length.

14 References

[1] Personal Communication at the ECOOP98 Joint workshop on "Technologies for High-Performance Computing" and "Parallel Object-oriented Scientific Computing", Brussels, July 98

[2] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993.

[3] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of parametric Polymorphism. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming*, p. 2-26, Arhus, Denmark, August 1995. Springer-Verlag LNCS 952..

[4] Ole Agesen and Urs Hölzle. *Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages*. Technical Report TRCS 95-04, Department of Computer Science, UCSB, March 1995.

[5] Gerald Aigner. *VPROF: A Virtual Function Call Profiler for C++*. Unpublished manuscript, 1995.

[6] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. *ECOOP '96 Conference Proceedings*, Linz, Austria, Springer Verlag Lecture Notes in Computer Science, July 1996.

[7] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *OOPSLA '94 Conference Proceedings*, pp. 244-258, October 1994. Published as SIGPLAN Notices 29(10), October 1994.

[8] A.V.Aho, J.E.Hopcroft, J.D.Ullman. *Data Structures and Algorithms.* Addison-Wesley 1983.

[9] P. André and J.-C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. *OOPSLA '92 Conference Proceedings*, Vancouver, Canada, October 1992.

[10] Apple Computer, Inc. *Object Pascal User's Manual*. Cupertino, 1988.

[11] The Beatles. A Little Help from my Friends. *Album: Stg. Pepper's Lonely Hearts Club Band.* 1967

[12] Francois Bodin and Andre Seznec. Skewed Associativity Improves Program Performance and Enhances Predictability. IEEE Transactions on Computers. Vol. 48, No. 5, May 1997.

[13] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Journal of Programming Languages, pages 313-351, Vol 2, Num 4, 1994.

[14] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

[15] *The Cambridge Encyclopedia*. Cambridge University Press 1990.

[16] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[17] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June 1991.

[18] Craig Chambers. Object-Oriented Multi-Methods in Cecil. *ECOOP'92 Conference Proceedings.*

[19] Craig Chambers. *The Cecil Language: Specification and Rationale.* Technical Report 93-03-05, Department of Computer Science and Engeneering, University of Washington.

[20] Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. To appear in *OOPSLA'99 Conference Proceedings*.

[21] Craig Chambers and Weimin Chen. *Efficient Predicate Dispatching*. Technical Report UW-CSE-98-12-02, Department of Computer Science and Engineering, University of Washington.

[22] Po-Yung Chang, Marius Evers, Yale N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *Proceedings of the International Conference on Parallell Architectures and Compilation Techniques Proceedings*, October 1996.

[23] Po-Yung Chang, Eric Hao, Yale N. Patt. Branch classification: A new mechanism for improving branch predictor performance. *MICRO '27* Novermber 1994.

[24] Po-Yung Chang, Eric Hao, Yale N. Patt. Alternative Implementations of Hybrid Branch Predictors. *MICRO '28 Proceedings*, November 1995.

[25] Po-Yung Chang, Eric Hao, Yale N. Patt. Target Prediction for Indirect Jumps. *ISCA'97 Proceedings*, July 1998

[26] Weimin Chen, Volker Turau, Wolfgang Klas. Efficient Dynamic Look-Up Strategie for Multi-Methods. In *ECOOP'94 Conference Proceedings*, Bologna, Italy, 1994.

[27] Weimin Chen, Volker Turau. Multiple-Dispatching Based on Automata. *Theory and Practice of Object Systems*, 1(1):41-59, 1995.

[28] I-Cheng K.Chen, John T.Coffey, Trevor N. Mudge. Analysis of Branch Prediction via Data Compression. *ASPLOS'96 Proceedings*.

[29] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993. Also published as Technical Report CSE-TR 93-06-06, University of Washington, 1993.

[30] T. Conroy and E. Pelegri-Llopart. An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In 75.

[31] Cypress Semiconductors. *CY7C601 SPARC processor*, 1990.

[32] O.-J. Dahl and B. Myrhaug. *Simula Implementation Guide*. Publication S 47, NCC, March 1973.

[33] Jeffrey Dean, Craig Chambers, and David Grove. Optimization of Object-Oriented Programs using Class Hierarchy Analysis. *ECOOP '95 Proceedings*, Aarhus, Denmark, August 1995.

[34] P. Dencker, K. Dürre, and J. Heuft. Optimization of Parser Tables for Portable Compilers. *TOPLAS* 6(4):546-572, 1984.

[35] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[36] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. *Proceedings of OOPSLA '96*, San Jose, CA, October, 1996.

[37] Theo D'Hondt. Personal communication, 1990.

[38] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *OOPSLA '89 Conference Proceedings*, pp. 211-214, New Orleans, LA, October 1989.

[39] Karel Driesen. Selector Table Indexing and Sparse Arrays. *OOPSLA '93 Conference Proceedings*, p. 259-270, Washington, D.C., 1993. Published as SIGPLAN Notices 28(10), September 1993.

[40] Karel Driesen. *Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages*. Master's Thesis, Vrije Universiteit Brussel, 1993.

[41] Karel Driesen. Compressing Sparse Tables using a Genetic Algorithm. In *Proceedings of the GRONICS '94 Student Conference*, Groningen, February 1994.

[42] Karel Driesen, Urs Hölzle, Jan Vitek. Message Dispatch on Modern Computer Architectures. *ECOOP '95 Conference Proceedings*, p. 253-282, Århus, Denmark, August 1995. Published as Springer Lecture Notes in Computer Science Vol. 952, Springer-Verlag, Berlin Heidelberg 1995

[43] Karel Driesen, Urs Hölzle. Minimizing Row Displacement Dispatch Tables. *OOPSLA'95 Conference Proceedings*: p. 141-155, Austin,Texas,October 1995, Published as SIGPLAN Notices 30(10), October 1995.

[44] Karel Driesen. *Multiple Dispatch Techniques: a survey.* Third place in the 1996 Student Writing Contest of the Society for Technical Communication, Santa-Barbara Chapter. White paper at (http://www.cs.ucsb.edu/~karel), Santa-Barbara, 1996.

[45] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA '96 Conference proceedings*, October 1996.

[46] Karel Driesen and Urs Hölzle. Limits of Indirect Branch Prediction. Technical Report TRCS97-10, Department of Computer Science, University of California Santa-Barbara,June 25, 1997 (http://www.cs.ucsb.edu/oocsb/papers/TRCS97-10.html)

[47] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. *ISCA '98 Conference Proceedings*, pp. 167-178, Barcelona, July 1998

[48] Karel Driesen and Urs Hölzle. Improving Indirect Branch Prediction With Source- and Arity-based Classification and Cascaded Prediction. Technical Report TRCS98-07, Computer Science Department, University of California, Santa Barbara, 15 March 1998 (http://www.cs.ucsb.edu/oocsb/papers/TRCS98-07.html)

[49] Karel Driesen and Urs Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. *Micro'98 Conference Proceedings*, Dallas, Texas, December 1998.

[50] Karel Driesen and Urs Hölzle. Multi-stage Cascaded Prediction. *Euro-Par'99 Conference Proceedings*, Toulouse, France, September 1999

[51] P. Dussud. TICLOS: An implementation of CLOS for the Explorer Family. *OOPSLA '89 Conference Proceedings*, pp. 215-220, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[52] A.N.Eden and T.Mudge. The YAGS Branch Prediction Scheme. *Micro'98 Conference Proceedings*, Dallas, Texas, December 1998.

[53] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[54] Joel Emer and Nikolas Gloy. A language for describing predictors and its application to automatic synthesis. *ISCA'97 Proceedings,* July 1997.

[55] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *ECOOP '98 Proceedings*, Brussels, Belgium, July 1998.

[56] Marius Evers, Po-Yung Chang, Yale N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the presence of context switches. *Proceedings of ISCA'96*.

[57] E.Federovsky, M.Feder, S.Weiss. Branch Prediction based on Universal Data Compression Algorithms. *ISCA '98 Conference Proceedings*, pp. 62-72, Barcelona, July 1998

[58] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. To appear in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[59] Charles D. Garrett, Jeffrey Dean, David Grove, and Craig Chambers. *Measurement and Application of Dynamic Receiver Class Distributions*. Technical Report CSE-TR-94-03-05, University of Washington, February 1994.

[60] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Second Edition, Addison-Wesley, Reading, MA, 1985.

[61] J. Gosling, B.Joy and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading Massachussetts,1996.

[62] David Grove, Jeffrey Dean, Charles D. Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications,* p. 108-123, Austin, TX, October 1995.

[63] Linley Gwennap. Digital leads the pack with 21164. *Microprocessor Report* 8(12), September 12, 1994.

[64] Eric Hao, Po-Yung Chang, and Yale Patt. The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited. *Proceedings of the 27th International Symposium on Microarchitecture*, San Jose, California, November, 1994.

[65] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach. Second Edition*. Morgan Kaufmann Publishers Inc. 1996.

[66] W. Holst, and D. Szafron, A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages. *Proceedings of ECOOP'97*, Finland, Lecture Notes in Computing Science 1241, Springer Verlag, June 1997, pp. 276-301

[67] W. Holst, and D. Szafron. Incremental Table-Based Method Dispatch for Reflective Object-Oriented Languages. Proceedings of TOOLS USA 97, Santa Barbara, California, July 1997.

[68] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin, 1991.

[69] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. *Proceedings of the ACM SIGPLAN `92 Conference on Programming Language Design and Implementation*, pp. 32-43, San Francisco, June, 1992. Published as SIGPLAN Notices 27(7), July, 1992.

[70] Urs Hölzle and David Ungar. Optimizing Dynamically-dispatched Calls With Run-Time Type Feedback. In *PLDI '94 Conference Proceedings*, pp. 326-335, Orlando, FL, June 1994. Published as *SIGPLAN Notices 29(6)*, June 1994.

[71] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Ph.D. Thesis, Department of Computer Science, Stanford University, August 1994.

[72] Urs Hölzle and David Ungar. Do Object-Oriented Languages Need Special Hardware Support? *ECOOP '95 Conference Proceedings*, Århus, Denmark, August 1995.

[73] Urs Hölzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Trans. Programming Languages and Systems* 18(4):355-400, 1996.

[74] Shih-Kun Huang, Deng-Jyi Chen. Two-way Coloring Approaches for Method Dispatching in Object-Oriented Programming Systems. *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference*, pp. 39-44, Chicago, 1992.

[75] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. *OOPSLA '86 Conference Proceedings.*

[76] Intel press release. *The Next Generation of Microprocessor Architecture: A 64-bit Instruction Set Architecture (ISA) Based on EPIC Technology.* Intel Corporation October 1997 (http://www.intel.com/pressroom/archive/backgrnd/sp101497.HTM)

[77] Quinn Jacobson, Steve Bennet, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. *HPCA-3 proceedings*, February 1996.

[78] Ralph Johnson. Workshop on Compiling and Optimizing Object-Oriented Programming Languages. *OOPSLA '87 Addendum to the Proceedings*, 1988.

[79] N.P.Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of ISCA'90.*

[80] T.Juan, S.Sanjeevan, J.Navarro. Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction. *ISCA '98 Conference Proceedings*, pp. 155-166, Barcelona, July 1998

[81] John Kalamatianos and David Kaeli. Predicting Indirect Branches via Data Compression. *Micro'98 Conference Proceedings*, Dallas, Texas, December 1998

[82] David Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. *ISCA '91 Proceed*ings, May 1991.

[83] Gregor Kiczales and Louis Rodriguez. Efficient Method Dispatch in PCL. *Proc. ACM Conf. on Lisp and Functional Programming*, 1990. Also in [107].

[84] Holger Kienle and Urs Hölzle. Introduction to the SUIF 2.0 Compiler System. Technical Report TRCS97-22, Department of Computer Science, University of California Santa-Barbara, December 1997 (http://www.cs.ucsb.edu/oocsb/papers/TRCS97-22.html)

[85] Andreas Krall, Jan Vitek and R. N. Horspool. Near Optimal Hierarchical Encoding of Types. In *ECOOP'97 conference Proceedings,* Jyväskylä, Finland, June 1997. Published as *Lecture Notes in Computer Science, Vol. 1241, Springer, 1997.*

[86] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[87] Stein Krogdahl. Multiple Inheritance in Simula-like Languages. *BIT* 25, pp. 318-326, 1985.

[88] James Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *PLDI '95 Conference Proceedings*, pp. 291-300, La Jolla, CA, June 1995. Published as *SIGPLAN Notices 30(6)*, June 1995.

[89] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer 17(1)*, January 1984.

[90] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley 1993.

[91] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-Based Debugging of Object-Oriented Programs. *OOPSLA'97 proceedings*, SIGPLAN Notices 32(10), October 1997.

[92] Mark Linton, John Vlissides, Paul Calder. Composing User Interfaces with Interviews. IEEE Computer 22(2), pp. 8-22, February 1989.

[93] Mikko H.Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), October 1996, pp. 138-147

[94] Mikko H.Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 226-237

[95] MIPS Inc. *R4000* Technical Brief, 1992

[96] MIPS Inc. *R10000* Technical Brief, September 1994

[97] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 217-227

[98] S. McFarling, Combining Branch Predictors, *WRL Technical Note TN-36*, Digital Equipment Corporation, June 1993.

[99] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, Wen-Mei W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. *To appear in the proceedings of ISCA'99.*

[100] Microprocessor Report. *HP PA8000 Combines Complexity and Speed*. Volume 8, Number 15, November 14, 1994.

[101] Microprocessor Report. *Intel's P6 Uses Decoupled Superscalar Design*. Volume 9, Number 2, February 16, 1995.

[102] S. Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages*. Technical Report TR-CS-94-02, The Australian National University, Canberra, January 1994.

[103] MIPS Inc. *R4000* Technical Brief, 1992.

[104] MIPS Inc. *R10000* Technical Brief, September 1994.

[105] Ravi Nair. Dynamic Path-Based Branch Correlation. *Proceedings of MICRO-28*, 1995.

[106] Nicholas Oxhøy, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *ECOOP '92 Conference Proceedings*, p. 329-349, Utrecht, The Netherlands, June/July 1992

[107] Andreas Paepcke (ed.). *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[108] Candy Pang, Wade Holst, Yuri Leontiev and Duane Szafron. Multi-Method Dispatch Using Multiple Row Displacement. *To appear in the proceedings of ECOOP'99*.

[109] Yale N.Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, Jared Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, September 1997

[110] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94 Conference Proceedings*, pp. 324-340, October 1994. Published as SIGPLAN Notices 29(10), October 1994.

[111] William Pugh and Grant Weddell. Two-Directional Record Layout for Multiple Inheritance. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 85-91, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June 1990.

[112] John Rose. Fast Dispatch Mechanisms for Stock Hardware. *OOPSLA'88 Conference Proceedings*, p. 27-35, San Diego, CA, November 1988. Published as *SIGPLAN Notices 23(11)*, November 1988.

[113] Stuart Sechrest, Chich-Chieh Lee, and Trevor Mudge. The role of adaptivity in two-level adaptive branch prediction. *Proceedings of MICRO-29*, November 1995.

[114] William Shakespeare. *Romeo and Juliet*. Bantam Books 1988.

[115] Dennis Shasha and Cathy Lazere. *Out of their MINDS: The Lives and Discoveries of 15 Great Computer Scientists*. Springer-Verlag New York, 1995.

[116] Michael Slater. Rise Joins x86 Fray With mP6. *Microprocessor Report* 12(15), November 16, 1998.

[117] Harini Srinivasan and Peter Sweeney. *Evaluating Virtual Dispatch Mechanisms for C++*. IBM Technical Report RC 20330, Thomas J. Watson Research Laboratory, December 1995.

[118] Jared Stark, Marius Evers, and Yale N. Patt. Variable Length Path Branch Prediction. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, October 1998

[119] Bjarne Stroustrup. *The C++ Programming Language.* Third edition. Addison-Wesley Publishing Company, Reading Massachusetts, 1997

[120] Adam R. Talcott. *Reducing the Impact of the Branch Problem in Superpipelined and Superscalar Processors*. Ph.D. Thesis, Department of Computer Science, University of California Santa Barbara, June 1995

[121] R.E.Tarjan, A.C.Yao. Storing a Sparse Table. Communications of the ACM, 22(11), November 1979, pp. 606-611.

[122] Kresten Krab Thorup. Optimizing Method Lookup in Dynamic Object-Oriented Languages with Sparse Arrays. *Proceedings of the Annual SUUG Conference on Free Software 1993*, Moscow, Russia 1993.

[123] Augustus K. Uht, Vijay Sindagi, Sajee Somanathan. Branch Effect Reduction Techniques. *IEEE Computer*, May 1997.

[124] *UltraSPARC ™ User's Manual Revision 2.0*, Sun Microelectronics, 1996

[125] David Ungar and David Patterson. Berkeley Smalltalk: Who Knows Where the Time Goes? In [58].

[126] David Ungar and David Patterson. What Price Smalltalk? In *IEEE Computer* 20(1), January 1987.

[127] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, Cambridge, MA, 1987.

[128] Jan Vitek and R. N. Horspool. Taming Message Passing: Efficient Method Look-Up for Dynamically-Typed Languages. In *ECOOP '94 Conference Proceedings*, Bologna, Italy, 1994.

[129] Jan Vitek and R. N. Horspool. Compact Dispatch Tables for Dynamically Typed Object-Oriented Languages. *Proceedings of 7th International Conference on Compiler Construction*, CC'98, Lisbon, Portugal, March 1998. Published as *Lecture Notes in Computer Science*, Vol. 1383, Springer, 1998

[130] Jan Vitek, R. N. Horspool, and J. Uhl. Compile-time analysis of object-oriented programs. *Proc. CC'92, 4th International Conference on Compiler Construction*, pp. 236-250, Paderborn, Germany, Springer-Verlag, 1992.

[131] A. Weinand, E. Gamma, and R. Marty. ET++—An Object-Oriented Application Framework in C++. *OOPSLA '88 Conference Proceedings*, pp. 46-57, October 1988.

[132] Jennifer West, Personal Communication, June 1999.

[133] Allen Wirfs-Brock. Large Java Applications Breaking the Speed Limit. Presentation at the JavaOne conference in San-Fransisco, March 24, 1998. Slides available at http://www.instantiations.com/java1talk/index.htm.

[134] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. *MICRO 24,* November 1991.

[135] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. *Proceedings of ISCA'93.*

[136] Olivier Zendra, Dominique Colnet, Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables: The Small Eiffel Compiler. *OOPSLA '97 Conference Proceedings*, pp. 125-141, October 1997

Appendix A Polymorphic call sequences in assembly

Table A-2 to 8 show the call code sequence of all techniques. These are expressed as assembly code sequences in the abstract instruction set of Table A-1. This instruction set captures the functionality of a generic load/store processor architecture [65].

R1	a register (any argument without #)
#immediate	an immediate value (prefix #)
load [R1+#imm], R2	load the word in memory location R1+#imm to register R2
store R1,[R2+#imm]	store the word in register R1 into memory location R2+#imm
setlo #imm, R1	set the least significant part of register R1 to #imm, the rest to 0
sethi #imm, R1	set the most significant part of R1 to #imm[a]
xor R1, R2, R3	bit-wise xor on register R1 and R2[b]. Result is put in R3
and R1, R2, R3	bit-wise and on register R1 and R2. Result is put in R3
add R1, R2, R3	add register R1 to R2 and put result in R3
call R1	jump to address in R1 (can also be immediate), saves return address
comp R1, R2	compare value in register R1 with R2 (R2 can be immediate)
bne #imm	if last compare is not equal, jump to #imm[c]
jump R1	jump to address in R1 (can also be immediate)

Table A-1. Abstract instruction set

[a] sethi always occurs after a setlo in our code. We use the same string to indicate the upper and lower part of #imm, to indicate which instructions depend on the bit length of a particular value.
[b] Operands of arithmetic and logic instructions can be immediates, if the bit length permits.
[c] PC-relative. We do not go into such details, unless they affect code size and/or speed.

The first column (I) of every table numbers the call code instructions of the second column (Assembly Code). Instructions are ordered as if they are executed on a processor without pipelining. Numbers in superscript indicate control- or data dependencies (for example, in Table A-3, instruction $11^{1,8}$ uses values produced by instruction 1 and 8). Every technique is then scheduled for best performance on P92 and P97. The *Time* column specifies, for each instruction, the earliest possible cycle it can be executed. For P92, each branch has a delay slot. If there is no instruction available to be fitted in the slot, one cycle is lost. Columns I1-4 represent the 4-way instruction issue of P97. This processor has no delay slots. Therefore each unpredicted branch causes a branch penalty (B). At most two memory operations and one branch can be executed in a cycle, and we assumed that a branch can be executed in the same cycle as the comparison it depends on[1]. Unconditional branches may disappear due to *branch folding* (see [65]), but we show them for clarity in the schedule.

Instructions in *italic* take care of multiple inheritance (only in Table A-2). Instructions in **bold** implement dynamic typing.

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #tableOffset], table	1	0	1	-	-	-	0
2^{1}	load [table + #selectorOffset], method	2	L	2	*3^{1}*	-	-	L
3^{1}	*load [table + #deltaOffset], delta*	*3*	L+1	5	*4^{3}*	-	-	2L
4^{3}	*add object, delta, object*	5	2L	6	-	-	-	2L+1+B
5^{2}	call method	*4*	2L+1					
		6	2L+2					
$6^{5,4}$	<first instruction of target>							

Table A-2. VTBL call code schedule

[1] This is not the case for all current processors in the P97 class. However, some architectures provide compare-and-branch instructions, which generate similar schedules. Also, the net effect of not being able to executed a compare and branch simultanously on P97 is one extra cycle for most techniques. In the PIC and CT cases, the compare instructions of the repeated tests can be scheduled one cycle more in advance than shown (branch x scheduled with test x+1), resulting in a total overhead of one cycle.

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #classOffset], class	1	0	1	2	3	4	0
2	setlo #cacheAddr, cache	2	1	5	-	-		L
3	sethi #cacheAddr, cache	3	2	6	-	-	-	1+L
4	setlo #selectorCode, selector	4	3	7	-	-	-	2+L
$5^{1,4}$	xor class, selector, index	5	max(L,4)	8	9	-	-	3+L
6^{5}	and index, #mask, index	6	1+max(L,4)	10	11	12	-	3+2L
$7^{3,6}$	add cache, index, cache	7	2+max(L,4)	13	14	-	-	4+2L
8^{7}	load [cache], cacheClass	8	3+max(L,4)	15	-	-	-	5+2L
9^{7}	load [cache + 4], cacheSelector	9	4+max(L,4)	16				6+2L+B
10^{7}	load [cache + 8], cacheTarget	10	5+max(L,4)					
$11^{1,8}$	comp class, cacheClass	11	3+max(2L,7)					
12^{11}	bne #miss	12	4+max(2L,7)					
$13^{2,9}$	comp selector, cacheSelector	13	5+max(2L,7)					
14^{13}	bne #miss	14	6+max(2L,7)					
15^{10}	call cacheTarget	15	7+max(2L,7)					
		-	8+max(2L,7)					
16^{15}	<first instruction of target>	16	9+max(2L,7)					

Table A-3. LC call code schedule

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #tableOffset], table	1	0	1	-	-	-	0
2^{1}	load [table + #colorOffset], method	2	L	2	**3**	-	-	L
3	**setlo #selector, selector**	**3**	**1+L**	4	-	-	-	2L
4^{2}	call method	4	2L	**5**	**6**	-	-	**1+2L+B**
		-	1+2L	7	-	-	-	2+2L+B
$\mathbf{5^{3,4}}$	**comp selector, #methSelector**	**5**	**2+2L**					
$\mathbf{6^{5}}$	**bne #messageNotUnderstood**	**6**	**3+2L**					
7^{6}	<first instruction of target>	7	4+2L					

Table A-4. SC call code schedule

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #classOffset], actualClass	1	0	1	2	3	-	0
2	setlo #class, predictedClass	2	1	4	5	-	-	L
3	call #method[a]	3	2	6	-	-	-	1+L
		4	max(3,L)					
$4^{1,2,3}$	comp actualClass, predictedClass	5	1+max(3,L)					
5^{4}	bne #inlineCacheMiss	6	2+max(3,L)					
6^{5}	<first instruction of target>							

Table A-5. IC call code schedule

[a] The message selector is placed right after this call instruction. This doesn't have any effect on speed, but does increase the code size by one word.

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #classOffset], actualClass	1	0	1	2	-	-	0
2	call #PIC_nnn	2	1	3	4	-	-	L
		3	L	5	-	-	-	(1+pB)[a]+L
3^{1}	comp actualclass, #predictedClass1	4	1+L	6	-	-	-	1+(1+pB)+L
4^{3}	bne #2nd	7	2+L	7	8	-	-	(1+pB)+L
5^{4}	jump #method1	5	3+L	9	-	-	-	2*(1+pB)+L
7^{4}	2nd: comp actualClass, #predictedClass2	6	4+L	10	-	-	-	1+2*(1+pB)+L
8^{7}	bne #3rd	8	3+L					
9^{8}	jump #method2	11	4+L	4i-1	4i	-	-	(i-1)*(1+pB)+L
...	...	9	5+L	4i+1	-	-	-	i*(1+pB)+L
4i-1	ith: comp actualClass, #predictedClassi	10	6+L	4i+2	-	-	-	1+i*(1+pB)+L
4i	bne #(i+1)th		...					
4i+1	jump #methodi	4i	-1+i*2+L					
...	...	4i+1	1+i*2+L					
4n-1	last: jump #PICmiss	4i+2	2+i*2+L					
6^{5}	<first instruction of method 1>							
10^{9}	<first instruction of method 2>							
4i+2	<first instruction of method i>							

Table A-6. PIC call code schedule

[a] A mispredicted conditional branch adds 1+B cycles, a predicted branch adds 1 cycle, p = average branch misprediction rate.

		P92		P97				
I	Assembly Code	I1	Time	I1	I2	I3	I4	Time
1	load [object + #classoffset], class	1	0	1	2	3	-	0
2	setlo #selector, selector	2	1	4	-	-	-	L
3	sethi #selector, selector[a]	3	2	5	-	-	-	1+L
4[1,3]	add class, selector, table	4	max(3,L)	6	-	-	-	1+2L
5[4]	load [table], method	5	1+max(3,L)	**7**	**8**	**-**	**-**	**2+2L+B**
6[5]	call method	6	1+L+max(3,L)	**9**	**10**	**-**	**-**	**3+2L+B**
		-	2+L+max(3,L)	11	-	-	-	4+2L+B
7[6]	**sethi #methSelector, thisSelector**	**7**	**3+L+max(3,L)**					
8[6]	**setlo #methSelector, thisSelector**	**8**	**4+L+max(3,L)**					
9[8]	**comp selector, thisSelector**	**9**	**5+L+max(3,L)**					
10[9]	**bne #messageNotUnderstood**	**10**	**6+L+max(3,L)**					
11[10]	<first instruction of target>	11	7+L+max(3,L)					

Table A-7. RD call code schedule

[a] selector offset is unlikely to fit in immediate field of instructions in large applications.

	I	Assembly Code	P92 I1	P92 Time	P97 I1	P97 I2	P97 I3	P97 I4	P97 Time
callsite	1	load [object + #classOffset], class	1	0	1	-	-	-	0
	2^{1}	load [class + #tableOffset], table	2	L	2	3	-	-	L
	3^{1}	load [class + #cidOffset], cid	3	1+L	4	-	-	-	2L
	4^{2}	load [table + #selector], method	4	2L	5	-	-	-	3L
	5^{4}	call method	5	3L					
			-	1+3L					
single-implementation	**6^{5}**	**setlo #mask, mask**	**6**	**1+1+3L**	**6**	**7**	-	-	**1+3L+B**
	7^{5}	**sethi #mask, mask**	**7**	**2+1+3L**	**8**	**9**	**10**	-	**2+3L+B**
	8^{7}	**and mask, cid, cid**	**8**	**3+1+3L**	**11**	**12**	-	-	**3+3L+B**
	9^{5}	**setlo #thisClass, thisClass**	**9**	**4+1+3L**	13	-	-	-	**3**+1+3L+B
	10^{5}	**sethi #thisClass, thisClass**	**10**	**5+1+3L**					
	$11^{8,10}$	**comp thisClass, cid**	**11**	**6+1+3L**					
	12^{11}	**bne #messageNotUnderstood**	**12**	**7+1+3L**					
	13^{12}	<first instruction of target>	13	**7**+2+3L					
overloaded	14^{5}	setlo #mask1, mask1	14	1+1+3L	14	15	-	-	1+3L+B
	15^{5}	sethi #mask1, mask1	15	2+1+3L	16	17	18	-	2+3L+B
	16^{15}	and mask1, cid, temp	16	3+1+3L	19	20	-	-	1+2+3L+B
	17^{5}	setlo #cid1, cid1	17	4+1+3L	21	-	-	-	1+(3+pB)[a]+3L+B
	18^{5}	sethi #cid1, cid1	18	5+1+3L	22	-	-	-	2+(3+pB)+3L+B
	$19^{16,18}$	comp cid1, temp	19	6+1+3L	23	24	-	-	1+(3+pB)+3L+B
	20^{19}	bne #2nd	20	7+1+3L	25	26	27	-	1+2+(3+pB)+3L+B
	21^{20}	jump #method1	23	1+7+1+3L	28	29	-	-	1+2+(3+pB)+3L+B
	23^{20}	2nd: setlo #mask2, mask2	21	7+2+3L	30	-	-	-	1+2*(3+pB)+3L+B
	24^{20}	sethi #mask2, mask2	22	7+3+3L	31	-	-	-	2+2*(3+pB)+3L+B
	25^{24}	and mask2, cid, temp	24	2+7+1+3L	...				
	26^{20}	setlo #cid2, cid2	25	3+7+1+3L	13+9n		-	-	2+n*(3+pB)+3L+B
	27^{20}	sethi #cid2, cid2	26	4+7+1+3L					
	$28^{25,27}$	comp cid2, temp	27	5+7+1+3L					
	29^{28}	bne #3nd	28	6+7+1+3L					
	30^{29}	jump #method2	29	7+7+1+3L					
	32^{29}	3rd: ...	32	1+2*7+1+3L					
			30	2*7+2+3L					
	22^{21}	<first instruction of method 1>	31	2*7+3+3L					
				...					
	31^{30}	<first instruction of method 2>	13+9i	i*7+3+3L					
	13+9i	<first instruction of method i>							

Table A-8. CT call code schedule

[a] A mispredicted conditional branch adds 1+B cycles, a predicted branch adds 1 cycle, p = average branch misprediction rate.

Table A-9 shows the average dispatch time, in cycles, for all techniques, for the parameter values of Tables 7 and 8. For each generic processor, both statically typed and dynamically typed versions of a technique are shown. In the dynamically cached cases, these are always equal. This table is shown as a graph in Figure 27 in the main text.

	P92		P95		P97	
	static	dyn.	static	dyn.	static	dyn.
LC	20.7		17.8		21.1	
VTBL	6.0	N/A	8.0	N/A	13.0	N/A
VTBL-MI	6.0	N/A	8.0	N/A	13.0	N/A
SC	6.0	8.0	8.0	10.0	13.0	14.0
RD	8.0	12.0	9.0	11.0	14.0	16.0
CT	8.0	15.0	10.0	14.0	16.0	19.0
IC	9.8		7.1		7.8	
PIC	8.8		6.3		7.5	

Table A-9. Dispatch timings (in cycles)

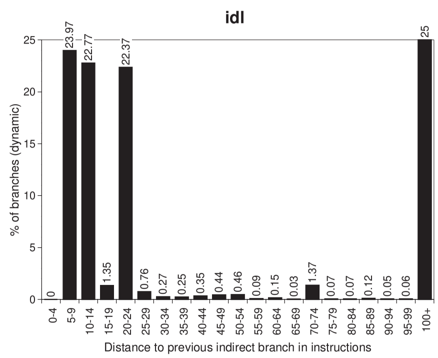
Table A-10 shows an approximation of the space cost, of all techniques, for the Smalltalk image (VisualWorks 1.0). Part of this table is shown as a graph in Figure 18 in the main text.

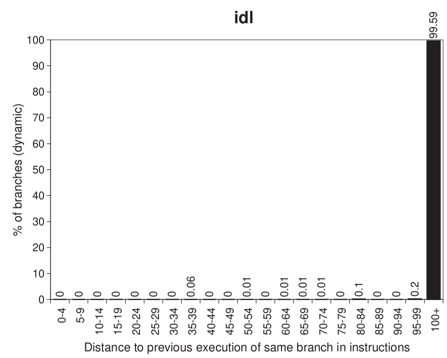
	single inheritance					
	static typing			dynamic typing		
	code	data	sum	code	data	sum
DTS	274	89	363	same as SI-ST		363
LC	1,916	137	2,053	same as SI-ST		2,053
IC	477	137	614	same as SI-ST		614
PIC	477	231	708	same as SI-ST		708
VTBL	274	696	970	N/A		
SC	274	1,219	1,493	341	1,219	1,560
RD	684	703	1,387	817	703	1,520
CT	548	107	655	782	107	889

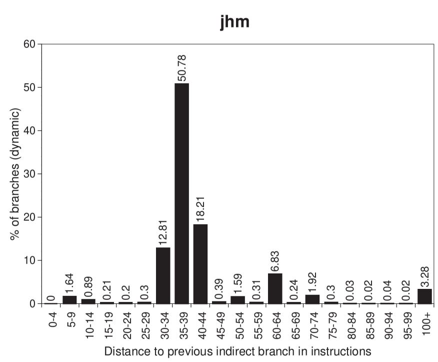
Table A-10. Approximate space cost for dispatch in Smalltalk image (in Kbytes)

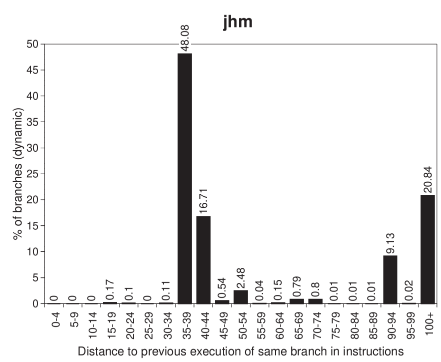
Appendix B Indirect branch execution intervals

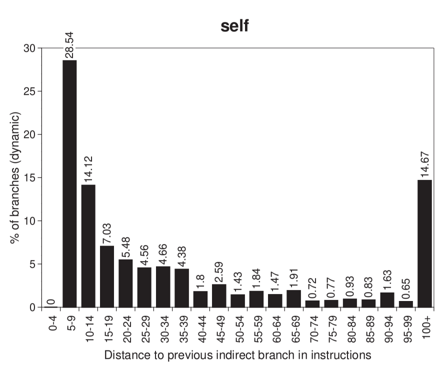
Below we show the number of instructions occuring between indirect branches. Histograms show the percentage of total branches executed that fall in each category of five instructions. The last category, 100+, groups all branches that occur more than 100 instructions after the last branch. For each benchmark, the left histogram shows the distance between indirect branch executions, and the right histogram shows the distance between executions of the same branch.

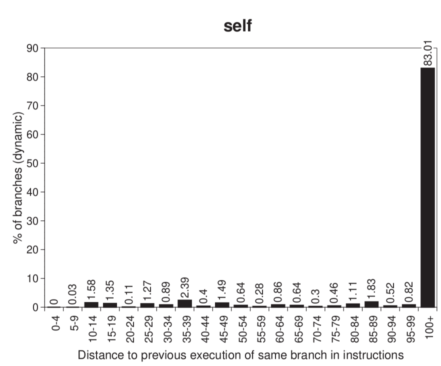


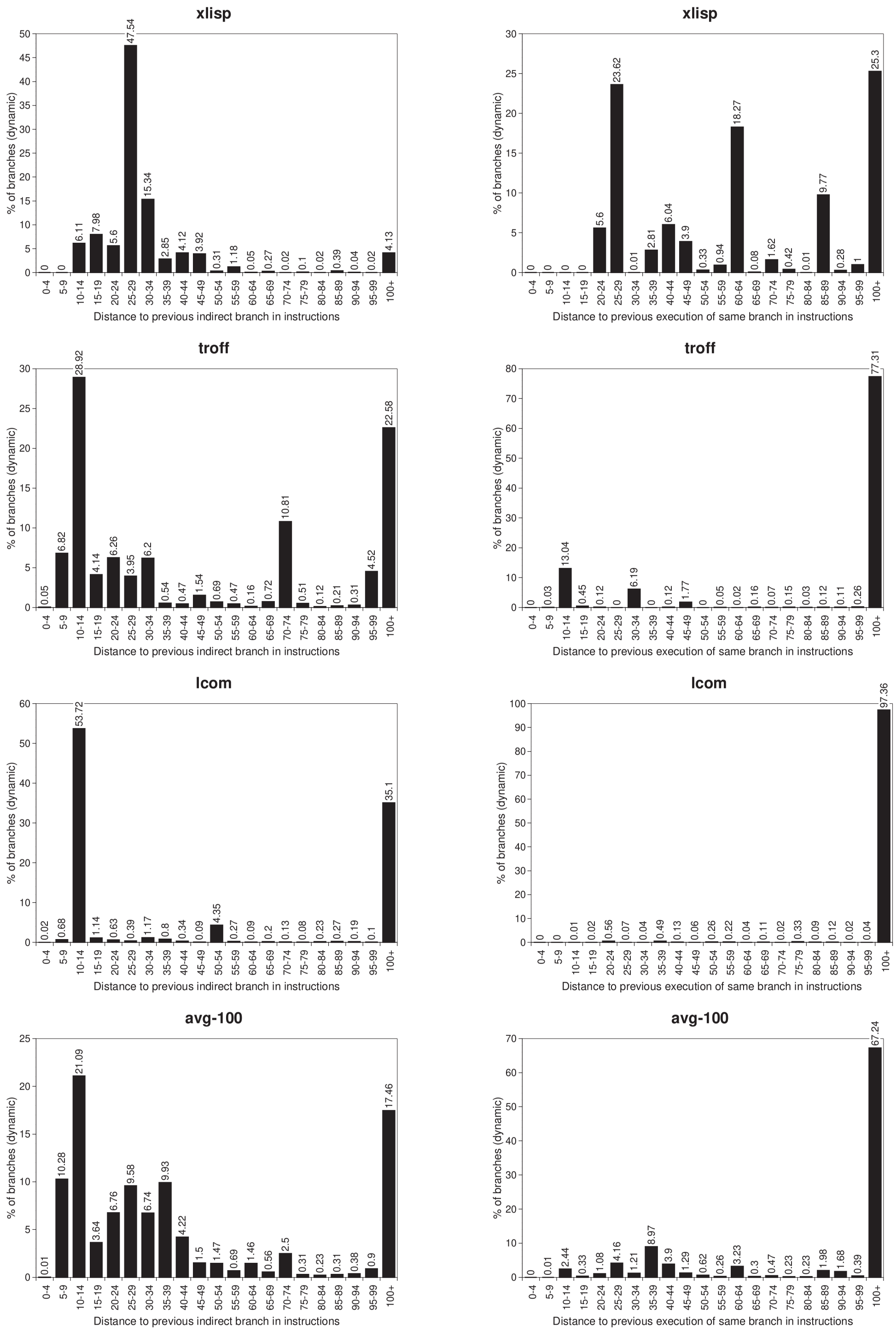


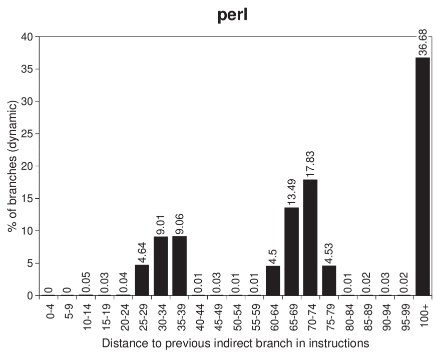


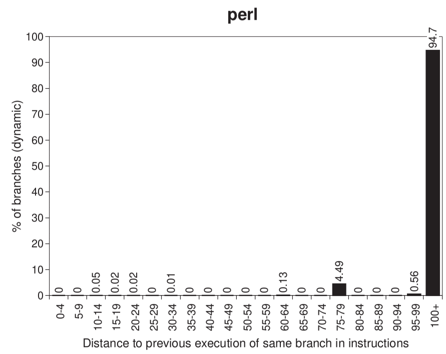


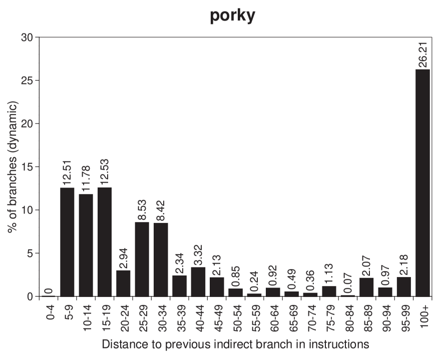


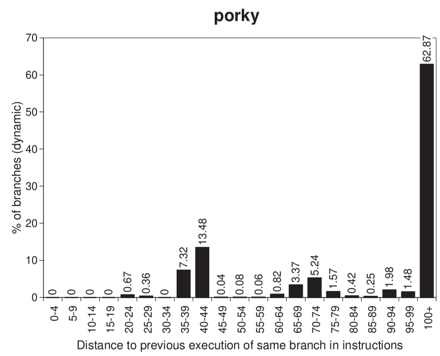


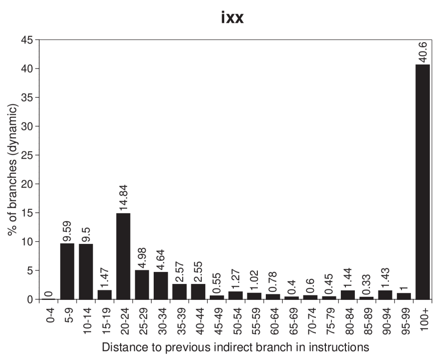

xlisp
% of branches (dynamic)
Distance to previous indirect branch in instructions
xlisp
% of branches (dynamic)
Distance to previous execution of same branch in instructions
troff
% of branches (dynamic)
Distance to previous indirect branch in instructions
troff
% of branches (dynamic)
Distance to previous execution of same branch in instructions
lcom
% of branches (dynamic)
Distance to previous indirect branch in instructions
lcom
% of branches (dynamic)
Distance to previous execution of same branch in instructions
avg-100
% of branches (dynamic)
Distance to previous indirect branch in instructions
avg-100
% of branches (dynamic)
Distance to previous execution of same branch in instructions

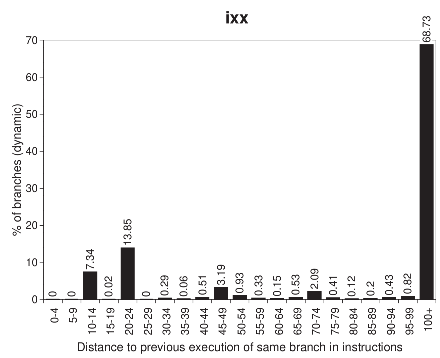

perl
% of branches (dynamic)
Distance to previous indirect branch in instructions

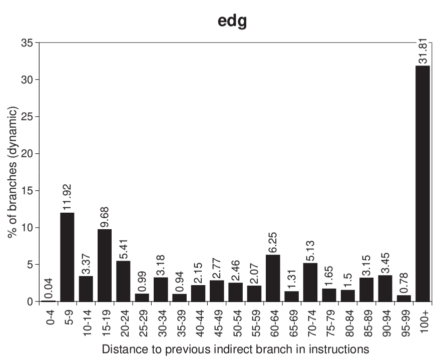

perl
% of branches (dynamic)
Distance to previous execution of same branch in instructions

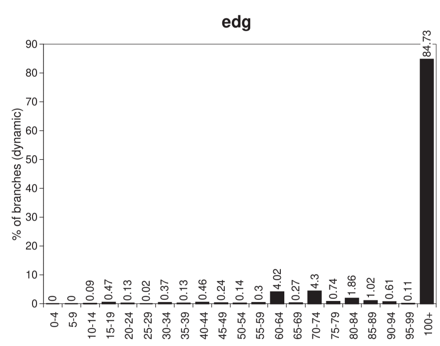

porky
% of branches (dynamic)
Distance to previous indirect branch in instructions

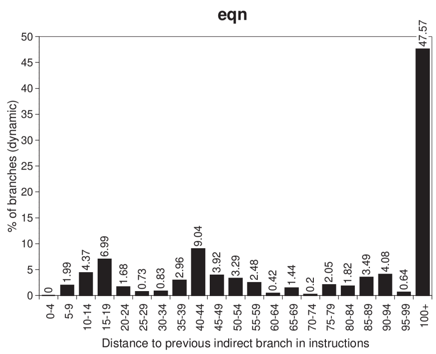

porky
% of branches (dynamic)
Distance to previous execution of same branch in instructions

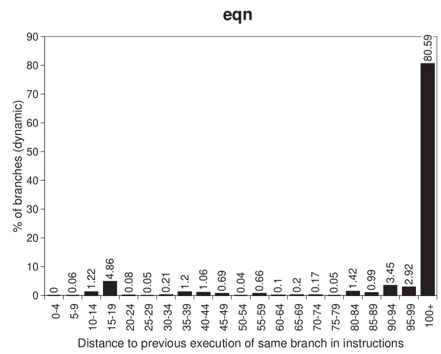

ixx
% of branches (dynamic)
Distance to previous indirect branch in instructions

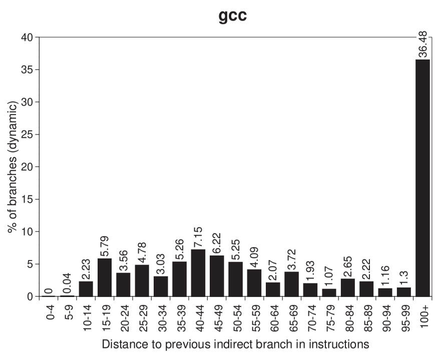

ixx
% of branches (dynamic)
Distance to previous execution of same branch in instructions

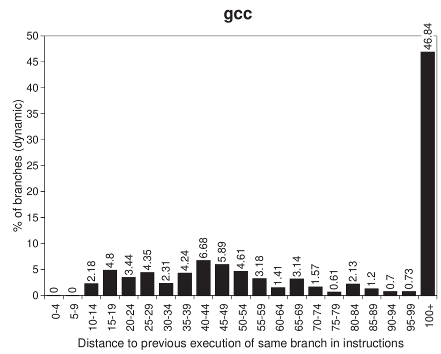

edg
% of branches (dynamic)
Distance to previous indirect branch in instructions

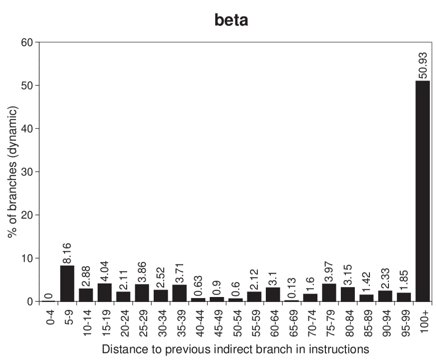

edg
% of branches (dynamic)
Distance to previous execution of same branch in instructions

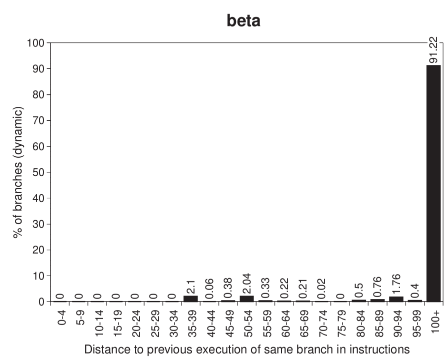

eqn
% of branches (dynamic)
Distance to previous indirect branch in instructions
0-4 5-9 10-14 15-19 20-24 25-29 30-34 35-39 40-44 45-49 50-54 55-59 60-64 65-69 70-74 75-79 80-84 85-89 90-94 95-99 100+
0 1.99 4.37 6.99 1.68 0.73 0.83 2.96 9.04 3.92 3.29 2.48 0.42 1.44 0.2 2.05 1.82 3.49 4.08 0.64 47.57

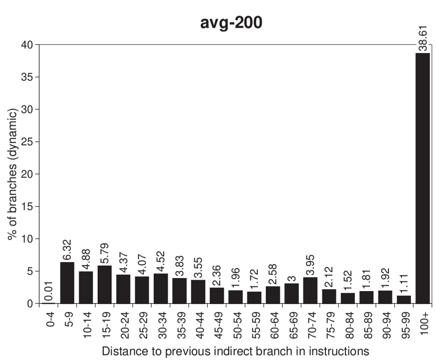

eqn
% of branches (dynamic)
Distance to previous execution of same branch in instructions
0 0.06 1.22 4.86 0.08 0.05 0.21 1.2 1.06 0.69 0.04 0.66 0.1 0.2 0.17 0.05 1.42 0.99 3.45 2.92 80.59

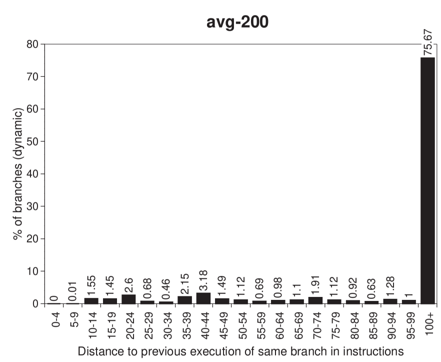

gcc
% of branches (dynamic)
Distance to previous indirect branch in instructions
0 0.04 2.23 5.79 3.56 4.78 3.03 5.26 7.15 6.22 5.25 4.09 2.07 3.72 1.93 1.07 2.65 2.22 1.16 1.3 36.48

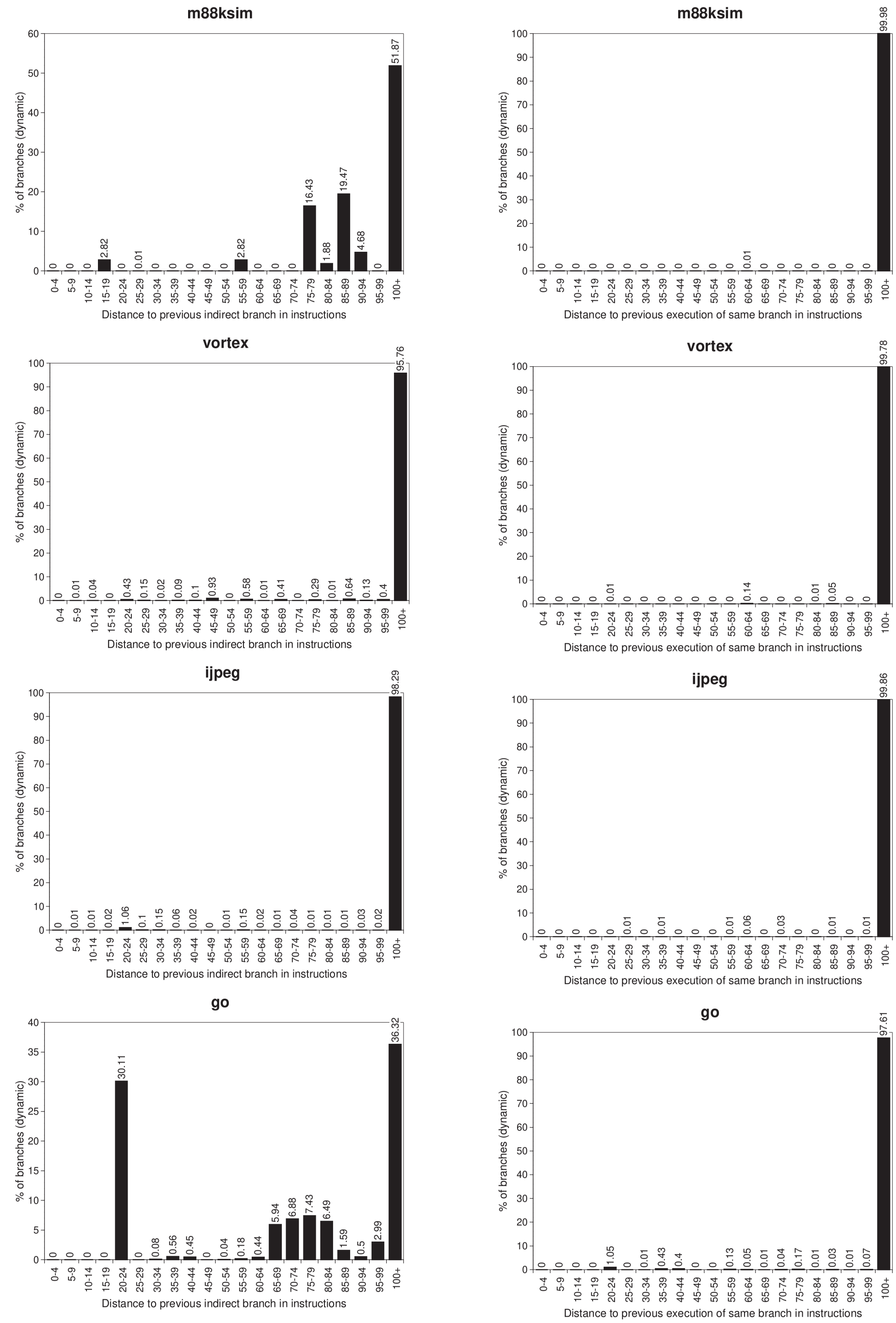

gcc
% of branches (dynamic)
Distance to previous execution of same branch in instructions
0 0 2.18 4.8 3.44 4.35 2.31 4.24 6.68 5.89 4.61 3.18 1.41 3.14 1.57 0.61 2.13 1.2 0.7 0.73 46.84


beta
% of branches (dynamic)
Distance to previous indirect branch in instructions
0 8.16 2.88 4.04 2.11 3.86 2.52 3.71 0.63 0.9 0.6 2.12 3.1 0.13 1.6 3.97 3.15 1.42 2.33 1.85 50.93


beta
% of branches (dynamic)
Distance to previous execution of same branch in instructions
0 0 0 0 0 0 0 2.1 0.06 0.38 2.04 0.33 0.22 0.21 0.02 0 0.5 0.76 1.76 0.4 91.22


avg-200
% of branches (dynamic)
Distance to previous indirect branch in instructions
0.01 6.32 4.88 5.79 4.37 4.07 4.52 3.83 3.55 2.36 1.96 1.72 2.58 3 3.95 2.12 1.52 1.81 1.92 1.11 38.61


avg-200
% of branches (dynamic)
Distance to previous execution of same branch in instructions
0 0.01 1.55 1.45 2.6 0.68 0.46 2.15 3.18 1.49 1.12 0.69 0.98 1.1 1.91 1.12 0.92 0.63 1.28 1 75.67


m88ksim
% of branches (dynamic)
Distance to previous indirect branch in instructions
m88ksim
% of branches (dynamic)
Distance to previous execution of same branch in instructions
vortex
% of branches (dynamic)
Distance to previous indirect branch in instructions
vortex
% of branches (dynamic)
Distance to previous execution of same branch in instructions
ijpeg
% of branches (dynamic)
Distance to previous indirect branch in instructions
ijpeg
% of branches (dynamic)
Distance to previous execution of same branch in instructions
go
% of branches (dynamic)
Distance to previous indirect branch in instructions
go
% of branches (dynamic)
Distance to previous execution of same branch in instructions

186