ADAPTIVE OPTIMIZATION FOR SELF: RECONCILING HIGH PERFORMANCE WITH EXPLORATORY PROGRAMMING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Urs Hölzle

August 1994

© Copyright by Urs Hölzle 1994
All rights reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David M. Ungar (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John L. Hennessy (Co-Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

L. Peter Deutsch

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell

Approved for the University Committee on Graduate Studies:

Abstract

Object-oriented programming languages confer many benefits, including abstraction, which lets the programmer hide the details of an object's implementation from the object's clients. Unfortunately, crossing abstraction boundaries often incurs a substantial run-time overhead in the form of frequent procedure calls. Thus, pervasive use of abstraction, while desirable from a design standpoint, may be impractical when it leads to inefficient programs.

Aggressive compiler optimizations can reduce the overhead of abstraction. However, the long compilation times introduced by optimizing compilers delay the programming environment's responses to changes in the program. Furthermore, optimization also conflicts with source-level debugging. Thus, programmers are caught on the horns of two dilemmas: they have to choose between abstraction and efficiency, and between responsive programming environments and efficiency. This dissertation shows how to reconcile these seemingly contradictory goals by performing optimizations *lazily.*

Four new techniques work together to achieve high performance and high responsiveness:

- *Type feedback* achieves high performance by allowing the compiler to inline message sends based on information extracted from the runtime system. On average, programs run 1.5 times faster than the previous SELF system; compared to a commercial Smalltalk implementation, two medium-sized benchmarks run about three times faster. This level of performance is obtained with a compiler that is both simpler and faster than previous SELF compilers.
- *Adaptive optimization* achieves high responsiveness without sacrificing performance by using a fast non-optimizing compiler to generate initial code while automatically recompiling heavily used parts of the program with an optimizing compiler. On a previous-generation workstation like the SPARCstation-2, fewer than 200 pauses exceeded 200 ms during a 50-minute interaction, and 21 pauses exceeded one second. On a current-generation workstation, only 13 pauses exceed 400 ms.
- *Dynamic deoptimization* shields the programmer from the complexity of debugging optimized code by transparently recreating non-optimized code as needed. No matter whether a program is optimized or not, it can always be stopped, inspected, and single-stepped. Compared to previous approaches, deoptimization allows more debugging while placing fewer restrictions on the optimizations that can be performed.
- *Polymorphic inline caching* generates type-case sequences on-the-fly to speed up messages sent from the same call site to several different types of object. More significantly, they collect concrete type information for the optimizing compiler.

With better performance yet good interactive behavior, these techniques make exploratory programming possible both for pure object-oriented languages and for application domains requiring higher ultimate performance, reconciling exploratory programming, ubiquitous abstraction, and high performance.

Acknowledgments

No Ph. D. thesis can be written without the help of others, and I am deeply indebted to the people who made this dissertation possible. I was fortunate to find a great advisor in David Ungar, who taught me how to do research and, equally important, how to write about it. He always asked the right questions early on and persisted until I had an answer. I would also like to thank my dissertation readers, Peter Deutsch, John Hennessy, and John Mitchell, for their advice and their comments on drafts of this thesis. It was Peter and Dave's work on Smalltalk implementations that originally sparked my interest in the field many years ago.

Craig Chambers and Bay-Wei Chang helped me get started on SELF by patiently answering many questions when I joined the SELF group in 1989. Together with the more recent members of the project—Ole Agesen, Lars Bak, John Maloney, Randy Smith, and Mario Wolczko—they made sure we never ran out of topics to discuss, and created the unique atmosphere that made working on SELF so much fun. I will miss you, guys.

I have learned much from many interesting discussions with others, including Walter Bischofberger, Günther Blaschek, Bob Cmelik, Amer Diwan, Claus Gittinger, Robert Griesemer, Görel Hedin, Barry Hayes, David Keppel, Peter Kessler, Ole Lehrmann Madsen, Boris Magnusson, Eliot Moss, Stephan Murer, Jens Palsberg, Peter Schnorf, Michael Schwartzbach, Paul Wilson, and Ben Zorn. I would also like to thank Robert Griesemer for his comments on earlier drafts and for proofreading the final draft.

Many others in CIS, at Stanford, and at Sun made my stay here so enjoyable, as well as my friends of the Dinner Club and DINAC (Dessert Is Never A Chore) whom I have to thank for many pleasant evenings. Last but not least, I am deeply indebted to Geeske who changed my life much more than graduate study ever could.

Finally, I wish to thank all the organizations who generously provided financial support, especially Sun Microsystems Laboratories, the Swiss Nationalfonds, and the Fulbright program.

Table of Contents

Table of Contents ix
List of Figures xiii
List of Tables xvii

1. Introduction 1

2. Background and related work 3
2.1 The SELF language 3
2.2 The SELF system 4
2.2.1 Overview of the implementation 4
2.2.2 Efficiency 5
2.2.3 Source-level semantics 6
2.2.4 Direct execution semantics 7
2.3 Overview of the compilation process 7
2.4 Benefits of this work 8
2.5 Related work 9
2.5.1 Dynamic compilation 9
2.5.2 Customization 9
2.5.3 Previous SELF compilers 10
2.5.4 Smalltalk-80 compilers 11
2.5.4.1 The Deutsch-Schiffman system 11
2.5.4.2 SOAR 11
2.5.4.3 Other Smalltalk compilers 11

3. Polymorphic inline caching 13
3.1 Out-of-line lookup caches 13
3.2 Dispatch tables 13
3.3 Inline caches 13
3.4 Handling polymorphic sends 15
3.5 Polymorphic inline caches 16
3.5.1 Variations 17
3.6 Implementation and results 18
3.6.1 Execution time 19
3.6.2 Space overhead 21
3.7 Summary 21

4. The non-inlining compiler 23
4.1 Simple code generation 23
4.2 Compilation speed 24
4.3 Execution speed 25
4.3.1 Overview 26
4.3.2 Lookup cache misses 27
4.3.3 Blocks, primitive calls, and garbage collection 27
4.3.4 Register windows 27

4.3.5 Instruction cache misses 28
4.4 NIC vs. Deutsch-Schiffman Smalltalk 28
4.4.1 Smalltalk without hardwired control structures 28
4.4.2 A NIC with inlined control structures 29
4.5 Predicting the performance of a SELF interpreter 31
4.6 The NIC and interactive performance 32
4.7 Summary 32

5. Type feedback and adaptive recompilation 35
5.1 Type feedback 36
5.2 Adaptive recompilation 37
5.3 When to recompile 38
5.4 What to recompile 39
5.4.1 Overview of the recompilation process 39
5.4.2 Selecting the method to be recompiled 40
5.5 When to use type feedback information 41
5.6 Adding type feedback to a conventional system 41
5.7 Applicability to other languages 42
5.8 Related work 43
5.8.1 Static type prediction 43
5.8.2 Customization 43
5.8.3 Whole-program optimizations 44
5.8.4 Inlining 44
5.8.5 Profile-based compilation 45
5.9 Summary 45

6. An optimizing SELF compiler 47
6.1 The front end 47
6.1.1 Finding receiver types 47
6.1.2 Code-based inlining heuristics 51
6.1.3 Splitting 52
6.1.4 Uncommon traps 53
6.1.4.1 Ignoring uncommon cases helps 53
6.1.4.2 Uncommon branch extensions 56
6.1.4.3 Uncommon traps and recompilation 57
6.2 The back end 58
6.2.1 Intermediate code format 58
6.2.2 Computing exposed blocks 59
6.2.3 Def / use information 59
6.2.4 Copy propagation 60
6.2.5 Register allocation 60
6.3 Runtime system issues 61
6.3.1 Type tests 61
6.3.2 Store checks 62
6.3.3 Block zapping 63
6.4 What's missing 64

6.4.1 Peephole optimization 65
6.4.2 Repeated type tests 65
6.4.3 Naive register allocation 65
6.5 Summary 66

7. Performance evaluation 67
7.1 Methodology 68
7.2 Type analysis does not improve performance of object-oriented programs 69
7.3 Type feedback works 71
7.3.1 Comparison to other systems 74
7.3.2 Different inputs 74
7.4 Stability of performance 75
7.4.1 Type analysis exhibits unstable performance 75
7.4.2 Static type prediction can fail 77
7.4.2.1 Non-predicted messages are much slower 77
7.4.2.2 Mispredictions can be costly 78
7.4.2.3 Performance variations in high-level languages 78
7.5 Detailed performance analysis 79
7.5.1 Type feedback eliminates type tests 79
7.5.2 Type feedback reduces the cost of type tests 80
7.5.3 Type feedback reduces the time spent in type tests 82
7.5.4 Type analysis vs. type feedback 84
7.5.5 Reoptimization effectiveness 86
7.5.6 Size of compiled code 87
7.6 Summary 87

8. Hardware’s impact on performance 89
8.1 Instruction usage 89
8.2 Register windows 93
8.3 Hardware support for tagged arithmetic 95
8.4 Instruction cache behavior 98
8.5 Data cache behavior 100
8.6 Possible improvements 103
8.7 Conclusions 104

9. Responsiveness 105
9.1 Pause clustering 105
9.2 Compile pauses 106
9.2.1 Pauses during an interactive session 106
9.2.2 Pauses on faster systems 109
9.3 Starting new code 110
9.4 Performance variations 112
9.4.1 Start-up behavior of large programs 113
9.4.2 Performance stability 115
9.4.3 Influence of configuration parameters on performance variation 118
9.4.4 Influence of configuration parameters on final performance 118

9.5 Compilation speed 120
9.6 Summary 122

10. Debugging optimized code 125
10.1 Optimization vs. debugging 125
10.1.1 Displaying the stack 125
10.1.2 Single-stepping 126
10.1.3 Changing the value of a variable 126
10.1.4 Changing a procedure 126
10.2 Deoptimization 126
10.2.1 Recovering the unoptimized state 127
10.2.2 The transformation function 128
10.2.3 Lazy deoptimization 129
10.2.4 Interrupt points 130
10.3 Updating active methods 131
10.4 Common debugging operations 131
10.4.1 Single-step 132
10.4.2 Finish 132
10.4.3 Next 132
10.4.4 Breakpoints and watchpoints 132
10.5 Discussion 132
10.5.1 Benefits 132
10.5.2 Current limitations 133
10.5.3 Generality 133
10.6 Implementation cost 134
10.6.1 Impact on responsiveness 134
10.6.2 Impact on runtime performance 134
10.6.3 Memory usage 134
10.7 Related work 136
10.8 Conclusions 137

11. Conclusions 139
11.1 Applicability 140
11.2 Future work 140
11.3 Summary 141
Glossary 143
Detailed Data 145
References 157
References 159

List of Figures

Figure 2-1. Customization 5
Figure 2-3. Splitting 6
Figure 2-2. Type prediction for the expression i + 1 6
Figure 2-4. Overview of dynamic recompilation in SELF-93....................7
Figure 3-2. Inline cache after first send 14
Figure 3-1. Empty inline cache....................14
Figure 3-3. Inline cache miss....................15
Figure 3-4. Size distribution (degree of polymorphism) of inline caches 16
Figure 3-5. Polymorphic inline cache....................17
Figure 3-6. Execution time saved by polymorphic inline caches 19
Figure 3-7. Polymorphism vs. miss ratio....................20
Figure 4-1. Profile of NIC compilation....................24
Figure 4-2. NIC compile time as a function of source method size 25
Figure 4-3. Histogram of NIC compile time....................26
Figure 4-4. Timeline of interactive session ("cold start" scenario) 33
Figure 4-5. Timeline of interactive session ("warm start" scenario) 34
Figure 5-1. SELF-93 compilation process....................37
Figure 5-2. Overview of the recompilation process....................39
Figure 5-3. Type feedback in a statically compiled system....................42
Figure 6-1. Organization of the optimizing SELF-93 compiler 48
Figure 6-2. Class hierarchy of expressions....................49
Figure 6-3. Finding type feedback information (I) 49
Figure 6-4. Finding type feedback information (II)....................50
Figure 6-5. Finding type feedback information (III) 50
Figure 6-6. Splitting 53
Figure 6-7. Code handling all cases....................54
Figure 6-8. Code handling only common cases....................55
Figure 6-9. Sequence of events when handling uncommon traps 57
Figure 7-1. Execution speed of SELF-91 relative to SELF-93-nofeedback 70
Figure 7-2. Number of sends in SELF-91 relative to SELF-93-nofeedback 70
Figure 7-3. Execution speed of SELF-93 relative to SELF-93-nofeedback....................71
Figure 7-4. Calls performed by SELF-93 relative to SELF-93-nofeedback....................71
Figure 7-5. Call frequency relative to unoptimized SELF....................72
Figure 7-6. Call frequency reduction vs. execution time reduction....................72
Figure 7-7. Reasons for SELF-93's improved performance 73
Figure 7-8. SELF-93 execution speed compared to other languages 74
Figure 7-9. Performance of SELF-91 on Stanford benchmarks....................76
Figure 7-10. Performance of SELF-93 on Stanford benchmarks....................76
Figure 7-11. Number of type tests executed in SELF-93-nofeedback and SELF-93 79
Figure 7-12. Sends inlined per additional inline type test 81
Figure 7-13. Path length of type tests 81
Figure 7-14. Arity of type tests (dynamic averages)....................82
Figure 7-15. Execution time consumed by type tests 83
Figure 7-16. Percentage of execution time spent in type tests for SELF-93 83

Figure 7-17. Performance on the Stanford integer benchmarks 85
Figure 7-18. Analysis of inner loop of `sieve` benchmark 85
Figure 7-19. SELF-93 type testing overhead in the Stanford benchmarks 86
Figure 7-20. Code size of SELF-93 and SELF-91 88
Figure 8-1. Dynamic instruction usage of SPECint89 vs. SELF-93 benchmarks 90
Figure 8-2. Dynamic instruction usage of SPECint89 vs. SELF-93 benchmarks (excluding no-ops and unconditional branches) 91
Figure 8-3. Differences between SELF/C++ and the SPEC C programs 93
Figure 8-4. Register window overhead 94
Figure 8-5. Time overhead of instruction cache misses in SELF-93 99
Figure 8-6. Time overhead of instruction cache misses in SELF-91 99
Figure 8-7. Time overhead of data cache misses in SELF-93 (write-allocate with subblock placement) 101
Figure 8-8. Time overhead of data cache misses in SELF-93 (write-noallocate cache) 101
Figure 8-9. Write miss ratio for SELF-93 (write-noallocate cache) 102
Figure 8-10. Object allocation in SELF 102
Figure 9-1. Individual pauses and the resulting pause clusters 106
Figure 9-2. Distribution of individual vs. combined compile pauses 107
Figure 9-3. Distribution of compile pause length 107
Figure 9-4. Compile pauses during 50-minute interaction 108
Figure 9-5. Long-term clustering of compilation pauses 108
Figure 9-6. Compilation pauses on faster CPUs 110
Figure 9-7. Overview of performance development 113
Figure 9-8. Start-up phase of selected benchmarks 114
Figure 9-9. Correlation between program size and time to stabilize performance 115
Figure 9-10. Performance variations of SELF-93 116
Figure 9-11. Performance variations on subset of benchmarks (SELF-93) 116
Figure 9-12. Alternate visualization of data in Figure 9-11 117
Figure 9-13. Performance variations on subset of benchmarks (SELF-93, “worst” run) 117
Figure 9-14. Performance variations if invocation counters don’t decay 118
Figure 9-15. Influence of recompilation parameters on performance (overall) 119
Figure 9-16. Influence of recompilation parameters on performance (`CecilComp`) 120
Figure 9-17. Compilation speed of SELF-93-nofeedback 121
Figure 9-18. Compilation speed of SELF-91 121
Figure 9-19. Profile of SELF-93-nofeedback compilation 122
Figure 10-1. Displaying the stack 126
Figure 10-2. Pseudo-code declarations for scope data structures 127
Figure 10-3. Recovering the source-level state 128
Figure 10-4. Transforming an optimized stack frame into unoptimized form 129
Figure 10-5. Lazy deoptimization of stack frames 130
Figure A-1. I-cache miss ratios in SELF-93 152
Figure A-2. Data read miss ratios in SELF-93 152
Figure A-3. Data write miss ratios in SELF-93 153
Figure A-4. Data read miss ratios in SELF-93 (write-noallocate cache) 153

Figure A-5. Data write miss ratios in SELF-93 (write-noallocate cache) 154
Figure A-6. Data read miss ratios in SELF-93 (50K eden) 154

List of Tables

Table 3-1. Lookup timings (cycles on SPARCstation-2) 18
Table 3-2. Inline cache miss ratios of benchmark programs 20
Table 3-3. Space overhead of PICs 21
Table 4-1. Performance of unoptimized code relative to optimized code 26
Table 4-2. Software profile of unoptimized code 27
Table 4-3. Hardware profile of unoptimized code 27
Table 4-4. Performance of ParcPlace Smalltalk without inlined control structures 29
Table 4-5. Performance of Richards with various system configurations 30
Table 4-6. Performance of improved NIC versions relative to original NIC 30
Table 7-1. Benchmark programs 67
Table 7-2. Systems used for benchmarking 68
Table 7-3. Implementation characteristics of benchmarked systems 68
Table 7-4. Performance of long-running benchmarks 75
Table 7-5. Performance variation related to static type prediction 77
Table 7-6. Performance variations caused by failing static type prediction 78
Table 7-7. Time taken up by unoptimized code 87
Table 8-1. Differences between C++ and C (from [16]) 89
Table 8-2. Summary of main differences between SELF and SPECint89 92
Table 8-3. Arithmetic operation frequency and estimated benefit of tagged instructions 97
Table 8-4. Cache parameters of current workstations 100
Table 8-5. Sources of possible performance improvements 103
Table 9-1. Speed of workstation and PCs 109
Table 9-2. UI interaction sequence 111
Table 9-3. Configuration parameters impacting compilation pauses in SELF-93 112
Table 9-4. Start-up behavior of dynamic compilation 114
Table 10-1. Space cost of debugging information (relative to instructions) 135
Table A-1. Distribution of degree of polymorphism in call sites 145
Table A-2. Execution time saved by PICs 145
Table A-3. Execution time comparison 145
Table A-4. Number of dynamically-dispatched calls 146
Table A-5. Performance relative to C++ (all times in ms simulated time) 146
Table A-6. Performance relative to Smalltalk and Lisp 146
Table A-7. Time saved per inlined call 147
Table A-8. Time spent performing type tests 147
Table A-9. Type tests in SELF-93-nofeedback 148
Table A-10. Type tests in SELF-93 148
Table A-11. Number of comparisons per type test sequence 149
Table A-12. Performance on the Stanford integer benchmarks 149
Table A-13. SPECInt89 instruction usage (from [33]) 150
Table A-14. SELF-93 instruction usage 150
Table A-15. Instruction usage on Richards and Deltablue (C++) 151
Table A-16. Allocation behavior of benchmarks (SELF-93) 151
Table A-17. SELF-93 compiler configuration parameters 155
Table A-18. Pause length histogram data 155

1. Introduction

Object-oriented programming is becoming increasingly popular because it makes programming easier. It allows the programmer to hide implementation details from the object's clients, turning each object into an abstract data type whose operations and state can only be accessed via message sends. *Late binding* greatly enhances the power of abstract data types by allowing different implementations of the same abstract data type to be used interchangeably at runtime. That is, the code invoking an operation on an object is not aware exactly what code will be executed as a result of the invocation: late binding (also called *dynamic dispatch*) selects the appropriate implementation of the operation based on the object's exact type. Since late binding is so essential to object-oriented programming, implementations need to efficiently support it.

Ideally, object-oriented languages should use late binding even for very basic operations such instance variable access. The more pervasively encapsulation and dynamic dispatch is used, the more flexible and reusable the resulting code becomes. But unfortunately, late binding creates efficiency problems. For example, if all instance variable accesses were performed using message sends, the compiler could not translate an access to the `x` attribute of an object into a simple load instruction because some objects might implement `x` differently than others. For example, a Cartesian point might just return the value of an instance variable whereas a Polar point might compute `x` from `rho` and `theta`. This variation is precisely what late binding means: the binding of the operation `x` to the implementation (instance variable access or computation) is delayed until runtime. Therefore, the x operation has to be compiled as a *dynamically-dispatched call* (also called indirect procedure call or virtual call) that selects the appropriate implementation runtime. What could have been a one-cycle instruction has become a ten-cycle call. The increased flexibility and reusability of the source code exacts a significant run-time overhead; it seems that encapsulation and efficiency cannot coexist. This dissertation shows how to reconcile the two.

A similar efficiency problem arises from the desire to use *exploratory programming environments*. An exploratory programming environment increases programmer productivity by giving immediate feedback to all programming actions; the pause-free interaction allows the programmer to concentrate on the task at hand rather than being distracted by long compilation pauses. Traditionally, system designers have used interpreters or non-optimizing compilers in exploratory programming environments. Unfortunately, the overhead of interpretation, combined with the efficiency problems created by dynamic dispatch, slows down execution and thus limits the usefulness of such systems. This dissertation describes how to reduce the overhead of dynamic dispatch while preserving the responsiveness required for an interactive exploratory programming environment.

Our research vehicle is the object-oriented language SELF [133]. SELF's pure semantics exacerbate the implementation problems faced by object-oriented languages: in SELF, every single operation (even assignment) involves late binding. Thus, the language is an ideal test case for optimizations that reduce the overhead of late binding. Equally importantly, SELF was designed for exploratory programming. Any exploratory programming environment must provide quick turnaround after programming changes and easy debugging of partially complete programs in order to increase programmer productivity. Therefore, an optimizing compiler for such a system must not only overcome the performance problems created by dynamic dispatch but must also be compatible with interactiveness: compilation must be quick and non-intrusive, and the system must support full source-level debugging at all times.

We have implemented such a system for SELF. Among the contributions of our work are:

- A new optimization strategy, *Type Feedback*, that allows any dynamically-dispatched call to be inlined. In our example implementation for SELF, type feedback reduces the call frequency by a factor of four and improves performance by 70% compared to a system without type feedback. Despite the radically different language models of SELF and C++, the new system brings SELF to about half the performance of optimized C++ for two medium-sized object-oriented programs.

- A recompilation system that dynamically *reoptimizes* the "hot spots" of an application. The system quickly generates the initial code with a fast non-optimizing compiler, and recompiles the time-critical parts with a slower optimizing compiler. Introducing adaptive recompilation dramatically improved interactive performance in the SELF system, making it possible to combine optimizing compilation with an exploratory programming environment. Even on a previous-generation workstation like the SPARCstation-2, fewer than 200 pauses exceeded 200 ms during a 50-minute interaction, and 21 pauses exceeded one second.
- An extension to inline caching, *polymorphic inline caching*, that speeds up dynamically-dispatched calls from polymorphic call sites. In addition to improving performance by a median of 11%, polymorphic inline caches can be used as a source of type information for type feedback.
- A debugging system that dynamically *deoptimizes* code to provide source-level debugging of globally optimized code. Even though the optimized code itself supports only restricted debugging, the resulting system can hide these restrictions and provide *full* source-level debugging (i.e., including debugging operations such as single-stepping).

Although our implementation relies on dynamic compilation, most of the techniques described in this thesis do not require it. Type feedback would be straightforward to integrate into a conventional compiling system (see Section 5.6), similar to other profile-based optimizations. Source-level debugging using deoptimized code could be implemented by keeping precompiled unoptimized code in a separate file (see Section 10.5.3). Polymorphic inline caches only require a simple stub generator, not full-fledged dynamic compilation. Only the reoptimization system described in Section 5.3 is—by its very nature—specific to dynamic compilation.

Neither are the techniques described in this thesis specific to the SELF language. The debugging system is largely language independent, as discussed in Section 10.5.3. Type feedback could optimize late binding in any language; besides object-oriented languages, non-object-oriented languages that make heavy use of late binding (e.g., APL with its generic operators, or Lisp with its generic arithmetic) could profit from this optimization. Finally, any system using dynamic compilation might profit from adaptive recompilation to improve performance or to reduce compile pauses.

In the remainder of this dissertation, we will describe the design and implementation of these techniques and evaluate their performance impact on the SELF system. All of the techniques are fully implemented and stable enough to be part of the public SELF distribution.[†] Chapter 2 presents an overview of SELF and the work described in subsequent chapters, and discusses related work. Chapter 3 discusses how dynamic dispatch can be optimized by the runtime system, i.e., outside the compiler. Chapter 4 describes the non-optimizing SELF compiler and evaluates its performance. Chapter 5 describes how type feedback reduces the overhead of dynamic dispatch, and Chapter 6 describes the implementation of the optimizing SELF compiler. Chapters 7 and 8 then evaluate the new SELF compiler's performance relative to previous SELF systems and to other languages and investigate the impact of hardware features on performance. Chapter 9 discusses how optimizing compilation impacts the interactive behavior of the system. Chapter 10 describes how our system can provide full source-level debugging despite the optimizations performed by the compiler.

If you are in a hurry and already familiar with previous Smalltalk or SELF implementations, we recommend that you skim Chapter 2 (overview) and read the summaries at the end of each chapter, plus the conclusions.

The glossary on page 143 contains short definitions for the most important terms used in this thesis.

[†] Available via anonymous ftp from self.stanford.edu (WWW: http://self.stanford.edu).

2. Background and related work

This chapter presents the background and related work for the dissertation. First, we briefly introduce the SELF language and its goals, followed by an overview of the SELF system. Then, we will describe the compilation process of our new system, and finally we will review related work.

2.1 The SELF language

SELF is a dynamically-typed prototype-based object-oriented language originally designed in 1986 by David Ungar and Randall B. Smith at Xerox PARC [115]. Conceived as an alternative to the Smalltalk-80 programming language [58], SELF attempts to maximize programmer productivity in an exploratory programming environment by keeping the language simple and pure without reducing expressiveness and malleability.

SELF is a *pure object-oriented language*: all data are objects, and all computation is performed via dynamically-bound message sends (including all instance variable accesses, even those in the receiver object). Thus, SELF merges state and behavior: syntactically, method invocation and variable access are indistinguishable—the sender does not know whether the message is implemented as a simple data access or as a method. Consequently, all code is *representation independent* since the same code can be reused with objects of different structure, as long as those objects correctly implement the expected message protocol. In other words, SELF supports fully abstract data types: only the interface of an object (the set of messages that it responds to) is visible, and all implementation-level details such as an object's size and structure are hidden, even to code in the object itself.

SELF's main other highlights are listed below.

- SELF is dynamically-typed: programs contain no type declarations.
- SELF is based on prototypes [115, 90, 91] rather than classes. Every object is self-describing and can be changed independently. In addition to the flexibility of this approach, prototype-based systems can also avoid the complexity introduced by metaclasses.
- SELF has multiple inheritance. The inheritance design underwent several changes over the years. SELF-87 only had single inheritance, but SELF-90 introduced prioritized multiple inheritance combined with a new privacy mechanism for better encapsulation [25, 134] and a "sender path tiebreaker" rule for disambiguating between equal-priority parents [115]. More recently, the pendulum has swung back towards simplicity: SELF-92 [115] eliminated the sender path tiebreaker because it tended to hide ambiguities, and SELF-93 eliminated prioritized inheritance and privacy from the language.
- All control structures are user-defined. For example, SELF has no `if` statement. Instead, control structures are implemented with message sends and blocks (closures), just like in Smalltalk-80. However, unlike Smalltalk-80, the SELF implementation does not hardwire any control structure; in other words, a programmer can change the definition of *any* method (including those implementing `if` statements, loops, or integer addition), and the system will faithfully reflect these changes.
- All objects are heap-allocated and are deallocated automatically by a garbage collector.

These features are designed to harness the computing power of modern hardware to make the programmer's life easier. For example, representation independence makes it easier to reuse and reorganize code but creates implementation problems because every data access involves a message send. Rather than concentrating on minimizing program execution time, SELF concentrates on minimizing programming time.

2.2 The SELF system

The goal of the SELF implementation reflects that of the language: maximize programming productivity. Several features contribute towards this goal:

- *Source-level semantics*. The system's behavior can always be explained in source-level terms. Programmers should never be confronted with error messages such as "segmentation fault," "arithmetic exception: denormalized float," and so on, because such errors cannot be explained without diving into low-level implementation details that are outside the language definition and therefore hard to understand. Therefore, all SELF primitives are safe: arithmetic operations test for overflow, array accesses perform index bounds checks, and so on.
- *Direct execution semantics (interpreter semantics)*. The system should always behave as if it directly executed the source methods: any source change is immediately effective. Direct execution semantics frees the programmer from worrying about having to explicitly invoke compilers and linkers, and from having to deal with tedious details such as makefile dependencies.
- *Fast turnaround time*. After making a change, the programmer should not have to wait for a slow compiler or linker; such delays should be kept as short as possible (ideally, just a fraction of a second).
- *Efficiency*. Finally, programs should run efficiently despite SELF's pureness: the programmer should not be penalized for choosing SELF over a more conventional language.

These goals are not easy to reach. In particular, because of the emphasis on being the right language for the programmer rather than for the machine, SELF is hard to implement efficiently. Several key features of the language create particularly hard problems:

- Since all computation is performed by sending messages, the call frequency in a naive implementation is extremely high. If simple and frequent computations that usually require a single instruction in conventional languages (e.g., an instance variable access or an integer addition) all involve dynamically-dispatched procedure calls, programs will run dozens if not hundreds of times slower than in conventional languages.
- Similarly, since there are no built-in control structures, a simple `for` loop involves dozens of message sends and the creation of several blocks (closures). Because the definitions of control structures can be changed by the user, the compiler cannot take shortcuts and hardwire their translation with hand-optimized code patterns.
- Because the overall goal is to maximize programmer productivity, the system should be highly interactive and provide immediate feedback to the user. Therefore, long compilation pauses (as they often occur with optimizing compilers) are unacceptable, further restricting the implementor's freedom when searching for efficient SELF implementations.

The next few sections give an overview of the current SELF implementation and how it attempts to overcome the problems described above. After describing the base system, we will outline the new solutions that are the topic of this thesis, and how they fit into the existing system.

2.2.1 Overview of the implementation

The SELF virtual machine consists of several subsystems:

- The *memory system* handles allocation and garbage collection. Objects are stored in the heap; a generation scavenging garbage collector [131] reclaims unused objects. Object references are tagged with a two-bit tag in the lower two bits of every 32-bit word (tag 00 is for integers, 01 for pointers, 10 for floats and 11 for object headers). All objects except integers and floats consist of at least two words: a header word and a pointer to the object's *map* [23]. A map describes an object's format and can be viewed as the low-level type of the object. Objects with the same format share the same map, so that the object layout information is stored only once. To preserve the illusion

of self-describing objects mandated by the language, maps are copy-on-write: for example, when a slot is added to an object, that object gets a new map.

- The *parser* reads textual descriptions of objects and transforms them into real objects stored on the heap. Methods are represented with a set of very simple *byte codes* (essentially, “send,” “push literal,” and “return”).
- Given a message name and a receiver, the *lookup system* determines the result of a lookup, i.e., the matching slot. Lookups first check a hash table (the *code table*) to see if compiled code already exists for the lookup. If so, that code is executed; otherwise, the system performs an actual object lookup (traversing the receiver’s parent objects if necessary) and invokes the compiler to generate machine code for the resulting method or data access.
- The *compiler* translates the byte codes of a method into machine code and stores this code in the *code cache*. If there isn’t enough room in the code cache for the newly compiled method, existing methods are flushed to make room. The code cache keeps approximate LRU information to determine which compiled methods to flush.
- Finally, the virtual machine also contains numerous *primitives* that can be invoked by SELF programs to perform arithmetic, I/O, graphics, and so on. New primitives can be dynamically linked into the system at runtime.

References [88], [115], and [21] contain further details about the system.

2.2.2 Efficiency

Since SELF’s pure semantics threatened to make programs extremely inefficient, much of the early implementation effort went into compiler techniques for optimizing SELF programs. Some of these techniques were very successful:

Dynamic compilation. The compiler dynamically translates source methods into compiled methods on demand. That is, there is no separate compilation phase: execution and compilation are interleaved. (SELF’s use of dynamic compilation was inspired by the Deutsch-Schiffman Smalltalk system [44].)

Customization. Customization allows the compiler to determine the types of many message receivers in a method [23]. It extends dynamic compilation by exploiting the fact that many messages within a method are sent to `self`. The compiler creates a separate compiled version of a given source method for each receiver type (Figure 2-1). For

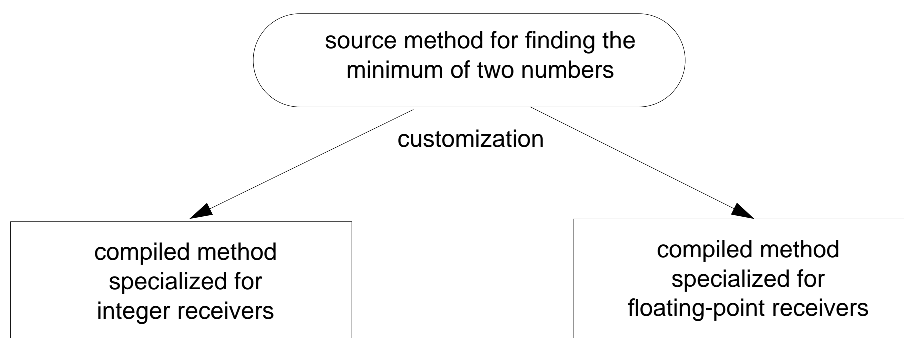


Figure 2-1. Customization

example, it might create two separate methods for the `min:` source method computing the minimum of two numbers. This duplication allows the compiler to customize each version to the specific receiver type. In particular, knowing the type of `self` at compile time enables the compiler to inline all sends `self`. Customization is especially important in SELF, since so many messages are sent to `self`, including instance variable accesses, global variable accesses, and many kinds of user-defined control structures.

Type Prediction. Certain messages are almost exclusively sent to particular receiver types. For such messages, the compiler uses an optimization originally introduced by early Smalltalk systems [44, 132]: it predicts the type of the

receiver based on the message name and inserts a runtime type test before the message send to test for the expected receiver type (Figure 2-2). Similar optimizations are performed in Lisp systems to optimize generic arithmetic. Along

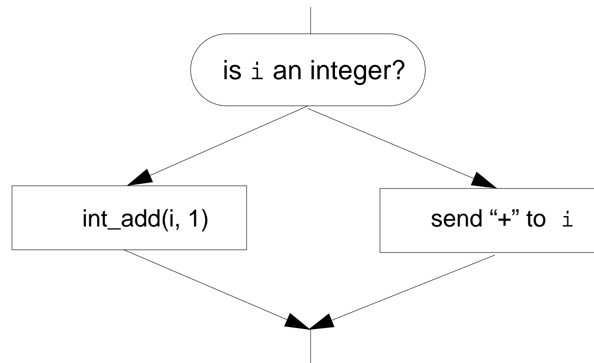


Figure 2-2. Type prediction for the expression i + 1

the branch where the type test succeeds, the compiler has precise information about the type of the receiver and can statically bind and inline a copy of the message. For example, existing SELF and Smalltalk systems predict that '+' will be sent to an integer [128, 58, 44], since measurements indicate that this occurs 90% of the time [130]. Type prediction improves performance if the cost of the test is low and the likelihood of a successful outcome is high.

Splitting is another way to turn a polymorphic message into several separate monomorphic messages. It avoids type tests by copying parts of the control flow graph [23, 22, 24]. For example, suppose that an object is known to be an integer in one branch of an `if` statement and a floating-point number in the other branch (Figure 2-3). If this object is

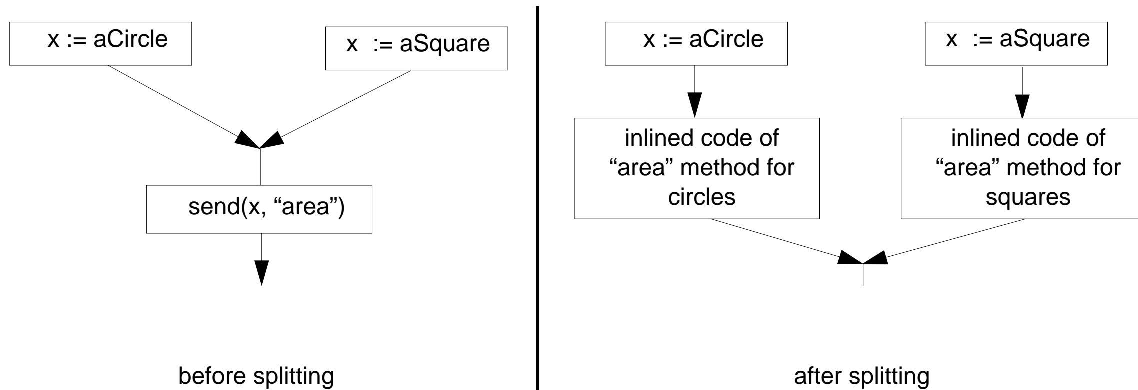


Figure 2-3. Splitting

the receiver of a message send following the `if` statement, the compiler can copy the send into the two branches. Since the exact receiver type is known in each branch, the compiler can then inline both copies of the send.

Together, these optimizations elevated SELF's performance to a very reasonable level. For example, Chambers and Ungar reported that SELF significantly outperformed the ParcPlace Smalltalk-80 implementation for a suite of small C-like integer benchmarks [26].

2.2.3 Source-level semantics

A combination of language and implementation features ensures that the behavior of all programs, even erroneous ones, can be understood solely in source-level terms. The language guarantees that every message send either finds a matching slot (accessing its data or running its method), or results in a "message not understood" error. Consequently, the only errors on that level are lookup errors. Furthermore, the implementation guarantees that all primitives are

safe. That is, all primitives check their operands and their results and fail in a well-defined way if the operation cannot be performed. For example, all integer arithmetic primitives check for overflow, and array access primitives check for range errors. Finally, since SELF does not allow pointer arithmetic and uses garbage collection, the system is pointer-safe: it is not possible to accidentally overwrite random pieces of memory or dereference "dangling" pointers. The combination of all these features makes it easier to find program errors.

Safe primitives make efficient implementation harder [21]. For example, every integer addition has to check for overflow, thus slowing it down. More importantly, the result type of integer operations is unknown: all primitives take a "failure block" argument, and if the operation fails, a message is sent to this argument. The result of this send then becomes the result of the primitive call. For example, the "+" method for integers in the current SELF system invokes the `IntAdd:` primitive with a failure block that converts the arguments into arbitrary-precision integers and adds them. Thus, the exact result type of the expression `x + y` is unknown even if both `x` and `y` are known to be integers: without overflow, the result will be an integer, but when the result is too big to be represented as a machine-level integer, the result will be an arbitrary-precision number. Therefore, even if `x` and `y` are known to be integers, the compiler does not know statically whether the second "+" in the expression `x + y + 1` will invoke the "+" method for integers or the "+" method for arbitrary-precision numbers. While safe primitives help the programmer, they potentially slow down execution.

2.2.4 Direct execution semantics

SELF mimics an interpreter by using dynamic compilation. Whenever a source method is invoked that has no corresponding compiled code, the compiler is invoked automatically to generate the missing code. Conversely, whenever the user changes a source method, all compiled code depending on the old definition is invalidated. To accomplish this, the system keeps *dependency links* between source and compiled methods [71, 21].

Since there is no explicit compilation or linking step, the traditional edit-compile-link-run cycle is collapsed into an edit-run cycle. Programs can be changed while they are running so that the application being debugged need not even be restarted from scratch.

2.3 Overview of the compilation process

Figure 2-4 shows the compilation process of the new SELF-93 system in more detail. SELF source methods are stored as objects in the heap, just like other data objects. The methods' code is encoded as a string of byte codes ("instructions") for a simple stack machine. These byte codes could directly be executed by an interpreter; however, the

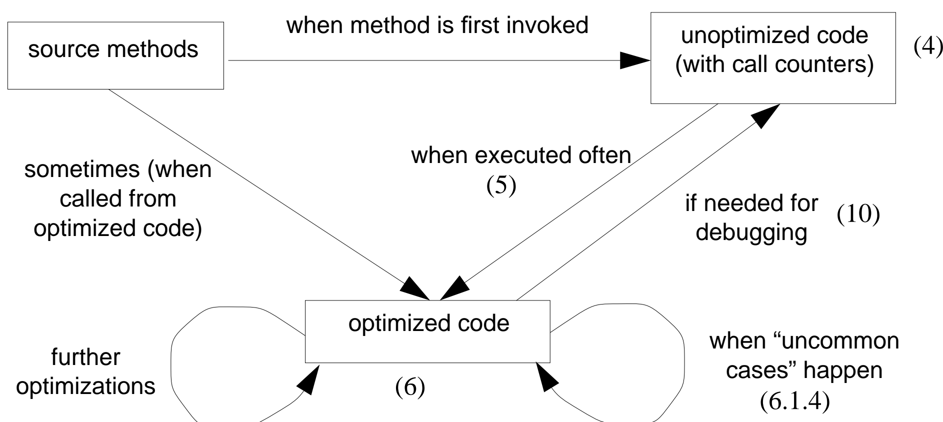


Figure 2-4. Overview of dynamic recompilation in SELF-93
(The numbers in the diagram refer to the sections discussing that part of the compilation process.)

current SELF system never does that. Instead, the byte codes are always translated into native machine code on demand. When a source method is executed for the first time, machine code is generated on the fly. Usually, this first compiled code is generated by a "fast but dumb" compiler. Unoptimized object code is a straightforward translation of the byte codes and thus quite slow; Chapter 4 describes the non-optimizing compiler in more detail.

Unoptimized methods contain invocation counters. Whenever such a counter exceeds a certain threshold, the system invokes the recompilation system to determine if optimization is needed and which methods should be recompiled. The recompilation system then calls the optimizing compiler. Chapter 5 explains the recompilation process in detail, and Chapter 6 describes the optimizing compiler.

As part of our work, we have developed a new variant of inline caches, polymorphic inline caches (PICs). PICs not only speed up the runtime dispatching at polymorphic call sites, they also provide valuable information about receiver types to the compiler. Chapter 3 describes how PICs work, and Chapter 5 explains how the optimizing compiler takes advantage of the type feedback information contained in PICs.

In some cases, optimized methods may be recompiled again. For example, the recompiling system could decide to reoptimize because the first optimized version of a method still contains too many calls that should be inlined (e.g., if there wasn't enough type information when optimizing the first time). An optimized method is also recompiled when it encounters "uncommon cases": the compiler may predict that certain situations never occur (e.g., integer overflow) and omit generating code for such cases. If such an omitted *uncommon case* happens anyway, the original compiled method is recompiled and extended with the code needed to handle that case (see Section 6.1.4).

In the remainder of the thesis we will discuss the individual components of the system in the order they usually come into play: first, polymorphic inline caches (PICs) and the non-inlining compiler, then recompilation, and finally the optimizing compiler itself.

2.4 Benefits of this work

All of the techniques described in this thesis were implemented in the SELF-93 system, thus improving it in several areas:

- *Higher performance on realistic programs*. By automatically recompiling and reoptimizing the performance-critical sections of a program, our optimizing compiler can take advantage of type information collected at runtime. This additional information often allows the compiler to produce more efficient code than previously possible, even though the compiler performs less compile-time analysis of the program. The `DeltaBlue` constraint solver, for example, runs more than 3 times faster with than with the previous optimizing compiler (see Chapter 7).

- *More stable performance*. The new compiler relies more on the dynamically observed program behavior and less on static analysis, and is thus less susceptible to breakdowns of static analysis techniques. Fragile performance was a major problem with the previous SELF compiler: small source changes could result in dramatic performance loss when a particularly important optimization was disabled by the change. For example, a set of small changes to the Stanford integer benchmarks slowed them down by a factor of 2.8 when compiled with the previous SELF compiler but only by 16% when compiled with the new compiler (see Section 7.4).

- *Faster and simpler compilation*. Our approach of relying on dynamic feedback from the runtime system rather than employing sophisticated static type analysis techniques has additional advantages besides better runtime performance. First, it results in a considerably simpler compiler (11,000 vs. 26,000 non-comment source lines). Second, the new compiler runs 2.5 times faster than the previous SELF compiler (see Section 9.5). For the user, compile pauses are reduced even more (see Section 9.2) because the system only needs to optimize the time-critical parts of an application.

- *Support for source-level debugging*. In the previous SELF system, users could print the stack of optimized programs, but they could not change programs while they were running, and they could not perform common

debugging operations such as single-stepping. Our new system provides this functionality by automatically *deoptimizing* code whenever needed to make the debugger's job easier (see Chapter 10).

2.5 Related work

This section compares the SELF system to generally related work; more detailed comparisons are given in later chapters.

2.5.1 Dynamic compilation

SELF uses dynamic compilation, i.e., generates code on-the-fly at runtime. The idea of dynamically creating compiled code rather than using traditional batch-style compilation grew of the quest for faster interpreters; by compiling to native code, some interpreter overhead (especially the decoding of the pseudo-code instructions) can be avoided. For example, Mitchell [97] proposed to convert parts of dynamically-typed interpreted programs into compiled form, assuming that the types of variables remained constant. Compiled code was generated as a side-effect of interpreting an expression for the first time. Similarly, threaded code [12] was used early on to remove some of the interpretation overhead.

Interpreters or dynamic compilers traditionally have been used for two reasons: first, some languages are hard to efficiently compile statically, usually because the source code alone does not contain enough low-level implementation type information to generate efficient code. Second, some languages strongly emphasize interactive use and thus were implemented with interpreters rather than slow compilers.

APL is a language that is both hard to compile statically (because most operators are polymorphic) and that emphasizes interactive use. Not surprisingly, APL systems were among the first to exploit dynamic compilers. For example, Johnston [80] describes an APL system using dynamic compilation as an efficient alternative to interpretation. Some of these systems used mechanisms similar to customization (e.g., Guibas and Wyatt [61]) and inline caching (Saal and Weiss [111]).

Deutsch and Schiffman pioneered the use of dynamic compilation for object-oriented systems. Their Smalltalk-80 implementation [44] dynamically translates the byte codes defined by the Smalltalk virtual machine [58] into native machine code and caches the compiled code for later use; Deutsch and Schiffman estimate that just using simple dynamic compilation instead of interpretation speeds up their system by a factor of 1.6, and that a more sophisticated compiler gains almost a factor of two.

Franz [55] describes a variation of dynamic compilation that generates machine code at load time from a compact intermediate-code representation. Like the byte codes contained in Smalltalk "snapshots," the intermediate code is architecture-independent; in addition, it is also intended to be language-independent (although the current implementation supports only Oberon). Compilation from intermediate code to machine code is fast enough to make the loader competitive with standard loaders.

Dynamic compilation is also useful for applications other than language implementation [82] and has been used in a wide variety of ways. For example, Kessler et al. use it to implement fast breakpoints for debuggers [84]. Pike et al. speed up "bit-blt" graphics primitives by dynamically generating optimal code sequences [103]. In operating systems, dynamic compilation has been used to efficiently support fine-grain parallelism [32, 105] and to eliminate the overhead of protocol stacks [1], and dynamic linking [67]. Dynamic compilation has also been used in other areas such as database query optimization [19, 42], microcode generation [107], and fast instruction set emulation [34, 93].

2.5.2 Customization

The idea of customizing portions of compiled code to some specific environment is closely related to dynamic compilation since the environment information is often not available until runtime. For example, Mitchell's system [97] specialized arithmetic operations to the runtime types of the operands. Whenever the type of a variable changed, all

compiled code which depended on its type was discarded. Since the language did not support user-defined polymorphism and was not object-oriented, the main motivation for this scheme was to reduce interpretation overhead and to replace generic built-in operators by simpler, specialized code sequences (e.g. to replace generic addition by integer addition).

Similarly, APL compilers created specialized code for certain expressions [80, 51, 61]. Of these systems, the HP APL compiler [51] comes closest to the customization technique used by SELF. The HP APL/3000 system compiles code on a statement-by-statement basis. In addition to performing APL-specific optimizations, the compiled code is specialized according to the specific operand types (number of dimensions, size of each dimension, element type, and storage layout). This so-called "hard" code can execute much more efficiently than more general versions since the computation performed by an APL operator may vary wildly depending on the actual argument types. To preserve the language semantics, the specialized code is preceded by a prologue verifying that the argument types actually conform to the specific assumptions made when compiling the expression. Therefore, the compiled code can safely be reused for later executions of the expression. If the types are still the same (which hopefully is the common case), no further compilation is necessary; if the types differ, a new version with less restrictive assumptions is generated (so-called "soft" code). From the descriptions of the system, it is not clear whether the old "hard" code is retained, or whether multiple hard versions can exist at the same time (each specialized for a different case).

Customization has also been applied to more traditional, batch-oriented compilers. For example, Cooper et al. describe a FORTRAN compiler that can create customized versions of procedures to enable certain loop optimizations [36]. Nicolau describes a FORTRAN compiler that dynamically selects the appropriate statically-generated version of a loop [99]. Saltz et al. delay loop scheduling until runtime [112]. Przybylski et al. generate a specialized cache simulator for each different set of simulation parameters [104]. Keppel et al. discuss value-specific runtime specialization for a variety of applications [83]. In more theoretically-oriented areas of computer science, customization is known as partial evaluation [15].

2.5.3 Previous SELF compilers

The SELF compiler described in this thesis has two predecessors. The first SELF compiler [22, 23] introduced customization and splitting and achieved reasonably good performance. For the Stanford integer benchmarks and the `Richards` benchmark, programs ran on the order of 4-6 times slower than optimized C[†], or about two times faster than the fastest Smalltalk-80 system at that time. The compiler's intermediate code was tree-based, which made it hard to significantly improve its code quality since there was no explicit representation of control flow. For example, it was hard to find the successor (in control-flow terms) of a given node, and this made optimizations spanning several nodes difficult to implement. Compilation speed was usually good but could vary widely since some of the compiler's algorithms were superlinear in the size of the source methods. Compile pauses were always noticeable, however, since all code was compiled with full optimization. The compiler was implemented in about 9,500 lines of C++.[‡]

The second compiler [21, 24] (called SELF-91) was designed to remove the restrictions of the first compiler and to further increase runtime performance. It introduced iterative type analysis and had a much more sophisticated back end. As a result, it achieved excellent performance on the Stanford integer benchmarks (around half the speed of optimized C); however, the performance of the `Richards` benchmark did not improve significantly. Unfortunately, users experienced similar discrepancies on their programs: while some programs (particularly small integer loops) showed excellent performance, many larger programs did not perform as well. (We will discuss the reasons for this discrepancy in more detail in Section 7.2.) The sophisticated analyses and optimizations performed by the compiler also exacted a price in compile time: compared to the previous SELF compiler, compilation was several times slower.

[†] Compared to the standard Sun C compiler as of 1989; since then the C compiler has improved, so that these ratios cannot be directly compared to current numbers.

[‡] Excluding empty lines, comments, or preprocessor lines; usually, there is no more than one statement per line.

The pauses were distracting enough to cause many users not to use the new compiler. Compared to the first compiler, the SELF-91 compiler was considerably more complex, consuming about 26,000 lines of C++ code.

2.5.4 Smalltalk-80 compilers

Smalltalk-80 is probably the object-oriented language closest to SELF, and several projects have investigated techniques to speed up Smalltalk programs.

2.5.4.1 The Deutsch-Schiffman system

The Deutsch-Schiffman system [44] represents the state of the art of commercial Smalltalk implementations. It contains a simple but very fast dynamic compiler that performs only peephole optimizations but no inlining. However, unlike SELF the Smalltalk-80 implementation hardwires certain important methods such as integer addition and messages implementing `if` statements and certain loops. As a result, the Deutsch-Schiffman compiler can generate efficient code for those constructs which could otherwise only be achieved with optimizations similar to those performed by the SELF compilers. The Deutsch-Schiffman compiler uses on the order of 50 instructions to generate one compiled machine instruction [45] so that compile pauses are virtually unnoticeable.

In addition to dynamic compilation, the Deutsch-Schiffman system also pioneered several other optimization techniques. *Inline caching* speeds up message lookup by caching the last lookup result at the call site (see Chapter 3). As a result, the cost of many sends can be reduced to the cost of a call and a type test. SELF also uses inline caching, and polymorphic inline caches extend its usefulness to polymorphic call sites (see Chapter 3). *Type prediction* speeds up common sends by predicting the likely receiver type (see Section 2.2.2). All SELF compilers except the non-optimizing SELF compiler described in Chapter 4 use type prediction to various degrees. The optimizing SELF compiler extends static type prediction by dynamically predicting receiver types based on type feedback information (Chapter 5).

2.5.4.2 SOAR

The goal of the SOAR ("Smalltalk On A RISC") project was to speed up Smalltalk through a combination of hardware and software [132, 129]. On the software side, SOAR used a non-dynamic native-code compiler (i.e., all methods were compiled to native code immediately after being parsed in), inline caching, type prediction, and a generation scavenging garbage collector [131]. Like the Deutsch-Schiffman compiler, the SOAR compiler did not perform extensive global optimizations or method inlining. On the hardware side, SOAR was a variation of the Berkeley RISC II processor [98]; the most important hardware features were register windows and tagged integer instructions. The combination of SOAR's software and hardware features was very successful when compared with other Smalltalk implementations on CISC machines: with a 400 ns cycle time, SOAR ran as fast as the 70 ns microcoded Xerox Dorado workstation and about 25% faster than the Deutsch-Schiffman Smalltalk system running on a Sun-3 with a cycle time of about 200 ns.[†] However, as we will see in Chapter 8, the optimization techniques used by the SELF compiler greatly reduces the performance benefit of special hardware support.

2.5.4.3 Other Smalltalk compilers

Other Smalltalk systems have attempted to speed up programs by annotating them with type declarations. Atkinson partially implemented a Smalltalk compiler that used type declarations as hints to generate better code [11]. For example, if the programmer declared a local variable to be of type "class X", the compiler could look up message sends to that variable and inline the called method. To make the code safe, the compiler inserted a type test before the inlined code; if the current value wasn't an instance of class X, an unoptimized, untyped version of the method was invoked. Although Atkinson's compiler was never completed, he could run some small examples and reported speedups of about a factor of two over the Deutsch-Schiffman system on a Sun-3. Of course, to obtain that speedup

[†] The cycle time of the Sun-3 is 180 ns for instructions that hit in the (very small) on-chip I-cache and 270 ns for instructions that miss.

the programmer had to carefully annotate the code with type declarations, and only invocations using the "right" types were sped up.

The TS compiler [79] took a similar approach. Types were specified as sets of classes,† and the compiler could statically bind calls where the receiver type was a single class. Methods were only inlined if the programmer had marked them as inlinable. If the receiver type was a small set of classes, the compiler inserted a type-test for each class and optimized the individual branches separately. This optimization can be viewed as a form of type prediction that is based on the programmer-supplied type declaration rather than a list of messages and expected types that is hardwired into the compiler. However, unlike type prediction and Atkinson's compiler, the programmer's type declarations were not just hints but firm promises. (The validity of these promises was to be checked by a type checker, but this checker was never completed and thus couldn't analyze significant portions of the Smalltalk system.)

The back end of the TS compiler used a RTL-based intermediate form and performed extensive optimizations. Thus (and because it was written in Smalltalk itself), the compiler was very slow, needing 15 to 30 seconds to compile the benchmarks which were all very short [79]. Few published data on the efficiency of the generated code are available since the TS compiler was never fully completed and could compile only very small benchmark programs. Data comparing TS to the Tektronix Smalltalk interpreter on a 68020-based workstation indicate that TS ran about twice as fast as the Deutsch-Schiffman system for small benchmarks that were annotated with type declarations for TS [79]. However, this comparison is not entirely valid because TS did not implement the full semantics of some Smalltalk constructs. For example, integer arithmetic is not checked for overflow. This could have a significant impact on the performance of the small TS benchmarks. For example, the loop of the `sumTo` benchmark (adding up all integers between 1 and 10,000) contains only very few instructions so that an overflow check could add a significant overhead. More importantly, the type of the variable accumulating the result sum could not be specified as SmallInteger because the result of a SmallInteger operation is not necessarily a SmallInteger (when an overflow occurs, the Smalltalk-80 primitive failure code converts the arguments into arbitrary-length integers and returns the sum of these). Thus, the declared type of the sum (and of the upper bound) would have to be more general, which would significantly slow down the generated code.

† The Typed Smalltalk system also had a second kind of type called "signature type," but this information was not used by the compiler for optimizations.

3. Polymorphic inline caching

Object-oriented programs send many messages, and thus sends must be fast. This chapter first reviews existing well-known techniques for improving the efficiency of lookups in dynamically-typed object-oriented languages and then describes polymorphic inline caches, an extension to standard inline caching that we have developed.

3.1 Out-of-line lookup caches

Object-oriented languages substitute message sends for procedure calls. Sending a dynamically-bound message takes longer than calling a statically-bound procedure because the program must find the correct target method according to the runtime type of the receiver and the inheritance rules of the language. Although early Smalltalk systems had simple inheritance rules and relatively slow interpreters, method lookup (also known as message lookup) was still responsible for a substantial portion of execution time.

Lookup caches reduce the overhead of dynamically-bound message passing. A lookup cache maps the pair (receiver type, message name) to the target method and holds the most recently used lookup results. Message sends first consult the cache by indexing to it with the given receiver type and message name. Only if the cache probe fails do they call the (expensive) lookup routine which traverses the inheritance graph to find the target method and then stores the result in the lookup cache, possibly replacing an older lookup result. Lookup caches are very effective in reducing the lookup overhead. Berkeley Smalltalk, for example, would have been 37% slower without a lookup cache [128].

3.2 Dispatch tables

Statically-typed languages often implement message lookup with dispatch tables. A common variant is to use the message name (encoded as an integer) to index into a type-specific dispatch table which contains the address of the target function. A message send then consists of loading the address of the dispatch table (which could be stored in the first word of each object), indexing into that table to get the address of the target function, and then calling that function.†

Dispatch tables are simple to implement in statically-typed languages because the range of the table index, the set of possible messages that can be sent to an object, is known statically, and thus the sizes (and contents) of these tables are easy to compute. However, it is possible to use dispatch tables in dynamically-typed languages as well, albeit with some added complications [7, 50, 135]. A major drawback of these methods is that they are hard to integrate into an interactive system since the dispatch tables need to be reorganized periodically after introducing new messages or new types; in a large system like the Smalltalk-80 system, such reorganizations can take many minutes, and the dispatch table can consume several hundred Kbytes. Therefore, we will not discuss dispatch tables further.‡

3.3 Inline caches

Even with a lookup cache, sending a message still takes considerably longer than calling a simple procedure because the cache must be probed for every message sent. Even in the ideal case, a cache lookup involves on the order of ten instructions to retrieve receiver type and message name, form the cache index (for example, by XORing and shifting the receiver type and the message name), fetch the cache entry, and compare the entry against the actual receiver type and message name to verify that the cached target method is indeed the correct one. That is, even if a lookup cache is 100% effective (all accesses hit), sends are still relatively slow because the lookup cache probe adds roughly ten instructions to every send.

---

† Various complications ensue when using multiple inheritance; for a discussion, see e.g. [54].

‡ As an aside, it is also unclear whether dispatch tables could compete with inline caches in terms of performance, especially on modern superscalar processors where an indirect call introduces costly pipeline stalls.

Fortunately, message sends can be sped up by observing that 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 very likely that the next time the send is executed it will also have a receiver of type *X*. For example, several studies have shown that in Smalltalk code, the receiver type at a given call site remains constant 95% of the time [44, 130, 132].

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., in the case of a hit no separate lookup cache is accessed), the technique is called *inline caching* [44, 132].† Figure 3-1 shows an empty inline cache; the calling method simply

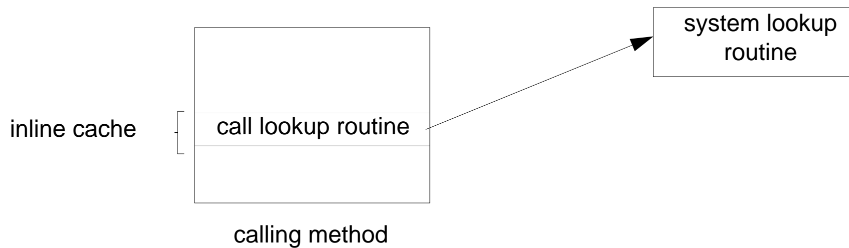


Figure 3-1. Empty inline cache

contains a call to the system's lookup routine. The first time this call is executed, the lookup routine finds the target method. But before branching to the target, the lookup routine changes the call instruction to point to the target method just found (Figure 3-2). Subsequent executions of the send jump directly to 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.

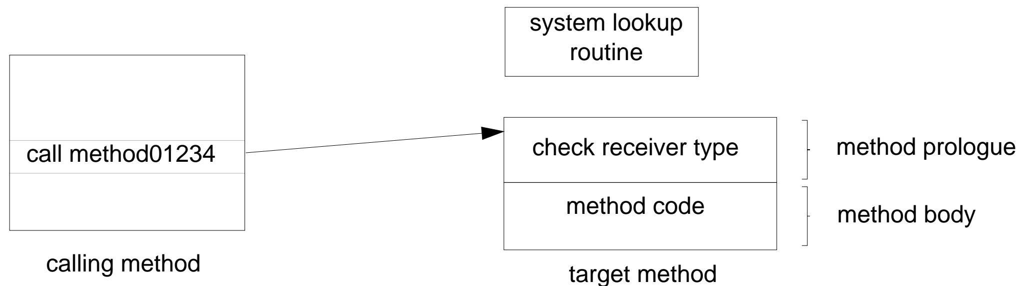


Figure 3-2. Inline cache after first send

Inline caches are faster than out-of-line lookup caches for several reasons. First, because each call site has a separate cache, the message name need not be tested to verify the cache hit—this test must be done only for misses (by the system's lookup routine). Second, inline caching doesn't need to execute any load instructions to fetch the cache entry; this function is implicitly performed by the call instruction. Finally, since there is no explicit indexing into any table, we can omit the XOR and shift instructions of a hash function. The only overhead that remains is the check of the receiver type which can usually be accomplished with very few instructions (two loads and a compare-and-branch in a typical Smalltalk system, one load and a comparison against a constant in the SELF system).

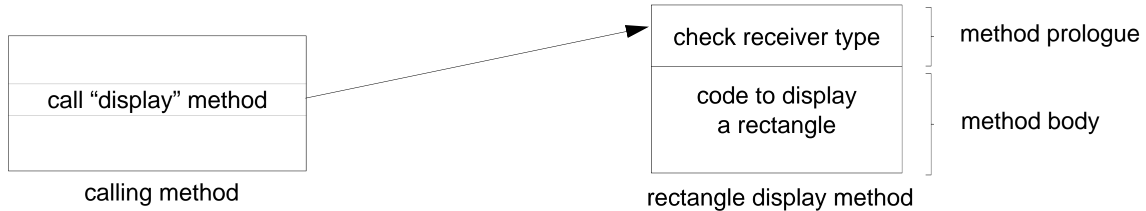
Inline caching is very effective in reducing lookup overhead because the hit rates are high and the hit cost is low. For example, SOAR (a Smalltalk implementation for a RISC processor) would have been 33% slower without inline caching [132]. All compiled implementations of Smalltalk that we know of incorporate inline caches, as does the SELF system.

† A similar technique was used by Saal and Weiss' APL interpreter [111] which used specialized routines for some operations (such as generic arithmetic) and speculatively used the previous specialization for the next call. Of course, inline caching also resembles some hardware mechanisms such as branch target buffers or direct-mapped processor caches.

3.4 Handling polymorphic sends

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† with several equally likely receiver types because the call target switches back and forth between different methods.

For example, suppose that a method is sending the `display` message to all elements in a list, and that the first list element is a rectangle:



If the next element in the list is a circle, control passes through the call in the inline cache to the display method for rectangles. The receiver test in the method prologue detects the type mismatch and calls the lookup routine which rebinds the inline cache to the circle's display method:

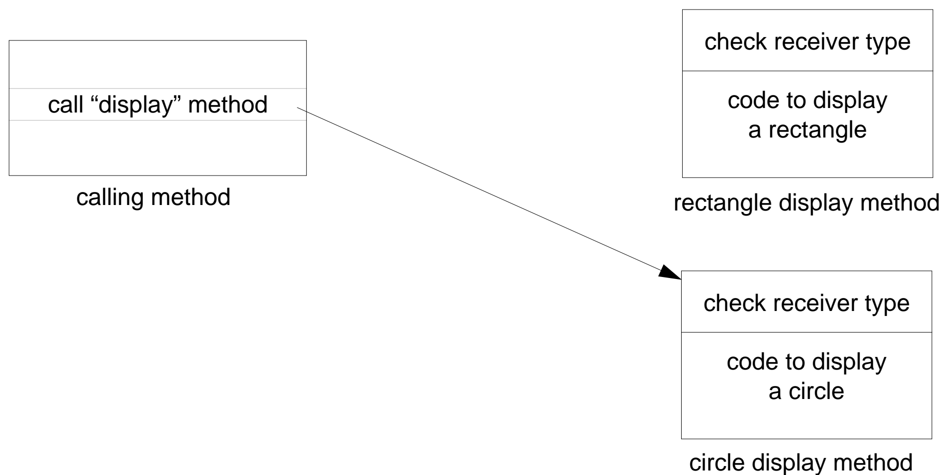


Figure 3-3. Inline cache miss

Unfortunately, if the list contains circles and rectangles in no particular order, the inline cache will miss again and again because the receiver type changes often. In this case, inline caching may even be slower than using an out-of-line lookup cache because the miss time is higher. In particular, the miss code includes changing the call instruction which requires invalidating parts of the instruction cache on most modern processors. Often, the miss handler also includes considerable additional overhead. For example, the SELF system links all inline caches calling a particular method into a list so that they can be updated when the method is relocated to a different address or if it is discarded.

The performance impact of inline cache misses becomes more severe in highly efficient systems, where it can no longer be ignored. For example, measurements for the SELF-90 system showed that the `Richards` benchmark spent about 25% of its time handling inline cache misses [23].

† We will use the term "polymorphic" for call sites where polymorphism is *actually* used. Consequently, we will use "monomorphic" for call sites which do not actually use polymorphism even though they might *potentially* be polymorphic.

An informal examination of polymorphic call sites in the SELF system showed that in most cases the degree of polymorphism is small, typically less than ten. The degree of polymorphism of sends has a trimodal distribution: most sends are *monomorphic* (only one receiver type), some are *polymorphic* (a few receiver types), and very few are *megamorphic* (very many receiver types). Figure 3-4[†] shows the arity distribution of non-empty inline caches, i.e.,

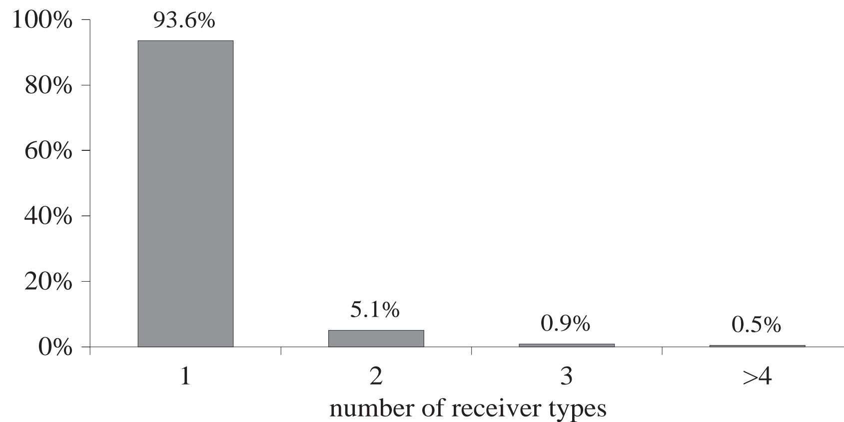


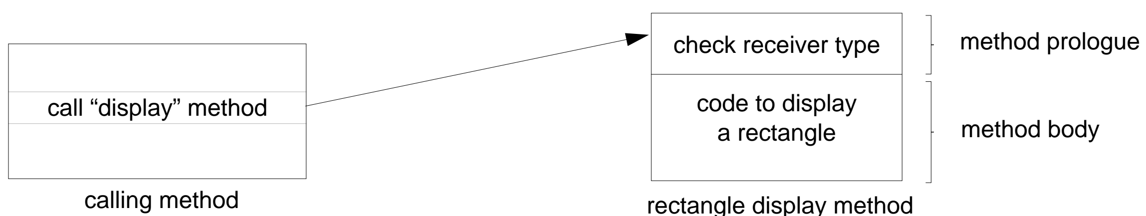
Figure 3-4. Size distribution (degree of polymorphism) of inline caches

the degree of polymorphism exhibited by call sites. The distribution was taken after using the prototype SELF user interface [28] for about a minute. The overwhelming majority of call sites have only one receiver type; this is why normal inline caching works well. Quite a few call sites have two different receiver types; a frequent case are boolean messages, since true and false are two different types in SELF. Very few call sites have more than five receiver types.[‡] (All call sites with more than ten receiver types invoked only blocks; Section 4.3.2 explains this megamorphic behavior.) The data in Figure 3-4 suggests that the performance of polymorphic calls could be improved with a more flexible form of caching since most non-monomorphic sends have only very few receiver types.

3.5 Polymorphic inline caches

In order to reduce the inline cache miss overhead, we have designed and implemented *polymorphic inline caches*, a new technique that extends inline caching to handle polymorphic call sites efficiently. Instead of merely caching the last lookup result, a polymorphic inline cache (PIC) caches *several* lookup results for a given polymorphic call site in a specially-generated stub routine.

In our example of sending the `display` message to the elements of a list, a system using PICs initially works like normal inline caching: after the first send, the inline cache is bound to the rectangle display method.



[†] See Table A-1 in Appendix A for detailed data.

[‡] Ifor Williams obtained similar results when analyzing Smalltalk-80 traces [141]. He reports 67.2% monomorphic call sites, 12.9% bimorphic sites, and 4.5% trimorphic sites.

But when the first circle is encountered, instead of simply switching the call's target to the circle display method the miss handler constructs a short stub routine and rebinds the call to this stub routine:

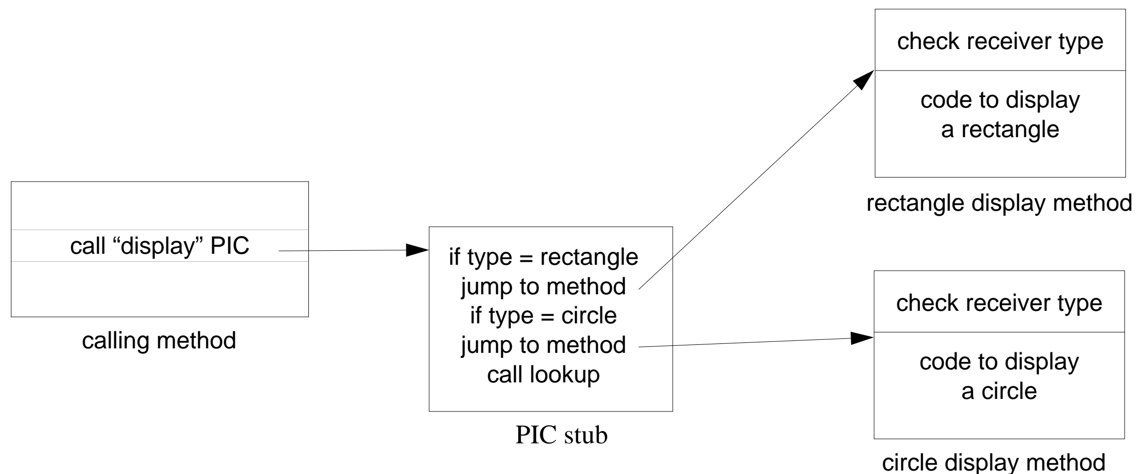


Figure 3-5. Polymorphic inline cache

The stub (a type-case) checks if the receiver is either a rectangle or a circle and branches to the corresponding method. The stub can branch directly to the method's body (skipping the type test in the method prologue) because the receiver type has already been verified. However, methods still need a type test in their prologue because they can also be called from monomorphic call sites which have a standard inline cache.

Because the PIC now caches both the rectangle and the circle case, no more misses will occur if the list contains just rectangles and circles. All sends will be fast, involving only one or two comparisons before they find their target. If the cache misses again (i.e. the receiver is neither a rectangle nor a circle), the stub routine will simply be extended to handle the new case. Eventually, the stub will contain all cases seen in practice, and there will be no more cache misses or lookups.

3.5.1 Variations

The scheme described above works well in most cases and reduces the cost of a polymorphic send to a few machine cycles. This section discusses some remaining problems and possible solutions.

Coping with megamorphic sends. Some send sites may send a message to a very large number of types. For example, a method might send the `writeSnapshot` message to every object in the system. Building a large PIC for such a send wastes time and space. Therefore, the inline cache miss handler should not extend the PIC beyond a certain number of type cases; rather, it should mark the call site as being megamorphic and adopt a fallback strategy, possibly just the traditional monomorphic inline cache mechanism.

In the SELF system, PICs are marked as megamorphic if the exceed a certain size (currently, 10 cases). When a megamorphic PIC misses, it does not grow to include the new type. Instead, one of its cases is picked at random and replaced with the new case. An earlier version of the system used a "move-to-front" strategy which inserted the new case in the front, shifting all other cases back and dropping the last case. However, this strategy was abandoned because its miss cost was high (all 10 entries needed to be changed, not just one) and its miss ratio was very high occasionally, namely if types changed in a circular fashion. (Yes, Murphy's law holds—real programs exhibited this behavior.)

Improving linear search. If the dynamic usage frequency of each type were available, PICs could be reordered periodically in order to move the most frequently occurring types to the beginning of the PIC, reducing the average number of type tests executed. If linear search were not efficient enough, more sophisticated algorithms like binary

search or some form of hashing could be used for cases with many types. However, since the number of types is very small on average (see Figure 3-4), this optimization is probably not worth the effort: a PIC with linear search is probably faster than other methods for most situations.

Improving space efficiency. Polymorphic inline caches are larger than normal inline caches because of the stub routine associated with every polymorphic call site. If space were tight, call sites with identical message names could share a common PIC to reduce the space overhead. In such a scenario, PICs would act as fast message-specific lookup caches. The average cost of a polymorphic send would likely be higher than with call-site-specific PICs because the number of types per PIC would increase due to the loss of locality (a shared PIC will contain all receiver types for the particular message name, whereas a call-specific PIC only contains the types which actually occur at that call site).

3.6 Implementation and results

We have designed and implemented polymorphic inline caches for the SELF system and measured their effectiveness. All measurements in this section were done on a lightly-loaded Sun-4/260 with 48 MB of memory; the base system used for comparison was the SELF system as of September 1990.† The base system uses inline caching; a send takes 8 instructions (9 cycles) until the method-specific code is reached. An inline cache miss takes about 15 microseconds or 250 cycles.‡ The miss time could be reduced by some optimizations and by recoding critical parts in assembly. We estimate that such optimizations could reduce the miss overhead by about a factor of two. Thus, our measurements may overstate the direct performance advantage of PICs by about the same factor. On the other hand, measurements of a commercial Smalltalk-80 implementation (the ParcPlace Smalltalk-80 system, version 2.4) indicate that it also takes about 15 microseconds to handle a miss, and thus our current implementation does not seem to be unreasonably slow.

Monomorphic sends in our experimental system use the same inline caching scheme as the base system. For polymorphic sends, a stub is constructed which tests the receiver type and branches to the corresponding method. The stub has a fixed overhead of 8 cycles (to load the receiver type and to jump to the target method), and every type test takes 4 cycles. The PICs are implemented as described in Section 3.5. None of the variations mentioned in the previous section are implemented, except that a call site is treated as megamorphic if it has more than ten receiver types (but such calls do not occur in our benchmarks).

	base SELF	with PICs	Smalltalk
Inline cache hit	9	9	unknown
Inline cache miss	approx. 250	N/A	approx. 250
Polymorphic send	250	8 + 2n	250

Table 3-1. Lookup timings (cycles on SPARCstation-2)

Table 3-1 summarizes the lookup timings. The cost of a polymorphic send in the base system depends on the inline cache miss ratio at the polymorphic call site. For example, if the receiver type changes every third time the send is executed, the cost would be approximately 250/3 = 83 cycles. For the system with PICs, the average polymorphic lookup cost is a function of the number of cases; if all n cases are equally likely, the average dispatch will involve n/2 type tests and therefore have a cost of 8 + (n/2) * 4 = 8 + 2n cycles.

† These measurements were done in September 1990, not with the current SELF system.

‡ The actual time taken may vary somewhat depending on hardware influences such as processor cache misses and on algorithmic details that are beyond the scope of this discussion. The 15 microsecond estimate was obtained by profiling the execution of the `PolyTest` benchmark with the `gprof` profiler.

In order to evaluate the effectiveness of polymorphic inline caches, we measured a suite of SELF programs. The programs (with the exception of `PolyTest`) can be considered fairly typical object-oriented programs and cover a variety of programming styles. More detailed data about the benchmarks is given in [70].

`Parser`. A recursive-descent parser for an earlier version of the SELF syntax (550 lines).

`PrimitiveMaker`. A program generating C++ and SELF stub routines from a description of primitives (850 lines).

`UI`. The SELF user interface prototype (3000 lines) running a short interactive session. Since the Sun-4 used for our measurements has no special graphics hardware, runtime is dominated by graphics primitives (e.g. polygon filling and full-screen bitmap copies). For our tests, the three most expensive graphics primitives were turned into no-ops; the remaining primitives still account for about 30% of total execution time.

`PathCache`. A part of the SELF system which computes the names of all global objects and stores them in compressed form (150 lines). Most of the time is spent in a loop which iterates through a collection containing about 8 different kinds of objects.

`Richards`. An operating system simulation benchmark (400 lines). The benchmark schedules the execution of four different kinds of tasks. It contains a frequently executed polymorphic send (the scheduler sends the `runTask` message to the next task).

`PolyTest`. An artificial benchmark (20 lines) designed to show the highest possible speedup with PICs. `PolyTest` consists of a loop containing a polymorphic send of degree 5; the send is executed a million times. Normal inline caches have a 100% miss rate in this benchmark (no two consecutive sends have the same receiver type). Since `PolyTest` is a short, artificial benchmark, we do not include it when computing averages for the entire set of benchmarks.

3.6.1 Execution time

To obtain more accurate measurements, all benchmarks were run 10 times in a row and the average CPU time was computed. This process was repeated 10 times, and the smallest average was chosen (the assumption being that longer execution times were caused by external influences such as other UNIX processes). A garbage collection was performed before every measurement in order to reduce inaccuracies. Figure 3-6 shows the execution time saved by the system using PICs (see Table A-2 in Appendix A for details).

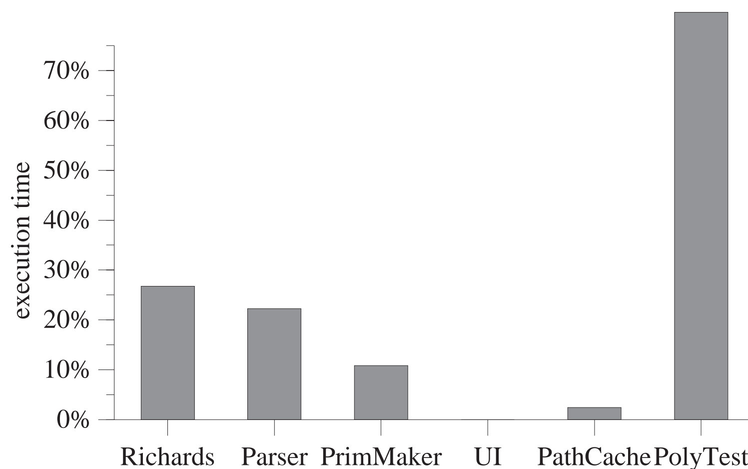


Figure 3-6. Execution time saved by polymorphic inline caches

The savings vary from none (for `UI`) to substantial (`Richards`, `Parser`) to spectacular (`PolyTest`); the median saving for the benchmarks (without `PolyTest`) is 11%. The speedup observed for the individual benchmarks closely corresponds to the time required to handle inline cache misses in the base system. For example, in the base system `PolyTest` spends more than 80% of its execution time in the miss handler, and it is more than 80% of its execution time are eliminated by PICs. This close correlation indicates that PICs eliminate virtually all of the overhead of inline cache misses.

Benchmark	miss ratio
Richards	4.43%
Parser	6.18%
PrimitiveMaker	6.66%
UI	0.62%
PathCache	5.76%

Table 3-2. Inline cache miss ratios of benchmark programs

Table 3-2 shows the miss ratios of our benchmarks in the base system. Although the SELF implementation with its optimizing compiler is quite different from Smalltalk systems, the miss ratios agree well with those observed in previous studies of Smalltalk systems, which observed miss ratios on the order of 5% [44, 130, 132]. The miss ratios do not directly correlate to the speedups observed when introducing PICs because the benchmarks have very different call frequencies (differing by more than a factor of five).

One might expect that the inline cache miss ratio is correlated to the degree of polymorphism exhibited by a program, measured by the fraction of messages sent from polymorphic call sites (i.e., from inline caches that encounter at least two different receiver types). For example, one would assume that a program sending 80% of its messages from polymorphic call sites has a higher inline cache miss ratio than a program that sends only 20% of its messages from polymorphic call sites. Interestingly, this is not the case for our benchmark programs (Figure 3-7). For example, more

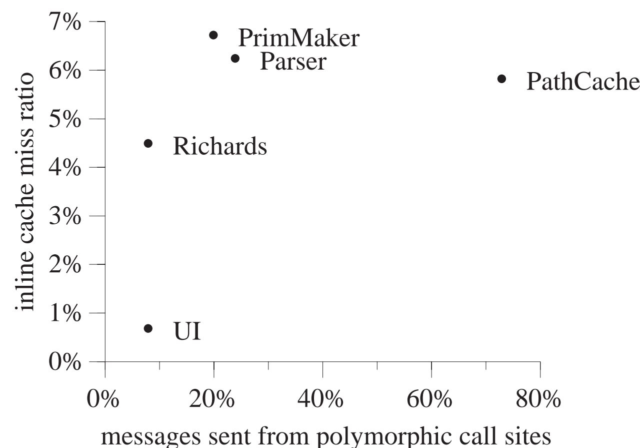


Figure 3-7. Polymorphism vs. miss ratio

than 73% of the messages sent in `PathCache` are from polymorphic call sites vs. 24% for `Parser`, but `PathCache`'s inline cache miss ratio (5.8%) is slightly *lower* than `Parser`'s miss ratio (6.2%). Apparently, one receiver type dominates at most polymorphic call sites in `PathCache` (so that the receiver type rarely changes), whereas the receiver type frequently changes in `Parser`'s inline caches. Thus, ordering a PIC's type tests by frequency of occurrence (as suggested in section 3.2) might be a win for programs like `PathCache`.

3.6.2 Space overhead

The space overhead of PICs is low, typically less than 2% of the compiled code (see Table 3-3). The main reason for

Benchmark	Overhead
Richards	0.8%
Parser	1.5%
PrimMaker	1.7%
PathCache	6.9%
UI	not available

Table 3-3. Space overhead of PICs

this low overhead is that most call sites do not need a PIC because they are monomorphic. Thus, even though a 2-element PIC stub occupies about 100 bytes, the overall space overhead is very modest.

3.7 Summary

Traditional inline caching works well for most sends. However, truly polymorphic call sites (i.e., call sites where the receiver type changes frequently) can cause significant inline cache miss overheads, consuming up to 25% of total execution time in the programs we measured.

We have extended traditional inline caching with polymorphic inline caches (PICs) that cache multiple lookup targets. PICs are very effective in removing the inline cache miss overhead, speeding up execution of our benchmark programs by up to 25%, with a median of 11%.

Although we have discussed PICs as a way to speed up message sends, their primary purpose in the SELF system is to provide type information to the optimizing compiler. The performance improvements gained by this type information by far outweigh the speedups observed in this chapter. We will discuss this aspect of PICs in detail in Chapter 5.

4. The non-inlining compiler

The long compile pauses caused by the previous SELF compilers were very distracting to users and threatened to compromise the goal of increasing programmer productivity. For example, starting the prototype user interface took about 60 seconds[†], out of which about 50 seconds were compile time. Although the SELF compiler was about as fast as the standard C compiler [26], it was still too slow.

4.1 Simple code generation

To improve the responsiveness of the system, we decided to implement a non-inlining compiler ("NIC"). Its task is to compile methods as quickly as possible, without attempting any optimizations. Consequently, its code generation strategy is extremely simple. The compiler translates the byte codes of a source method directly into machine code without building up an intermediate representation.

Byte codes are translated as follows:

- Source literals (e.g. 'foo') are loaded into a register. If the literal is a block, the BlockClone primitive is first called to create the block (except for primitive failure blocks which are handled specially, see below).
- Primitive sends (e.g., `_IntAdd:`) are translated into calls to the corresponding C or assembly primitive.
- Sends are translated into dynamically-dispatched calls by generating an inline cache, except if the send accesses a local variable of the method, in which case the corresponding stack location is accessed.

Register allocation is equally simple: expression stack entries (i.e. expressions that have been evaluated but not yet consumed by a send because other expressions need to be evaluated first) are allocated to registers, and a bit mask keeps track of the available registers. Locals are stack allocated since they might be uplevel-accessed by nested blocks; all incoming arguments (which are passed in registers) are flushed to the stack for the same reason. This is one of the areas where a little bit of analysis of the source code (for example, to see if a method contains any blocks) could speed up the generated code. The non-inlining compiler consists of about 2600 lines of C++.

The compiler performs only two optimizations, both of which were very simple to implement and known to have a significant benefit. First, the creation of primitive failure blocks is delayed until they are needed. For example, the integer addition primitive `IntAdd:` takes a failure block as an argument. This block is invoked when the primitive fails, for example because of an overflow. However, primitive calls usually succeed, and thus the failure block usually isn't needed. Therefore, the compiler delays the creation of failure blocks until the primitive actually fails (i.e. until it returns a special error value). This optimization speeds up programs by about 10-20% since it greatly reduces the allocation rate of programs.

The second optimization is that accesses to slots in the receiver are inlined, replacing a send with a load instruction.[‡] While the resulting speedup is very modest (usually only a few percent), this optimization reduces the size of the compiled code by about 15% since it replaces 10-word inline caches with one-word load instructions.

In addition, the NIC also uses customization, even though it does not profit much from it (the inlining of receiver instance variable accesses is the only optimization taking advantage of customization). Customization is used because the assumption that all code is customized permeates many parts of the SELF system (especially the lookup system), so that the NIC was much easier to integrate into the system in this way. Removing customization from the NIC could be advantageous since it could reduce code duplication.

† All times are on a SPARCStation-2 (a workstation rated at about 22 SPECInt92) unless noted otherwise.

‡ This optimization was added by Lars Bak.

The NIC does not perform some optimizations commonly performed by similar compilers for other languages. Most notably, it does not optimize integer arithmetic in any way, and it does not special-case or inline any control structures (e.g., `ifTrue:`). The performance impact of these decisions will be examined in Section 4.3 and 4.4.

4.2 Compilation speed

The NIC uses roughly 400 instructions to generate one SPARC instruction. This is significantly slower than the simple compiler described by Deutsch and Schiffman which uses around 50 instructions per generated instruction [45].

There are several reasons for this difference. First, SELF's byte codes are much higher-level than the byte codes of the Smalltalk-80 virtual machine. For example, SELF uses the same `send` byte code for general sends, accesses to instance variables, accesses to local variables, and primitive calls since all these are syntactically equivalent in SELF. In contrast, Smalltalk-80 has a much more machine-oriented byte code format; in fact, the byte codes were directly interpreted by microcode in early Smalltalk implementations [43]. For example, Smalltalk has special byte codes for instance variable accesses (specifying the slot number), local accesses, and the most frequent primitives (arithmetic and control transfer). Therefore, the Smalltalk-80 byte code compiler (the equivalent of the SELF parser) already performs some of the compilation work such as replacing simple control structures with jump byte codes and replacing arithmetic operations with special arithmetic byte codes. As a result, the Deutsch-Schiffman compiler has to perform less work to translate the Smalltalk-80 byte codes into machine code. In contrast, such information isn't explicit in the SELF byte codes since it was not designed with interpretation or fast compilation in mind.

Figure 4-3 shows a coarse profile of NIC compilation obtained by profiling about 8,000 compilations with the `gprof` profiler. Interestingly, the actual compilation takes only twice as long as the allocation of the compiled method.† Not surprisingly, handling message sends takes up a good portion of the actual compilation (33% of compilation time, or 23% of the total). But the compiler also spends 17% of its time (12% of the total) on tasks that are handled by the parser in the Smalltalk system, such as searching for slots, determining if a send is a local slot access, and computing which byte codes are statement boundaries.

Activity	Time
Total compilation time	100%
Allocating and initializing compiled method	31%
Total time in NIC itself	69%
Compiling a method	64%
Compiling a send	23%
Generating a real send	7%
Generating a local slot access	6%
Generating an instance variable access	3%
Generating method prologue and epilogue	12%
Compiling methods for slot access and assignment	3%
Determining statement boundaries in source method	4%
Determining if send accesses a local slot	4%
Searching for a slot in the receiver object	4%

Figure 4-1. Profile of NIC compilation

† The allocator is fairly efficient; the vast majority of allocation requests is filled from free lists kept for the most frequently requested sizes. The time listed in the profile also includes time for flushing unused compiled methods (if necessary to make room for the new method), copying the compiled code and auxiliary data such as dependencies and debugging information into the newly allocated space, and relocating object code and dependency pointers.

Another source of overhead is that the compiler uses data abstraction liberally in order to simplify the implementation. For example, it uses a code generator object to abstract code generation details from the front end, so that porting the compiler is simplified. Similarly, the code generator uses an assembler object to generate the instructions rather than manipulating instruction words itself. While the assembler is quite efficient (the compiler calls methods such as load(base_reg, offset, destination) rather than passing the string "load base, offset, dest"), this organization prevents some optimizations such as using precomputed patterns for certain frequent instructions or instruction sequences.† On the other hand, it simplifies the system because all compilers can share the same assembler.

Another way to characterize compile time is to measure compile time per byte code, i.e., per source code unit.

Figure 4-2. NIC compile time as a function of source method size

Figure 4-2 shows that compile time is approximately linear to the source method length, although the time for methods of equal size can vary somewhat since the different kinds of byte codes take different amounts of time to compile. For example, loading the constant 1 is much faster than compiling a send: the former directly generates one machine instruction, whereas the latter generates up to a dozen or so instructions and first has to check whether the send accesses a local variable or an instance variable in the receiver. On average, the compiler uses 0.2 ms per byte code and has a start-up overhead of 2 ms; the linear regression's correlation coefficient is 0.78.‡ Most compilations are short, with a mean of 3 ms (Figure 4-3).

4.3 Execution speed

Unoptimized code is quite slow; Table 4-1 shows its performance relative to optimized SELF programs.†† On large programs, unoptimized code runs about 9 times slower than the code generated by an optimizing SELF compiler, but smaller programs with tight loops can slow down even more. As an extreme case, `BubbleSort` slows down by two orders of magnitude.

† An earlier version of the NIC spent more than 50% of its time in the assembler, mostly because the C++ compiler was generating inefficient code for the bit field operations used to compose instructions. The current overhead is hard to estimate because most assembler functions are inline functions and thus don't show up in a `gprof` profile. However, based on the time taken by non-inlined functions, we can say that code generation takes up at least 10% of total compilation time.

‡ The time was measured as elapsed time on an unloaded system; the system's CPU timer resolution was too coarse to measure the very short compile times.

†† See Table 7-1 on page 67 for more details on the benchmarks.

Figure 4-3. Histogram of NIC compile time

Benchmark size	Benchmark	slowdown factor
tiny	BubbleSort	164
small	DeltaBlue	38.9
	Richards	49.8
	PrimMaker	12.3
large	CecilComp	13.8
	CecilInt	6.5
	Mango	7.5
	Typeinf	14.6
	UI1	9.2
	UI3	6.9
	geometric mean	9.25

Table 4-1. Performance of unoptimized code relative to optimized code

Where is the time spent in unoptimized programs? This section analyzes this questions using `CecilInt` (a large, object-oriented program) and `Richards` (a smaller, less polymorphic program) as examples and discusses the four main reasons: register windows, lookups, instruction cache misses, and primitive calls / garbage collection.

4.3.1 Overview

Table 4-2 shows a rough profile of the benchmarks' execution time. The actual compiled code accounts for about half of the total execution time. About 20-30% of the time is spent handling the inline cache misses (see Section 4.3.2), and another 20-30% goes to primitives (such as integer addition or block allocation) and garbage collection.

Category	Richards	CecilInt	discussed in section(s)...
Compiled SELF code	44%	50%	4.3.4
Primitives + GC	19%	30%	4.3.3
Lookup miss handler	35%	18%	4.3.2
Other	2%	1%	

Table 4-2. Software profile of unoptimized code

Table 4-3 shows a hardware-oriented view of the same two programs; it includes data from the entire program execution (i.e., not only from compiled SELF code). Both register windows and instruction cache misses contribute to the programs' low performance. These two factors are discussed in sections 4.3.4 and 4.3.5.

Category	Richards	CecilInt	discussed in section(s)...
Register windows	32%	32%	4.3.4
Instruction cache misses	23%	32%	4.3.5

Table 4-3. Hardware profile of unoptimized code

4.3.2 Lookup cache misses

In unoptimized code, more call sites are *megamorphic*, i.e., their receiver type changes very frequently between a very large number of receiver types. For example, the `ifTrue:` method of the `true` object sends `value` to its first argument:

```
ifTrue: aBlock = ( aBlock value )
```

`ifTrue:` is sent from many places, all passing different argument blocks. Therefore, the `value` send is megamorphic since it has a different receiver type for each use of `ifTrue:` in the program.[†] PICs cache only up to 10 different receiver types in the current system, the send often misses in the PIC, causing a costly lookup miss handler execution. Together, all these misses account for one third of `Richard`'s execution time and 18% of `Cecil`'s. In optimized code, `ifTrue:` would be inlined, allowing the compiler to inline the `value` send as well (since the argument usually is a block literal). Thus, in optimized code the `value` send incurs no overhead.

4.3.3 Blocks, primitive calls, and garbage collection

The NIC creates many more blocks (closures) than optimized SELF code because it does not optimize control structures that involve blocks (or anything else, for that matter). For example, several blocks are created for each iteration of a loop. Together, block creation accounts for about 5% of execution time in both benchmarks. Since so many block objects are created, many more scavenges (partial garbage collections) are necessary. Fortunately, these scavenges are very fast since most objects (mostly blocks) do not survive, so that the GC overhead is less than 3% for both programs. Unoptimized code calls many primitives, even for simple operations like integer arithmetic and comparisons. Together, all the primitives (including block allocation and GC) use about 20-30% of the total execution time.

4.3.4 Register windows

The SPARC architecture [118] defines a set of overlapping register windows which allows a procedure to save the caller's state by switching to a new set of registers. As long as the call depth does not exceed the number of available

[†] In SELF, each block has a different type because its `value` slot (the block method) is different. In contrast, all blocks have the same type (class) in Smalltalk-80.

register sets, such a save can be performed in one cycle without any memory accesses. If no free register set is available when a save instruction is executed, a “register window overflow” trap occurs and the trap handler transparently frees a set by saving its contents in memory. Similarly, a “window underflow” trap is used to reload such a flushed register set from memory if it is needed again. Unfortunately, unoptimized SELF code has both a high call density and a high call depth; for example, a `for` loop (implemented using messages and blocks) has a call depth of 13. Therefore, unoptimized code incurs very frequent window overflow and underflow traps and spends a significant amount of time handling these traps. Section 8.2 will analyze this overhead in detail; for now, it suffices to say that the register window overhead for unoptimized code can be as high as 40% of total execution time with current SPARC implementations.

4.3.5 Instruction cache misses

The code generated by the NIC also consumes a lot of space: every send takes about 15 32-bit words for the instructions loading the parameter registers, the call, and the inline cache. Additionally, each method prologue is around 15 words (the prologue tests the receiver map, increments and tests the invocation counter for recompilation, establishes the stack frame and tests for stack overflow). As a result, unoptimized code has a relatively high I-cache miss overhead: `Richards` spends 10% of its time waiting for instructions to be fetched from memory, and `CecilInt` spends 27% of its time in I-cache misses.

4.4 NIC vs. Deutsch-Schiffman Smalltalk

The NIC is similar to the Deutsch-Schiffman Smalltalk compiler in many respects. But ParcPlace Smalltalk-80 (version 4.0) runs the Richards benchmark in 3.2 seconds, more than ten times faster than the unoptimized SELF version. Why is the Smalltalk system so much faster? How much of the performance disparity is caused by differences in the language, and how much by differences in the implementation?

Several differences could explain the performance discrepancy; in particular, three reasons come to mind, two language differences and one implementation difference:

- Smalltalk hardwires a small set of performance critical methods. That is, the meaning of certain messages is fixed and the source code for these messages is ignored. In particular, Smalltalk hardwires the methods implementing `if` and `while` control structures (plus a few other loop control structures), and it also hardwires integer arithmetic. By doing so, the system can special-case (and greatly speed up) these frequent operations without implementing general inlining optimizations. In contrast, the NIC does not special-case any operation and always performs message sends for these operations.

- SELF programs execute more message sends since access to instance variables is always performed through messages, whereas Smalltalk methods can directly access instance variables in the receiver without sending messages. However, since the NIC inlines accesses to instance variables in the receiver, there shouldn’t be that many extra sends, certainly not enough to account for an order-of-magnitude performance difference.

- The code generated by the NIC could be more inefficient than the code generated by the ParcPlace Smalltalk compiler. For example, the Smalltalk compiler uses type prediction for arithmetic messages and generates inline code for the integer case, whereas the NIC doesn’t type-predict and never inlines primitives. Also, the NIC uses SPARC’s register windows whereas ParcPlace Smalltalk does not. However, it seems unlikely that special-casing integer arithmetic and differences in local code quality would account for anything near a factor of ten in performance difference.

4.4.1 Smalltalk without hardwired control structures

From this cursory examination of language and implementation differences, it appears that only the first—Smalltalk’s hardwiring of control structures—could have a drastic performance impact. To verify our suspicion that the hardwiring of `if` and `while` is responsible for most of the performance difference, we disabled these optimizations in the

ParcPlace Smalltalk system (version 4.0) so that all control structures generated real message sends. Furthermore, we changed the definitions of `whileTrue` et al. to resemble their SELF equivalents. The latter step was necessary because these methods are recursive in the standard Smalltalk system, so that their execution would be unnecessarily inefficient. Instead, they were implemented using a new `primitiveLoop` control structure that was recognized and open-coded by the changed compiler. For example, the implementation of `whileTrue:` was as follows:

```
whileTrue: aBlock
    "repeat aBlock as long as the receiver block yields true"
    [self value ifFalse: [ ^ nil ].     "test condition and exit if false"
     aBlock value.
    ] primitiveLoop
```

The changed Smalltalk compiler transforms the `primitiveLoop` message to a simple backward jump to the beginning of the method, just like the `_Restart` primitive that SELF uses to implement iteration. After these changes, the implementation of all important Smalltalk control structures was similar to their SELF counterparts, although the latter were still generally less optimized. For example, `whileFalse:` is implemented via `whileTrue:` in SELF but directly via `primitiveLoop` in Smalltalk. Since we were only interested in a rough estimation of the performance impact of non-inlined control structures, we did not bother to reimplement all Smalltalk control structures to be the exact equivalent of their SELF counterparts.

Benchmark	slowdown factor
Richards	10.
DeltaBlue	8.1
AllCallsOn	15.
AllImplementors	3.9
ClassOrganizer	7.6
Compiler	5.5
Decompiler	5.2
PrintHierarchy	5.8

Table 4-4. Performance of ParcPlace Smalltalk without inlined control structures

The changes had a profound effect on Smalltalk's performance: whereas the original system executed `Richards` in 3.2 seconds, the changed system needed 33.1 seconds. Other programs slowed down by similar amounts (Table 4-4).† Thus, it appears that the hardwiring of these few simple control structures speeds up Smalltalk by roughly a factor of five to ten! Somewhat surprisingly, this single change closes most of the performance gap between unoptimized SELF code and Smalltalk.

4.4.2 A NIC with inlined control structures

The surprisingly pronounced result of the above experiment suggests that similar optimizations in the NIC could dramatically speed up its code. To validate this assumption, we configured the optimizing SELF compiler to generate code resembling a hypothetical NIC which inlined the `if` and `while` control structures. The intention was to inline `if` and `while` so that the control structures themselves do not involve any block creations or sends. For example, the expression

```
x < 0 ifTrue: [ doSomething ]
```

† Richards and DeltaBlue are the only SELF benchmarks that have been translated to Smalltalk. Therefore, we included a subset of the commonly used Smalltalk Macro benchmarks [85] in our comparison.

is translated into machine code resembling the following sequence:

```
    <send "x">
    <load 0>
    <send "<">
    load true, r1
    load false, r2
    cmp r1, result
    beq L1
    cmp r2, result
    beq L2
    <code handling uncommon case>
L1: <send doSomething>
L2: ...
```

For the experiments, various parts of the optimizing compiler were turned off so that the quality of the generated code resembled NIC code as much as possible. For example, local variables were not register-allocated, and no back-end code optimization was performed. Table 4-5 compares the execution times of the standard NIC with this compiler

Category	time (seconds)			
	NIC	NIC+	Smalltalk	Smalltalk
Inlined control structures?	no	yes	no	yes
Compiled SELF code	18.7	5.9	unknown	unknown
Primitives + GC	6.2	3.3	unknown	unknown
Lookup miss handler	13.0	0.0	unknown	unknown
Other	3.9	0.5	unknown	unknown
Total	41.8	9.7	33.1	3.2

Table 4-5. Performance of Richards with various system configurations

("NIC+") for the `Richards` benchmark; the execution time of the two versions of the Smalltalk system are included for comparison. Clearly, inlining the two control structures tremendously improves the performance of the compiled SELF code, not only because the compiled code is more efficient (less call overhead and fewer block creations) but also because all of the lookup misses caused by the non-inlined control structures have been eliminated.

Benchmark	inlining `if` and `while`	inlining `if / while` + better code generation
DeltaBlue	2.6	3.2
Parser	1.7	1.9
PrimitiveMaker	1.8	2.1
CecilInt	1.9	2.2
Richards	4.3	7.8
BubbleSort	3.3	3.6
Puzzle	2.8	3.3
Geometric mean	2.5	3.1

Table 4-6. Performance of improved NIC versions relative to original NIC

Table 4-6 shows that inlining `if` and `while` speeds up programs by an average of a factor of 2.5. An even better compiler that additionally performs optimizations such as register allocation and copy propagation achieves a speedup factor of 3.1. (We simulated this compiler by enabling all back-end optimizations in the optimizing SELF compiler while keeping the other restrictions.) The additional speedup gained through better code quality is modest (except for `Richards`), and thus the performance of a real implementation of a compiler that inlined just `if` and `while` should be fairly similar to the above numbers, even if the code quality differed somewhat from our experimental setup.

Why does ParcPlace Smalltalk benefit more from inlined control structures (a factor of 5-10, see Table 4-4) than does SELF (Table 4-6)? There are many differences that could contribute to this discrepancy. For example, the two sets of benchmarks are different; also, the base Smalltalk system optimizes more control structures than just `if` and `while`. Furthermore, the two systems differ in many implementation details; for example, the NIC uses customization whereas the Smalltalk implementation doesn't. Finally, since the Smalltalk system was never intended to run without inlining the central control structures, it might not be properly tuned for the non-inlining case. Since details of the Smalltalk implementation were not available, we could not determine the exact source(s) of the discrepancy, and therefore it is not known how closely a non-optimizing SELF compiler could approximate the non-optimizing Smalltalk compiler.

4.5 Predicting the performance of a SELF interpreter

Given the relatively slow speed of unoptimized code and the relatively large space overhead for storing the compiled code, couldn't an interpreter provide similar performance with virtually no space cost?

A SELF interpreter faces one big challenge: it must interpret message sends efficiently, since almost everything in SELF involves message sends. Two problems make this hard:

- *Without inline caching, sends are expensive*. A straightforward interpreter could not use inline caching and would have to use a lookup cache instead. Unfortunately, lookups are extremely frequent: for example, Richards sends about 8.5 million messages (not counting accesses to local variables). If we want the interpreter to run at NIC speed and allow 50% of its time to go into lookups, a lookup must take less than 42 * 0.5 / 8,500,000 s = 2.5 us, or about 55 instructions (assuming a CPI of 1.8 for the SPARCStation-2, see Chapter 8). This is certainly enough for a lookup cache probe, but it is not clear if it is enough to cover the amortized cost of misses. The cost of a real message lookup is fairly high in the current system, probably on the order of between 100 and 1000 us. Therefore, even a relatively small miss ratio of 1% would result in an amortized miss cost of 1-10 us per send.
- *The byte code encoding is very abstract*. As discussed above, there is only one `send` byte code, and even local variable accesses are expressed with this byte code. For efficient interpretation, it is probably necessary to redesign the byte code format to separate the "trivial" sends from "real" message sends, or to cache an intermediate representation as discussed below.

With a redesigned byte code format, carefully optimized lookup code, and a large lookup cache, it is probably possible (though not easy[†]) to achieve NIC speed with an interpreter. By hardwiring a few important control structures, an interpreter could possibly even surpass the current NIC. It is unclear how big the space savings would be since the redesigned method representation might use more space, and the lookup cache would be fairly big since misses are very expensive. However, it appears very likely that an interpreter would use significantly less space than compiled code.

An alternate interpreter implementation approach promises better performance at the expense of higher space usage. The interpreter could dynamically translate source methods into interpreted methods (imethods) and then interpret the imethods. The main advantage of this approach is that the imethod format could be tuned for efficient interpretation.

[†] At least one SELF interpreter has been implemented elsewhere [122], and its author reports execution times that are "about 500 times slower than Oberon-2 code" on some small integer benchmarks. However, the interpreter was not especially optimized.

For example, it could have inline caches, which could significantly speed up sends. At the same time, the space overhead could be kept low since only the cached imethods use the expanded representation while the source methods could continue to use a more abstract and space-efficient representation.

The down side of this approach is that it introduces all the problems and overheads of cached code: the translation cost from source method to imethod, the cost of maintaining the imethod cache (allocation and deallocation, compaction, etc.), and the cost of keeping track of source changes (when a source method is changed, its imethod must be flushed). Depending on the speedup over the simple interpreter, the additional complexity and space cost may not be justified.

Alternatively, the interpreter could just add inline caching to the straightforward interpreter by adding one pointer per send byte code to cache the method last invoked by that send; each method would also cache the last receiver type.[†] Similar organizations have been used for Smalltalk interpreters [44]. Since most byte codes are sends, this approach would require roughly one word per byte code.

4.6 The NIC and interactive performance

One of the goals of adding a non-optimizing compiler to the SELF system was to reduce compile pauses, i.e. to improve the responsiveness of the system. How well does the current NIC achieve this goal? As discussed in the previous sections, there are a variety of trade-offs between compilation speed (and simplicity) and execution time. Would it be better to double compile speed or to double execution speed?

Figure 4-4 shows a timeline of a one-minute interaction with the SELF user interface. The interaction sequence was designed to stress initial responsiveness: most of the interactions were performed for the first time and with a “cold” code cache, so that new code had to be generated fairly often. Figure 4-5 shows a similar interaction sequence in a system where most of the compiled code was already present. We obtained the timelines by sampling the current activity of the system (e.g., whether it was compiling or running SELF code) 50 times per second and writing the resulting trace to a file.

Based on the timelines, we can make two observations:

- In the “cold start” timeline, compilation time outweighs execution time for the NIC. For example, during the first three seconds of Figure 4-4 the system spends a considerable fraction of its time in the NIC, generating hundreds of new compiled methods.
- In both timelines, unoptimized SELF code accounts for only a small fraction of total execution time and never runs for long periods of time. Given the very naive code generated by the NIC, it is quite surprising how little overall impact the code quality has.

Both observations suggest that a faster compiler or an interpreter could improve the responsiveness of the system even if they compromised execution speed. For example, it appears that an interpreter would be beneficial even if it executed programs twice as slowly.

4.7 Summary

The non-inlining compiler (NIC) is the first stage in the compilation process. It compiles code quickly (typically using less than 3 ms per compilation) but generates relatively slow code, significantly slower than that generated by the non-optimizing ParcPlace Smalltalk compiler. An experiment with the ParcPlace Smalltalk compiler shows that without the hardwiring of common control structures, Smalltalk execution slows down by an order of magnitude. Another experiment showed that a more sophisticated compiler that inlined (or hardwired) the `if` and `while` control structures could achieve a significant speedup (about a factor of 2.5) in SELF at the expense of added implementation

---

[†] A variant would cache the receiver type at the caller rather than at the callee by adding a second word per send bytecode. Such an organization would increase the inline cache hit ratio at the expense of roughly doubling the space overhead.

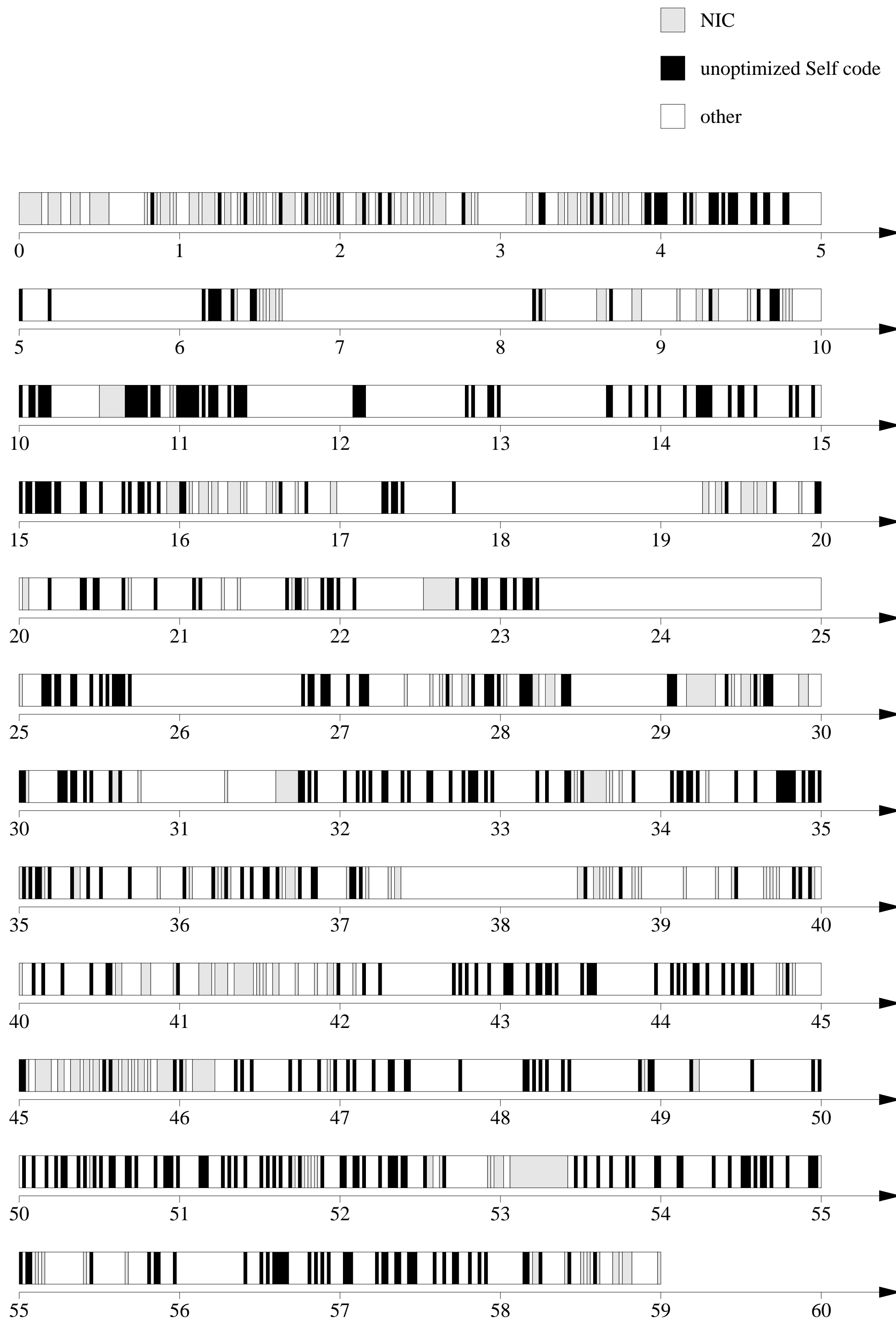


Figure 4-4. Timeline of interactive session (“cold start” scenario)

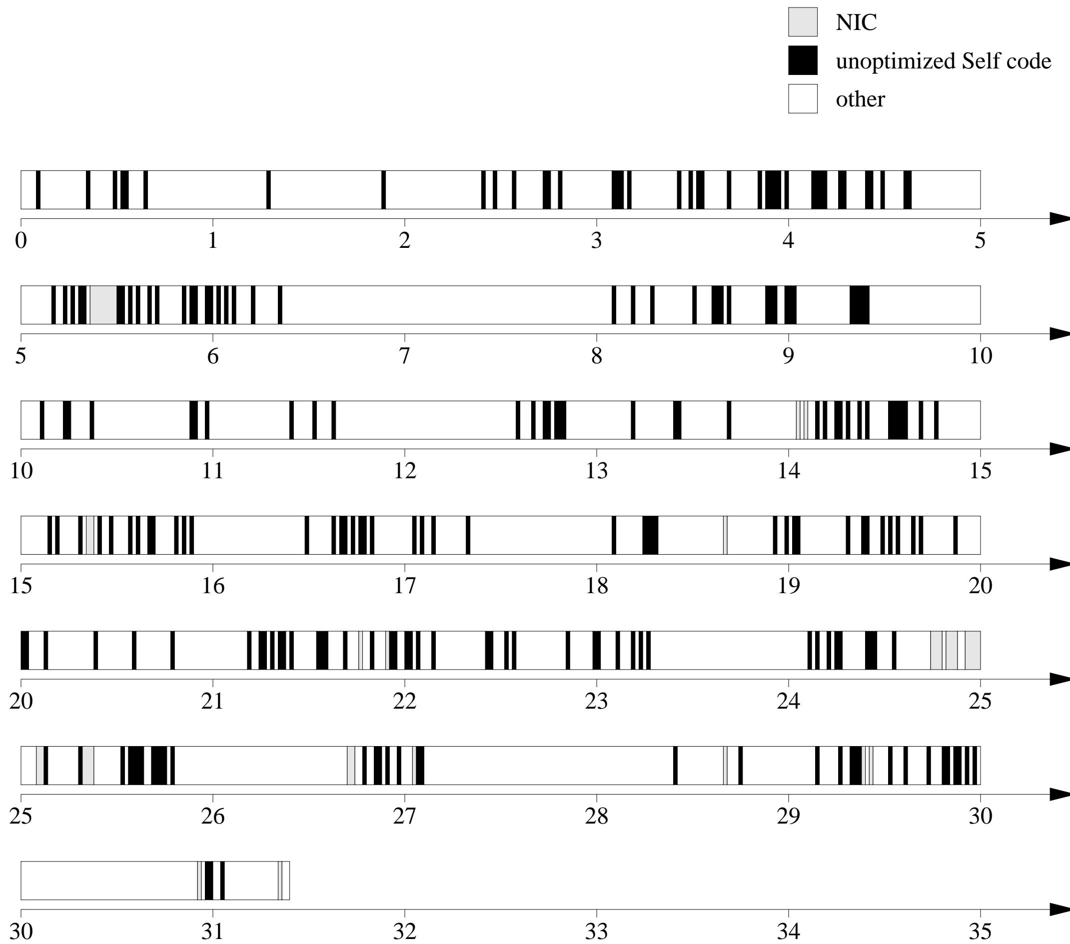


Figure 4-5. Timeline of interactive session ("warm start" scenario)

complexity or a loss of language simplicity (if the control structures were hardwired). Alternatively, a SELF interpreter could possibly achieve the current NIC speed while at the same time reducing memory usage.

Contrary to intuition, the relatively slow speed of unoptimized SELF code is not a problem in practice because the system doesn't spend much time executing unoptimized code. Timelines of interactive SELF sessions show that improvements in compilation speed would benefit the system's responsiveness more than improvements in the speed of compiled code, especially in "cold start" situations where hundreds of methods must be compiled quickly.

5. Type feedback and adaptive recompilation

Object-oriented programs are harder to optimize than programs written in languages like C or Fortran. The object-oriented programming style encourages code factoring and differential programming. As a result, procedures are smaller and procedure calls more frequent. Furthermore, it is hard to optimize these procedure calls because they use *dynamic dispatch*: the procedure invoked by the call is not known until runtime because it depends on the dynamic type of the receiver. Therefore, a compiler usually cannot apply standard optimizations such as inline substitution or interprocedural analysis to these calls. Consider the following example (written in pidgin C++):

```
class Point {
    virtual float get_x();                  // get x coordinate
    virtual float get_y();                  // ditto for y
    virtual Point distance(Point p);        // compute distance between receiver and p
}
```

When the compiler encounters the expression `p->get_x()`, where `p`'s declared type is `Point`, it cannot optimize the call because it does not know `p`'s exact runtime type. For example, there could be two subclasses of `Point`, one for Cartesian points and one for polar points:

```
class CartesianPoint : Point {
    float x, y;
    virtual float get_x() { return x; }
    // (other methods omitted)
}

class PolarPoint : Point {
    float rho, theta;
    virtual float get_x() { return rho * cos(theta); }
    // (other methods omitted)
}
```

Since `p` could refer to either a `CartesianPoint` or a `PolarPoint` instance at runtime, the compiler's type information is not precise enough to optimize the call: the compiler knows `p`'s *abstract type* (i.e., the set of operations that can be invoked, and their signatures) but not its *concrete type* (i.e., the object's size, format, and the implementation of the operations).

It is important to realize that the problem of optimizing dynamically-dispatched calls is not a result of dynamic typing; as we have seen in the above example, it also occurs in a statically-typed language. Rather, it is a result of encapsulation and polymorphism, i.e., hiding the implementation details of objects from the object's clients and providing multiple implementations of the same abstract type. A compiler for a statically-typed language can do no better[†] than generating a dynamically-dispatched call for a message send in the general case if it only has access to the program's source. Static typing enables the compiler to check that the `get_x()` message is legal (i.e., that *some* implementation is guaranteed to exist for the receiver p), but not to detect *which* implementation should be invoked. Therefore, for the problem of eliminating dynamically-dispatched calls, it is of little consequence whether we study a dynamically-typed language such as SELF or a statically-typed language such as C++.

Pure object-oriented languages exacerbate this problem because *every* operation involves a dynamically-dispatched message send. For example, even very simple operations such as instance variable access, integer addition, or array access conceptually involve message sends in SELF. As a result, a pure object-oriented language like SELF offers an ideal test case for optimization techniques tackling the problem of frequent dynamically-dispatched calls.

[†] Of course we are beating a strawman here, and our statement is not strictly true; see Section 5.8 for more details.

We have developed a simple optimization technique, *type feedback*, that feeds back type information from the runtime system to the compiler. With this feedback, a compiler can inline any dynamically-dispatched call. We have implemented type feedback for the SELF system and combined it with *adaptive recompilation*: compiled code is initially created in unoptimized form to save compile time, and time-critical code is later recompiled and optimized using type feedback and other optimization techniques. Although we have implemented type feedback only for SELF, the technique is language-independent and could be applied to statically-typed, non-pure languages as well. The rest of this chapter describes the basic ideas behind this work without going into too much SELF-specific detail. The next chapter will discuss the implementation details of the new SELF system that is based on the ideas described here.

5.1 Type feedback

The main implementation problems of object-oriented languages (or other languages supporting some form of late binding, such as APL) arise from the paucity of information statically available at compile time. That is, the exact meaning of some operations cannot be determined statically but is dependent on dynamic (i.e., runtime) information. Therefore, it is hard to optimize these late-bound operations statically, based on the program text alone.

There are two approaches to solving this problem. The first one, dynamic compilation (also called lazy compilation), moves compilation to runtime where additional information is available and can be used to better optimize the late-bound operations. This is the approach taken by SELF and several previous systems (e.g., van Dyke's APL compiler [51][†]). Compared to these systems, SELF takes laziness one step further by not trying to do the best possible job right away (i.e., when compiled code is first needed). Instead, the system generates unoptimized code first and only optimizes later, after it has become clear that the code is used often. Besides the obvious savings in compile time, this approach makes it possible to generate better code than "eager" systems because the compiler has even more information available.

If it is not possible (or not desired) to move compilation to runtime, one can use the second, more conventional approach of moving the additional runtime information to the compiler. Typically, the information is collected in a separate run and written to a file, and the compiler is re-invoked using the additional information to generate the final optimized program.

Type feedback works with either one of these approaches. For now, we will concentrate on the first approach since it is the one used by SELF; the second approach is briefly outlined later in Section 5.6.

The key idea of type feedback is to extract type information from the runtime system and feed it back to the compiler. Specifically, an instrumented version of a program records the program's *type profile*, i.e., a list of receiver types (and, optionally, their frequencies) for every single call site in the program. To obtain the type profile, the standard method dispatch mechanism has to be extended in some way to record the desired information, e.g., by keeping a table of receiver types per call site.

Having obtained the program's type profile, this information is then fed back into the compiler so it can optimize dynamically-dispatched calls (if desired) by *predicting* likely receiver types and inline the call for these types. In our example, type feedback information would predict `CartesianPoint` and `PolarPoint` receivers for the `get_x()` call, and thus the expression `x = p->get_x()` could be compiled as

```
if (p->class == CartesianPoint) {
    // inline CartesianPoint case
    x = p->x;
} else {
    // don't inline polar point case because method is too big; this branch also covers all other receiver types
    x = p->get_x();        // dynamically-dispatched call
}
```

[†] See the section on related work on page 9 for more details.

For `CartesianPoint` receivers, the above code sequence will execute significantly faster since it reduces the original virtual function call down to a comparison and a simple load instruction.

Type feedback substantially altered our understanding of the problem of optimizing object-oriented languages. In the traditional view, implementation-level type information is *scarce*, because the program text usually does not contain enough information to tell whether p is a `CartesianPoint` or a `PolarPoint`. The new viewpoint comes from realizing that although the implementation type information initially available to the compiler is indeed scarce, the information in the running system is *abundant* after a program has been running for a while. In SELF, every inline cache or PIC contains an *exact list of the receiver types* encountered for that send.[†] In other words, no additional instrumentation is needed for type feedback if the system uses polymorphic inline caches. Therefore, a program's type profile is readily available: the compiler just needs to inspect a program's inline caches to know which receiver types have been encountered so far at a call site. In our example, the inline cache for the `get_x` send would contain the `CartesianPoint` and `PolarPoint` types. If the compiler has access to this type information, it can inline *any* send because all receiver types are known.

Using the type information provided by type feedback can substantially simplify compilation. Lacking type feedback, the SELF-91 compiler performed extensive type analysis in an attempt to preserve and propagate the scarce type information available to the compiler. With type feedback, the main problem no longer is how to inline a dynamically-dispatched call but how to choose which calls to inline. Relieved from the arduous burden of performing type analysis, optimizing compilers for object-oriented languages should be simpler to write without sacrificing code quality. The new SELF-93 compiler based on type feedback is less than half the size of the previous compiler, yet it can inline many more calls and thus produce faster code.

5.2 Adaptive recompilation

The SELF system uses adaptive recompilation not only to take advantage of type feedback but also to determine which parts of an application should be optimized at all. The figure below shows an overview of the compilation process of the SELF-93 system:

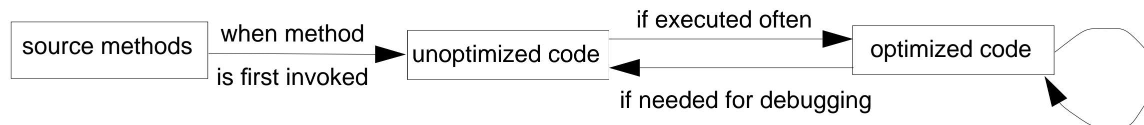


Figure 5-1. SELF-93 compilation process

When a source method is invoked for the first time, it is compiled by the simple, completely non-optimizing compiler (Chapter 4) in order to generate code very quickly. Having a very fast compiler is essential to reduce compile pauses in an interactive system using dynamic compilation, as we will see in Chapter 9. If the method is executed often, it is recompiled and optimized using type feedback. Sometimes, an optimized method is reoptimized to take advantage of additional type information or to adapt it to changes in the program's type profile.

The SELF system has to discover opportunities for recompilation without programmer intervention. In particular, the system has to decide

- when to recompile (how long to wait for type information to accumulate),
- what to recompile (which compiled code would benefit most from the additional type information), and
- which sends to inline, and which sends to leave as dynamically-dispatched calls.

The following sections discuss each of these questions. The solutions presented here all employ simple heuristics, but nevertheless work well as we shall see later.

[†] Except for megamorphic call sites, of course.

5.3 When to recompile

The SELF system uses counters to detect recompilation candidates. Each unoptimized method has its own counter. In its method prologue code, the method increments the counter and compares it to a limit. If the counter exceeds the limit, the recompilation system is invoked to decide which (if any) method should be recompiled. If the method overflowing its counter is not recompiled, its counter is reset to zero.

If nothing further were done, every method would eventually reach the invocation limit and would be recompiled even though it might not execute more often than a few times per second, so that optimization would hardly bring any benefits. Therefore, invocation counters decay exponentially. The decay rate is given as the half-life time, i.e., the time after which a counter loses half of its value. The decay process is approximated by periodically dividing counters by a constant p; for example, if the process adjusting the counters wakes up every 4 seconds and the half-life time is 15 seconds, the constant factor is $p = 1.2$ (since $1.2^{15/4} = 2$). The decay process converts the counters from invocation counts to invocation rates: given invocation limit N and decay factor p, a method has to execute more often than $N * (1 - 1/p)$ times per decay interval to be recompiled.[†]

Originally, counters were envisioned as a first step, to be used only until a better solution was found. However, in the course of our experiments we discovered that the trigger mechanism ("when") is much less important for good recompilation results than the selection mechanism ("what"). Since the simple counter-based approach worked well, we did not extensively investigate other mechanisms.[‡] However, there are some interesting questions relating to invocation counter decay:

- Is exponential decay the right model? Ideally, the system would recompile those methods where the optimization cost is smaller than the benefits that accrue over future invocations of the optimized method.[††] Of course, the system does not know how often a method will be executed in the future, but a rate-based measure also ignores the past: a method that executes less often than the minimum execution rate will never trigger a recompilation, even if it is executed a zillion times.

- The invocation limit N should not be a constant; rather, it should depend on the particular method. What the counters are really trying to measure is how much execution time is wasted by running unoptimized code. Thus, a method that would benefit much from optimization should count faster (or have a lower limit) than a method that would not benefit much from optimization. Of course, it may be hard to estimate the performance impact of optimization on a particular method.

- How should half life times be adapted when executing on a faster (or slower) machine? Suppose that the original half-life parameter was 10 seconds, but that the system now executes on a new machine that is twice as fast. Should the half-life parameter be changed, and if so, how? One could view that faster machine as a system where real time runs half as fast (since twice as many operations are completed per second), and thus reduce the half-life to 5 seconds. However, one could also argue that the invocation rate limit is absolute: if a method executes less than n times per second, it is not worth optimizing.

[†] Assume a method's count is C just before the decaying process wakes up. Its decayed value is C/p, and thus it has to execute $C * (1 - 1/p)$ times to reach the same count of C before the decay process wakes up again. Since the method eventually needs to reach $C = N$ to be recompiled, it must execute at least $N * (1 - 1/p) + 1$ times during a decay interval.

[‡] We did consider two alternatives. The first placed counters on the edges of the call graph rather than on the nodes, providing more information to the recompilation system. Unfortunately, the space cost of edge counts was prohibitive in the SELF system because the call graph of unoptimized code is extremely large. The second approach would use PC sampling to discover time-consuming methods (similar to some profilers). However, unoptimized SELF methods are very short and numerous, and even control structures like `if` are implemented via message sending, so that the coarse timer-based profile information would not be very helpful in making good recompilation decisions.

[††] This statement is actually not quite true—while such a system would theoretically minimize total execution time, it ignores the fact that optimization cannot begin until type feedback information has accumulated. Also, it ignores interactive behavior (clustering of compilations).

- Similarly, should the half-life time be measured in real time, CPU time, or some machine-specific unit (e.g., number of instructions executed). Intuitively, using real time seems wrong, since the user's think pauses (or coffee pauses) would influence recompilation. Using CPU time has its problems, too: for example, if most of the time is spent in the VM (e.g., in garbage collection, compilation, or graphics primitives), the half-life time is effectively shortened since compiled methods get less time to execute and increase their invocation counters. On the other hand, this effect may be desirable: if not much time is spent in compiled SELF code, optimizing that code may not increase performance by much (except, of course, if the optimizations reduce the VM overhead, e.g., by reducing the number of block closures created, thus reducing allocation costs and garbage collections).

Although these questions raise many interesting issues, they are not addressed further here, mostly because the simple scheme works well and because of time limitations. However, we believe that they are an interesting area for future research, and investigating them may ultimately lead to better recompilation decisions.

5.4 What to recompile

When a counter overflows, the recompilation system is invoked to decide which method to recompile (if any). A simple strategy would be to always recompile the method whose counter overflowed, since it obviously was invoked often. However, this strategy would not work well. For example, suppose that the method overflowing its counter just returns a constant. Optimizing this method would not gain much; rather, this method should be inlined into its caller. So, in general, to find a "good" candidate for recompilation, we need to walk up the call chain and inspect the callers of the method triggering the recompilation.

5.4.1 Overview of the recompilation process

Figure 5-2 shows an overview of the recompilation process. Starting with the method that overflowed its counter, the recompilation system walks up the stack to find a "good" candidate for recompilation (the next section will explain what "good" means). Once a recompilee is found, the compiler is invoked to reoptimize the method, and the old version is discarded. (If no recompilee is found, execution continues normally.) During the optimizing compilation,

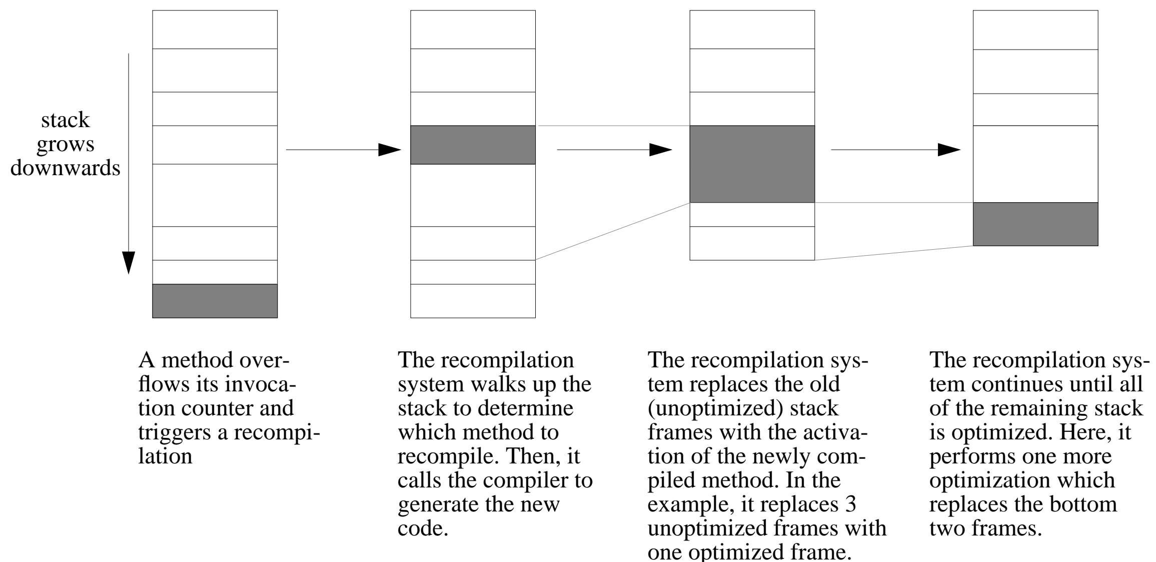


Figure 5-2. Overview of the recompilation process

the compiler marks the restart point (i.e., the point where execution will be resumed) and computes the contents of all live registers at that point. If this computation is successful,[†] the reoptimized method replaces the corresponding unoptimized methods on the stack, possibly replacing several unoptimized activation records with a single optimized activation record. Then, if the newly optimized method isn't at the top of the stack, recompilation continues with the

newly optimized method's callee. In this way, the system optimizes an entire call chain from the top recompilee down to the current execution point. (Usually, the recompiled call chain is only one or two compiled methods deep.)

If the unoptimized methods cannot be replaced on the stack, they are left to finish their current activations, but subsequent invocations will use the new, optimized method. The main effect of failing to replace the unoptimized methods is that additional recompilations may occur if the unoptimized code continues to execute for a while. For example, if the optimized method contains a loop but cannot be placed on the stack immediately, the reoptimization system may later try to replace just the loop body with optimized code.[†]

5.4.2 Selecting the method to be recompiled

The SELF-93 system selects the method to be recompiled by examining several metrics. For any compiled method *m*, the following values are defined:

- *m.size* is the size of *m*'s instructions.
- *m.count* is the number of times *m* was invoked.
- *m.sends* is the number of calls directly made from *m*.[‡]
- *m.version* records how many times *m* has been recompiled.

The search for a recompilee can be outlined as follows. Let *trip* be the method tripping its counter, and *recompilee* be the current candidate for recompilation.

1. Start with *recompilee* = *trip*.
2. If *recompilee* has closure arguments, choose the closure's lexically enclosing method if it meets the conditions described above. This rule eliminates closures by inlining the closure's use into the closure's home. If inlining succeeds, the closure can usually be optimized away completely.
3. Otherwise, choose *recompilee*'s caller if it meets the conditions below. This rule will walk up the stack until encountering a method that is either too large or does not appear to cause many message sends to be executed.
4. Repeat steps 2 and 3 until *recompilee* doesn't change anymore.

Whenever the recompilation system considers a new recompilee *m* (in steps 2 and 3 above), it will only accept the new recompilee if it meets both of the following conditions:[††]

- *m.count* > *MinInvocations* and *m.version* < *MaxVersion*. The first clause ensures that the method has been executed enough times to consider its type information representative. The second clause prevents endless recompilation of the same method.
- *m.sends* > *MinSends* or *m.size* < *TinySizeLimit* or *m* is unoptimized. The first clause accepts methods sending many messages, and the other two accept methods that are likely to be combined with the callee through inlining.

[†] The compiler cannot always describe the register contents in source-level terms since it does not track the effects of all optimizations in order to keep the compiler simple. However, it can always detect such a situation and signal it to the recompilation system.

[†] Recall the SELF implements control structures using blocks (closures) and message sends, and so the body of a loop is a method invoked via a message send and thus can be optimized like any other send.

[‡] Note that *n.count* and *n.sends* are based on incomplete data since our system does not count the invocations of optimized methods, nor does it use edge counts.

[††] The values of the parameters that our current system uses are MinInvocations = MinSends = 10,000, TinySizeLimit = 50 instructions (200 bytes), and MaxVersion = 7.

The assumptions underlying these rules are that frequently executed methods are worth optimizing, and that inlining small methods and eliminating closures will lead to faster execution. Although the rules are simple, they appear to work well in finding the "hot spots" of applications as shown in Chapter 7. (Exploring more accurate ways of estimating the potential savings and the estimated cost of recompilation remains an interesting area for future work. Of course, in a system making decisions dynamically (at run time), the additional cost of making more accurate estimates has to be considered as well.)

The rules used by the recompilation system for finding a "good" recompilation candidate in many aspects mirror the rules used by the compiler for choosing "good" inlining opportunities. For example, the rule skipping "tiny" methods has an equivalent rule in the compiler that causes "tiny" methods to be inlined. Ideally, the recompilation system should consult the compiler before every decision to walk upwards on the stack (i.e., towards a caller) to make sure the compiler would inline that send. However, such a system is probably unrealistic: to make its inlining decisions, the compiler needs much more context, such as the overall size of the caller when combined with other inlining candidates (see Section 6.1.2 on page 51). Therefore, recompilation decisions in such a system would be expensive, and this approach was therefore rejected. However, the recompilation system and the compiler do share a common structure in the SELF-93 system; essentially, the recompiler's criteria for walking up the stack are a subset of the compiler's criteria for inlining.

After a recompilation, the system also checks to see if recompilation was effective, i.e., if it actually improved the code. If the previous and new compiled methods have exactly the same non-inlined calls, recompilation did not really gain anything, and thus the new method is marked so it won't be considered for future recompilations.

5.5 When to use type feedback information

When compiling a particular message send, locating the corresponding type feedback information is relatively simple. Determining how much to trust this information is more difficult because a method's type information may have merged types from several callers. For example, one caller may always use a method *m* with polar point arguments whereas another caller uses Cartesian points. In this case, the type feedback information for sends to that argument would contain both types. If we used the information while inlining *m* into the first caller, the compiler would specialize a send within *m* for both kinds of points even though only polar points would ever occur, wasting code space and compile time. Therefore, if a compiled method has more than N callers[†], the type information is ignored by our compiler if the inline cache contains more than M types (N, M > 0). With larger N and M, code size and compile time increase because the predicted receiver type sets are too large; with smaller N and M, the effectiveness of type feedback declines since more receiver types become unknown because the type feedback information is considered "polluted." Our implementation currently uses N = 3 and M = 2 in an attempt to find a balance between the extremes. Fortunately, most inline caches contain just one type (M = 1), so that their information can be trusted regardless of the number of callers.

Of course, even with these precautions, predicting future receiver types based on past receiver types is nothing more than an educated guess. Similar guesses are made by optimizing compilers that base decisions on execution profiles taken from previous runs [137]. However, it is likely that a program's type profile is more stable than its time profile [57]—usually, no new types are created at runtime. Thus, once the application has run for a while, new types are unlikely to appear unless an exceptional case (e.g., an error) occurs or the programmer changes the application.

5.6 Adding type feedback to a conventional system

Type feedback does not require the "exotic" implementation techniques used in SELF-93 (e.g., dynamic compilation or adaptive recompilation). If anything, these techniques make it harder to optimize programs: using dynamic compilation in an interactive system places high demands on compilation speed and space efficiency. For these reasons, the

[†] Compiled methods are linked to the inline caches that call them, so it is relatively simple to determine the number of callers.

SELF-93 implementation of type feedback has to cope with incomplete information (i.e., partial type profiles and inexact invocation counts) and must refrain from performing some optimizations to achieve good compilation speed.

Thus, we believe that type feedback is probably easier to add to a conventional batch-style compilation system. In such a system, optimization would proceed in three phases (Figure 5-3). First, the executable is instrumented to record receiver types, for example with a `gprof`-like profiler [59]. (The standard `gprof` profiler already collects almost all information needed by type feedback, except that its data is caller-specific rather than call-site specific, i.e., it does not separate two calls of `foo` if both come from the same function.) Then, the application is run with one or more test inputs that are representative of the expected inputs for production use. Finally, the collected type and profiling information is fed back to the compiler to produce the final optimized code.

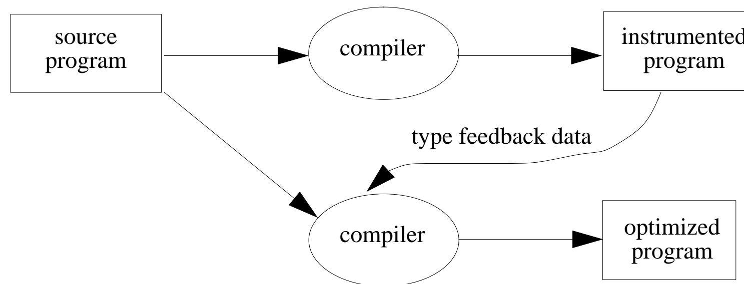


Figure 5-3. Type feedback in a statically compiled system

As mentioned above, static compilation has the advantage that the compiler has complete information (i.e., a complete call graph and type profile) since optimization starts after a complete program execution. In contrast, a dynamic recompilation system has to make decisions based on incomplete information. For example, it cannot afford to keep a complete call graph, and the first recompilations may be necessary while the program is still in the initialization phases so that the type profile is not yet representative. On the other hand, a dynamic recompilation system has a significant advantage because it can dynamically adapt to changes in the program's behavior.

5.7 Applicability to other languages

Obviously, type feedback could be used for other object-oriented languages (e.g., Smalltalk or C++), or for languages with generic operators that could be optimized with the type feedback information (e.g., APL or Lisp). But how effective would it be? We cannot give a definitive answer since it would require measurements of actual implementations, which are not available. Instead, we discuss the applicability of type feedback using Smalltalk and C++ as examples.

Type feedback is directly applicable to Smalltalk, and we expect the resulting speedups to be similar to those achieved for SELF. Despite some language differences (e.g. prototype- vs. class-based inheritance), the two languages have very similar execution characteristics (e.g., a high frequency of message sends, intensive heap allocation, use of closures to implement user-defined control structures, etc.) and thus very similar sources of inefficiency.

C++'s execution behavior (and language philosophy) is much further away from SELF, but we believe it will nevertheless benefit from type feedback. First, measurements of large C++ programs [16] have shown that calls are almost five times more frequent in C++ programs than in C programs, and that the average size of a C++ virtual function is only 30 instructions, six times smaller than the average C function. Second, two C++ programs we measured (see Section 7.3.1) slowed down by factors of 1.7 and 2.2 when using virtual functions everywhere, demonstrating that current C++ compilers do not optimize such calls well. Third, we expect that C++ programmers will make even more use of virtual functions in the future as they become more familiar with object-oriented programming styles; for example, recent versions of the Interviews framework [92] use virtual functions more frequently than previous versions.

To give a concrete example, the DOC document editor measured in [16] performs a virtual call every 75 instructions; given that a C++ virtual call uses about 5 instructions and usually incurs two load stalls and a stall for the indirect function call, we estimate that this program spends roughly 10% of its time dispatching virtual functions. If type feedback could eliminate a large fraction of these calls, and if the indirect benefits of inlining in C++ are similar to those measured for SELF (i.e., total savings are 4-6 times higher than the call overhead alone, see Figure 7-7 on page 73), substantial speedups appear possible.

For type feedback to work well, the dynamic number of receiver types per call site should be close to one, i.e., one or two receiver types should dominate. A large fraction of call sites in C++ have this property [17, 57], and it also holds in other object-oriented programming languages (e.g., Smalltalk, SELF, Sather, and Eiffel); this is the reason why *inline caching* [44, 70] works well in these languages as an implementation of dynamic dispatch. Therefore, we expect type feedback to work well for these languages; the higher the frequency of dynamically-dispatched calls, the more beneficial type feedback could be.

Type feedback also applies to languages with type-dependent generic operators (e.g., APL and Lisp). All these languages have operations that are expensive if the exact implementation types of the arguments are unknown but can be very cheap if the types are known. For example, Lisp systems could use type feedback to predict floating-point data as the common case for arithmetic operations in a floating-point intensive program instead of statically predicting integers and handling all other types with a call to a runtime routine, as current Lisp systems usually do.

5.8 Related work

5.8.1 Static type prediction

Previous systems have used static type prediction to inline operations that depend on the runtime type of their operands. For example, Lisp systems usually inline the integer case of generic arithmetic and handle all other type combinations with a call to a routine in the runtime system. The Deutsch-Schiffman Smalltalk compiler was the first object-oriented system to predict integer receivers for common message names such as "+" [44]. However, none of these systems predicted types dynamically as does our system; there was no feedback.

5.8.2 Customization

Other systems have used mechanisms similar to customization, which is a limited form of runtime type feedback. For example, Mitchell's system [97] specialized arithmetic operations to the runtime types of the operands. Similarly, APL compilers created specialized code for certain expressions [80, 51, 61]. Of these systems, the HP APL compiler [51] was the most flexible. The system compiled code on a statement-by-statement basis. In addition to performing APL-specific optimizations, compiled code was specialized according to the specific operand types (number of dimensions, size of each dimension, element type, etc.). This so-called "hard" code could execute much more efficiently than more general versions since the cost of an APL operator varies wildly depending on the actual argument types. If the code was invoked with incompatible types, a new version with less restrictive assumptions was generated (so-called "soft" code).

Similarly, Chambers and Ungar introduced customization for SELF: compiled methods were specialized to the type of their receiver, so that the same source method could have different translations if it was used with different receiver types. Type feedback surpasses customization:

- Type feedback can provide receiver type information for all sends, whereas customization only provides the type of `self` within the customized method.
- Type feedback can optimize polymorphic calls whereas customization cannot.
- Type feedback is optional—the compiler may use it for one send but not for another. Customization (as implemented in SELF) is mandatory: all code is always customized on the receiver type.

In all the other systems mentioned, runtime type information is only used when code is initially compiled. Compared to the system described in this section, compilation is *eager*: the compiler tries to do the best job it can, using the (runtime) information present at the time of the compilation; afterwards, it never revisits the code unless something exceptional happens (i.e., if a type prediction fails and the compiler has to generate less specific code). Thus, while these systems customize code using runtime type information, they do not employ type feedback. As a result, these systems have less information available to the compiler. Even though the compiler can obtain the receiver and argument types of the method or procedure being compiled, it cannot optimize operations *within* that method using runtime type information because the method has not been executed yet. In contrast, a type feedback system uses more complete runtime information; in our system, this is achieved by deferring optimization until the code has executed often, and in a batch-style type feedback system it would be achieved by executing complete "training" runs before optimizing.

5.8.3 Whole-program optimizations

Previous systems have attempted to eliminate dynamic dispatch with other means. For example, the Apple Object Pascal linker [8] turned dynamically-dispatched calls into statically-bound calls if a type had exactly one implementation (e.g., the system contained only a CartesianPoint class and no PolarPoint class). The disadvantage of such a system is that it still leaves the procedure call overhead even for very simple callees, does not optimize polymorphic calls, and precludes extensibility through dynamic linking.

Some forms of type inference can infer concrete receiver types, thus enabling the compiler to inline sends. For example, the inferencer described by Agesen et al. [5] can infer the set of concrete possible receiver types for every expression in a SELF program. Compared to type feedback information, type inference may compute better (smaller) sets for some sends because it can guarantee the absence of the unknown type. For other sends, though, the information may be inferior, for example, if only one type occurs in practice even though several types are theoretically possible. Type inference systems for real object-oriented languages are just beginning to emerge, so it is too early to assess their value for optimization purposes.

The main disadvantages of "whole-program" optimizations (such as concrete type inference or link-time optimizations) are:

- First, the whole program may not be available to the optimizer. Most programs on today's workstations are dynamically linked, and thus the compiler does not know the exact set of classes in the program since "link time" is deferred until runtime. Even if the program hardwired the dynamic link library version (i.e., refused to execute with any other version), optimization might still not be practical if only the compiled version of the library is available. Finally, even if this problem were solved, the optimizations would still prevent extensibility through dynamically-loaded user-defined libraries (e.g., such as "add-on" modules available for generic tools like word processors and spreadsheets).
- Second, whole-program optimizations can be expensive (since large amounts of code are analyzed) and thus could be hard to integrate into a rapid-turnaround programming environment.
- Third, it is unclear how well whole-program optimizers could perform without profile information to guide the optimizations. As soon as a profile is used, however, the optimizer could use the profile's type feedback information rather than performing a global type-flow analysis. In other words, it remains to be seen if global (whole-program) analyses can significantly improve performance over a type-feedback based system.

5.8.4 Inlining

Inlining has been studied extensively for procedural languages [29, 36, 39, 62, 74]. For these languages, inlining is simpler since they do not (or only rarely) use dynamic dispatch. Furthermore, inlining candidates are easier to analyze because the space and time cost of most language constructs is known. In contrast, the source reveals almost no

implementation information in a pure object-oriented language since every single operation involves dynamic dispatch, including arithmetic, boolean operations, and control structures.

5.8.5 Profile-based compilation

Dean and Chambers [40] describe a system based on the SELF-91 compiler that uses the first compilation as an "experiment" and records inlining decisions and their benefits in a database. Using the environment of a send (i.e., the known information about the receiver and arguments), the compiler then searches the database during later compilations. (The system does not use dynamic reoptimization or type feedback, so it can only take advantage of the recorded information if a method is inlined several times, or after compiled code was discarded.) Compared to our code-based heuristics, this system could be able to make better decisions since it records more information about the context of a call and about the optimizations enabled by inlining it.

Based on measurements of C++ programs, Calder and Grunwald [17] argue that type feedback would be beneficial for C++. Unfortunately, they apparently were not aware of previous work; their proposed "if conversion" appears to be identical to inline caching [44], and a proposed extension of "if conversion" is identical to type feedback (which was first presented in [70]).

Some modern compilers for conventional languages use runtime feedback in the from of execution profiling information to perform branch scheduling and to reduce cache conflicts [95, 96]. Other systems use profile information to assist classic code optimizations [30], procedure inlining [29, 94], trace scheduling [53], and register allocation [136, 98].

Hansen describes an adaptive compiler for Fortran [63]. His compiler optimized the inner loops of Fortran programs at runtime. The main goal of his work was to minimize the total cost of running a program which presumably was executed only once. All optimizations could be applied statically, but Hansen's system tried to allocate compile time wisely in order to minimize total execution time, i.e. the sum of compile and runtime. SELF tries to achieve a similar goal, namely to quickly optimize the important parts of a program while at the same time minimizing the compile pauses perceptible to the user.

5.9 Summary

An optimizing compiler for an object-oriented language cannot obtain enough information from the source code alone. For example, the compiler generally cannot determine the exact receiver types of message sends, or where the program spends most of its time. "Lazy" compilation solves the problem of optimizing such programs: since the system initially does not have enough information to do a good job in optimizing the program, it defers dealing with the hard problems until more information is known. Later, when this information is available, optimization is both simpler and more effective: using the additional information, an optimizing compiler can generate better code in less time. (We will support this contention with empirical data in chapters 7 and 8.) In other words, laziness wins.

Specifically, SELF-93 realizes lazy compilation with adaptive recompilation and type feedback:

- *Adaptive recompilation* optimizes the "hot spots" of an application. Initially, code is generated with a fast, non-optimizing compiler. If an unoptimized method is used frequently, it is recompiled so that execution will not be slowed down by running unoptimized code for long periods of time.

- *Type feedback* allows the compiler to inline any dynamically-dispatched call even if the exact receiver type is not known statically. Using type information collected during previous executions of calls (type feedback), the compiler can optimize a dynamically-dispatched call by inserting a type test for the common case(s) and then inline-substituting the callees. In other words, a dynamically-dispatched call that always found the target method in class A can be replaced by a type test (to verify that the receiver's type is A) and the inlined code of the callee.

Neither technique relies on specific features of the SELF language, and we believe both could be useful for optimizing other statically-typed or dynamically-typed object-oriented languages (e.g., CLOS, C++, Smalltalk) and for

languages with type-dependent generic operators (e.g., APL and Lisp). In addition, type feedback could also be implemented using static rather than dynamic compilation.

Together, type feedback and adaptive recompilation improve the performance of object-oriented programs by removing much of the overhead of message passing. We have designed and implemented these techniques for SELF; the next chapter details this implementation, and subsequent chapters evaluate its performance.

6. An optimizing SELF compiler

Based on the idea of using the type feedback information contained in the inline caches for optimization purposes, we have developed a new optimizing compiler for SELF. The compiler's design goals were threefold:

- *High compilation speed*. Since the SELF system uses dynamic (i.e., runtime) compilation, the compiler should be as fast as possible in order to minimize compilation pauses. Whenever possible, we avoided costly global optimizations or analyses and tried to restrict ourselves to algorithms whose time complexity is at most linear in the size of the source program.
- *Stable, high performance for large programs*. The second design goal was to surpass the performance of the previous SELF compiler for object-oriented programs (i.e., programs using data types other than integers and arrays), and especially to provide good performance for large applications. In addition, performance should be reasonably stable: minor changes in the source program (i.e., changes that do not affect the complexity of the algorithm and thus should intuitively be performance-neutral) should not significantly affect performance.
- *Simplicity.* The previous SELF compiler [21] consisted of over 26,000 lines[†] of C++ code and employed many complicated optimizations. For example, the control flow graph kept changing as a result of type analysis, which in turn influenced the results of the analysis. Consequently, the compiler was relatively hard to understand and change. Armed with the new-found type feedback information, we decided to omit type analysis from our new compiler because we anticipated that the benefits of the additional type information would more than compensate for the missing analysis. (Since we know the receiver type(s) anyway, all that type analysis would do is eliminate some type checks.) With about 11,000 lines of code, the resulting compiler is considerably simpler than the old one.

  Generally, we opted for a "RISC-style" approach when designing the new compiler and included a new optimization only if inspection of the generated code showed that it would be worthwhile to do so. Based on experience from previous SELF compilers, we included customization, splitting, and block creation optimizations from the start since they have been shown to be clear winners [21].

Figure 6-1 shows an overview of the optimizing compiler. The compiler is divided into three phases. The first phase performs high-level optimizations such as type feedback-based inlining and splitting, resulting in a graph of intermediate code nodes. The second phase then performs some common optimizations such as copy propagation, dead code elimination, and register allocation. Finally, the third phase completes the compilation process by generating machine code from the intermediate control flow graph.

6.1 The front end

The first phase of the compiler generates the intermediate control flow graph from the method objects (i.e., the byte code representation of the source methods). The basic code generation strategy is straightforward and relies on the second pass to eliminate redundancies. The main complexity of the front end lies in the inlining and splitting algorithms.

The next few sections highlight specific parts of the front end: obtaining type feedback information from the program's inline caches, deciding which methods to inline, and handling uncommon cases.

6.1.1 Finding receiver types

Whenever the compiler encounters a send, it tries to determine the possible receiver type(s)[‡] and to inline the send if possible and desirable. Expression objects (Figure 6-2) represent source-level values such as message receivers, argu-

[†] Unless noted otherwise, "lines of C++ code" excludes blank or comment lines.

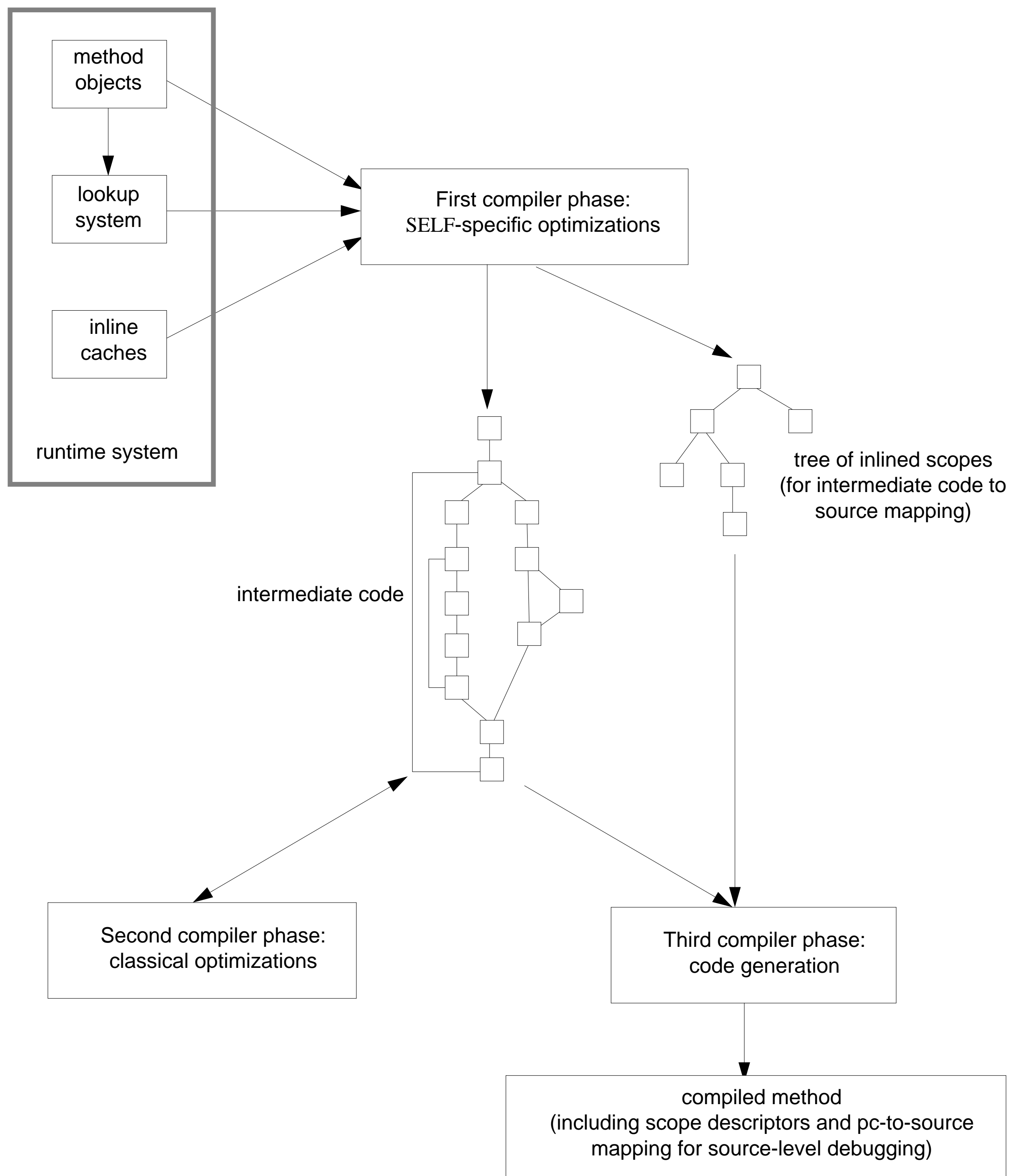


Figure 6-1. Organization of the optimizing SELF-93 compiler

ments, or results. The expression objects are propagated during message inlining to keep track of known types. For example, the two constant expressions generated by the numerical literals 3 and 4 in the expression 3 + 4 become the receiver and argument expressions of the "+" method if it is inlined.

‡ In the context of compilation, "type" always means "implementation type," i.e., a type describing the exact object layout and its set of methods. Although SELF has no explicit types at the language level, the implementation maintains type descriptors (called "maps" [22]), and each object contains a pointer to its type descriptor.

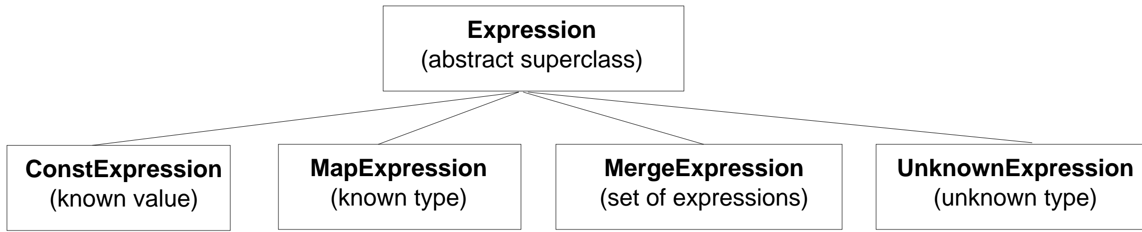


Figure 6-2. Class hierarchy of expressions

If the receiver is a constant or `self`, the receiver type is known trivially (in the latter case because customization creates a separate compiled method for each specific receiver type). If the receiver expression contains the unknown type, the compiler examines the corresponding inline cache (if any) of the previous version of compiled code. If it finds an inline cache and judges its information to be reliable (see below), the compiler adds the MapExpressions corresponding to the inline cache's receiver maps[†] to the Expression object. The resulting Expression object is always a MergeExpression since it contains at least two types, a MapExpression obtained via type feedback, and an UnknownExpression. MergeExpressions are also created whenever type information is merged as a result of a control flow merge, for example, if a method has two possible result types.

In order to find the inline cache corresponding to the send currently being inlined, the compiler keeps track of the current "recompilee scope," the current position in the old compiled code. Since every compiled method contains a description of its inlined scopes[‡], the compiler can find the new callee either by following information in the method's debugging information (if the callee was also inlined in the old compiled code) or by inspecting an inline cache.

An example will illustrate this process. Suppose that method `foo` sends `bar`, which in turn sends `baz`; also suppose that the compiler is currently recompiling `foo`. The initial situation is shown in Figure 6-3 (data structures in the runtime systems are always shown on the left, and compiler data structures are shown on the right). While generating

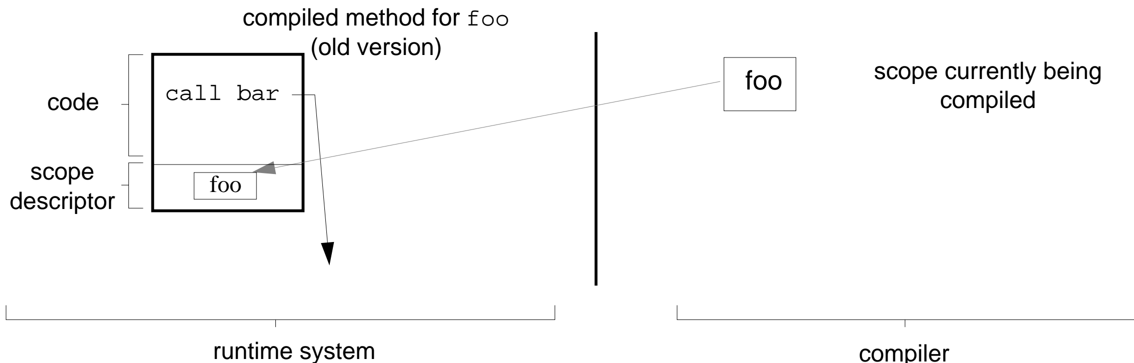


Figure 6-3. Finding type feedback information (I)

intermediate code for the `foo` source method, the compiler encounters the send of `bar`. Looking at its current position in the old compiled code for `foo`, the compiler finds the inline cache for `bar`. When inlining this send, the compiler follows the inline cache to its single target and starts generating intermediate code for `bar` (Figure 6-4).

[†] A "map" is the system's internal representation of an implementation type; all objects having exactly the same structure (i.e., the same list of slot names and the same constant slot contents) share the same map [23].

[‡] These scope descriptors are needed to support source-level debugging; see Chapter 10.

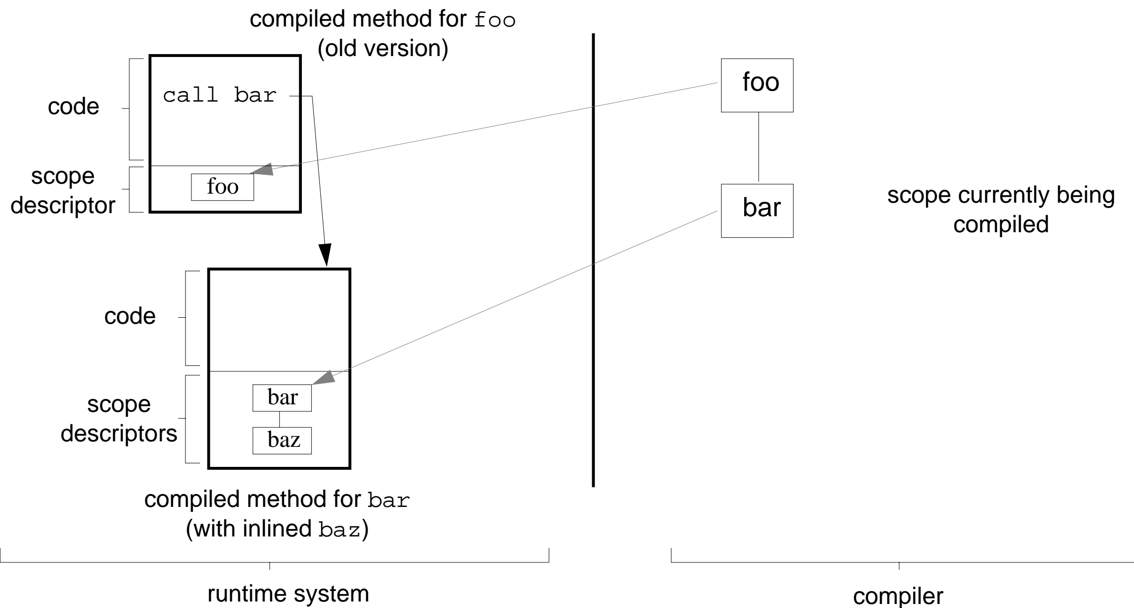


Figure 6-4. Finding type feedback information (II)

Now, the compiler encounters the send of `baz` within `bar`. This time, there is no inline cache corresponding to `baz` in the old compiled code since this send was inlined. However, the compiler can obtain the send's receiver type from the scope descriptors of the old compiled method for `bar` and continue by inlining `baz` (Figure 6-5).

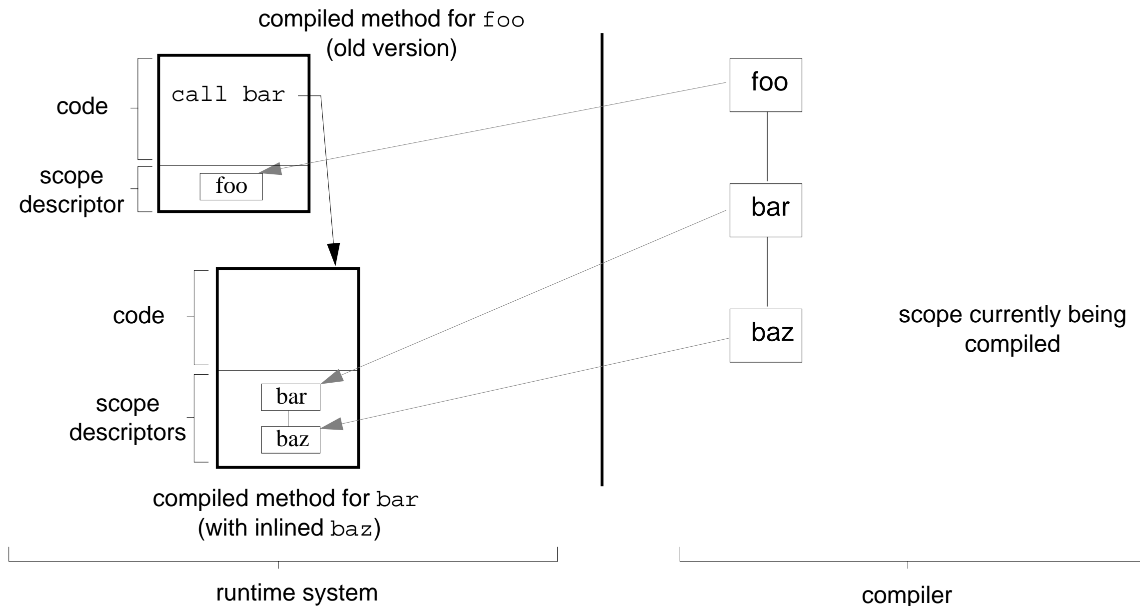


Figure 6-5. Finding type feedback information (III)

Thus, by following the send chain (inlined or not) in the old version of compiled code, the compiler can extract type feedback information with little overhead.

6.1.2 Code-based inlining heuristics

Once the compiler has determined the likely receiver types, it performs lookups to find the target methods. If the lookup with a given receiver type is successful, the compiler has to decide whether to inline the target method or not. (The compile-time lookup may fail for several reasons, such as dynamic inheritance.[†]) This is a difficult decision for several reasons:

- It is hard to estimate the impact of the decision based on that method alone, because inlining the method may cause other methods to be inlined as well, and the compiler cannot easily determine what these other methods are. SELF's pure message-based language model makes predictions especially hard because almost every source token is a dynamically-dispatched send whose target (and thus cost) is not yet known.
- For the same reasons, it is hard to determine from the method's source how much execution time would be saved if the send was inlined. Only if the method is very small (fewer than one or two sends) is it likely that inlining won't cost much in terms of code size and will give good speedups.
- Finally, even if we could accurately estimate the impact of inlining this particular send, the overall performance impact also depends on the result of other inlining decisions. For example, inlining a particular send may be beneficial in one case but may hurt performance in another case because other inlined sends increase the register pressure so much that important variables have to be stack-allocated. Another important constraint is compile time—the more inlining, the longer the compile pause. This is true beyond single compilations: if a method is called from several places and is inlined erroneously, it will slow down all compilations involving that method, not just a single compilation.

Given all these uncertainties, it seems unlikely that a set of simple heuristics would suffice and that the best approach would be to "try it out first." For example, the compiler could first use optimistic assumptions to build a (possibly too large) tree of inlined scopes and then look at the costs and benefits of each individual decision to come up with a pruned tree. In other words, it could try inlining some set of methods and then back out if inlining does not seem beneficial.

While this strategy promises to maximize the quality of the generated code, it is probably suboptimal in terms of system performance and system complexity. First, the inlining phase already consumes most of the compile time (see Chapter 9). Therefore, building the "optimistic," large call tree is likely to be expensive in terms of compile time. Second, "undoing" an inlining decision is tricky to implement and might significantly complicate the design of the compiler. For these reasons, we did not pursue this strategy further.

Dean and Chambers propose another strategy [40], using the first compilation as the "optimistic experiment" and to record the decisions and their benefits in an inlining database. Using the environment of the send (i.e., the known information about the receiver and arguments) the compiler could search the database for the "experiences" of previous compilations and make a better-informed inlining decision. While Dean and Chambers report encouraging results (compilation sped up by up to a factor of 2.2 while execution slowed down by at most a factor of 1.4), their study included only four programs and did not consider type feedback-based inlining, so that the usefulness of the technique for the compiler described here is unknown.[‡] An inlining database would add a considerable amount of code to the compiler: Dean and Chambers' implementation consists of about 2,500 lines of code, which would increase the size of our compiler by about 20%. Following our RISC-style design strategy, we therefore decided not to include such a mechanism until its necessity was proven.

[†] Dynamic inheritance lets SELF programs change the inheritance graph at runtime, so that the method being invoked may vary from send to send even if the receiver type is constant.

[‡] A later study using more programs showed a 20% reduction of compile time with no performance degradation, or a compile time reduction of 25% with an execution performance penalty of a factor of 1.6 [40]. However, it may well be that such a mechanism would benefit our system more than the SELF-91 system used by Dean and Chambers, since our system has many more inlining opportunities.

Instead, the SELF-93 compiler uses the simple local inlining heuristics of previous SELF compilers, extended with additional feedback from the runtime system. Currently, the compiler uses two sets of heuristics to make inlining decisions, code-based heuristics and source-based heuristics.

Code-based heuristics look at the previous version of compiled code (if available) to determine the size impact of the inlining decision. The advantage of this approach is that it doesn't only look at a single source method but at a bigger picture: the compiled method for a source method A includes not only code for A but also that of inlined calls. Furthermore, it lets us observe the size of real compiled code rather than the vaguely defined notion of "source method size."† Thus, its size is a better approximation of the code size impact of the inlining decision than an estimate based on source code metrics alone. In a sense, code-based heuristics employ compiled methods as primitive "inlining databases." Naturally, only optimized methods can be used since the size of unoptimized methods is a bad predictor of the size of optimized code (because almost every send is implemented as a ten-word inline cache).

Currently, the code-based heuristics consist of four rules:‡

1. If the estimated total size of the method being compiled already exceeds a threshold (currently 2,500 instructions), reject the method for inlining. This rule is designed to limit the maximum size of a compiled method. (The size estimate is based on the sum of node costs of the intermediate control flow graph.)

2. If the callee is small (less than 100 instructions, or 150 instructions if the method has block arguments), accept it for inlining.

3. If the callee is larger than the threshold, reject it, except if the estimated total size of the method being compiled remains small enough even if the callee is inlined. This exception allows large methods to be inlined if they are called from a small "wrapper" method, so that the total size won't be much larger than the callee itself.

4. If none of the above rules apply (or if no optimized callee is available), the compiler uses source-based heuristics as a fallback. The source length is defined by the number of message sends, with corrections for "cheap" sends like local slot accesses. Methods not exceeding a threshold will be inlined as long as the estimated total size allows it (see rule 1 above). Some well-known control structures are treated specially (for example, the message implementing a `for` loop).

In general, the first recompilation will mostly use source-based heuristics since the old code is completely unoptimized and thus cannot be used for the code-based heuristics. However, code-based heuristics are frequently used during the later stages of recompilation, i.e., when replacing an optimized method with a newer one. It might be possible to improve the effectiveness of the code-based heuristics by keeping a record of previous compiled method sizes (i.e., an "inlining database" just containing the size of optimized compiled methods), but we did not try to implement such a mechanism.

Compilers for more conventional languages (e.g., GNU C [120]) have used the size of intermediate code to guide inlining decisions, making the decision before or after optimizing the intermediate code [109]. Such an approach depends on having the intermediate code of the inlining candidates available at compile time; this is not the case in SELF because the inlining candidates were compiled at an earlier time.

6.1.3 Splitting

Splitting avoids type tests by copying parts of the control flow graph [23, 22, 24]. Our main motivation for using splitting was that it allows the compiler to generate good code for boolean expressions and `if` statements with relatively

† This is especially important because object code only remotely resembles source code—some sends expand to zero machine instructions while others may expand to hundreds of instructions.

‡ An additional rule limits the inlining of recursive calls. Strictly speaking, this rule isn't essential since inlining would stop once the maximum estimated method size is reached, but we decided to include it anyway since it results in better compile-time and execution performance.

little effort. In Chambers' terminology, our compiler uses extended reluctant splitting [21], i.e., it may split a send even if the send does not immediately follow the merge in control flow ("extended splitting," see Figure 6-6), but it does not split a send unless it is profitable ("reluctant splitting"). The current system only splits if the amount of copied code ("other code" in Figure 6-6) is small; the limit currently is 20 instructions.

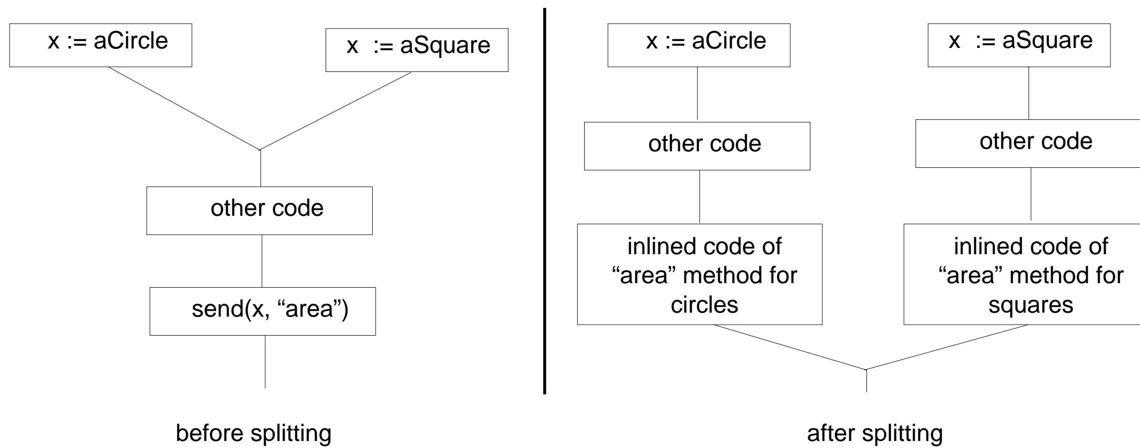


Figure 6-6. Splitting

Unlike the previous SELF-91 compiler, our compiler does not perform loop splitting, i.e. it will not create several copies of a loop body, each of them specialized for specific types. We did not include loop splitting for two reasons. First, we were not convinced that it could lead to significant speedups in nontrivial programs, since these programs are not usually dominated by the very small monomorphic loops where loop splitting would really pay off. Second, loop splitting would be relatively expensive to implement since it requires type analysis to realize its benefits [21].

Instead of performing splitting in the front end, we could obtain similarly good code with back end optimization. Heintz [64] proposes such an optimization (called "rerouting predecessors") for a Smalltalk compiler, and similar back-end optimizations for conventional languages are well-known [3, 10]. We have opted for splitting because it can achieve good code without extensive analysis and thus helps in building a fast compiler.

6.1.4 Uncommon traps

In SELF, many operations have many different possible outcomes, and the compiler creates specialized code for the common cases in order to improve efficiency. For example, the compiler specializes message sends according to type feedback information and generates optimized code for the predicted (common) cases. However, how should *uncommon* cases be compiled? What if an integer addition overflows, or if a send has a receiver type never seen before?

6.1.4.1 Ignoring uncommon cases helps

An obviously correct strategy would be to always include an "otherwise" clause in all type tests (or overflow checks) and include unoptimized code for that case. Typically, this code would perform a non-inlined message send. For example, a send optimized by type feedback would look like this:

```
if (receiver->type == PredictedType) {
    <optimized code for this case>
} else {
    <non-inlined message send>
}
```

However, this simple strategy has several serious drawbacks:

- *Code size*. Non-inlined sends (i.e., inline caches) consume a lot of code space (at least ten instructions). Since many sends are inlined using type predictions, the code to handle infrequent cases could consume a significant portion of total code space.
- *Compile time*. Similarly, a larger control flow graph containing all "otherwise" branches would slow down compilation.
- *Slowing down the common case*. As the next section will show, including code for uncommon cases may slow down the common case, especially if the send has block arguments.

The last drawback is the most severe. Figure 6-7 shows the code for the expression `i < max ifTrue: [ vector at: i ]` as it would be generated including all the uncommon cases (assuming that `i`, `max`, and `vector` are local variables and predicted to be integers).

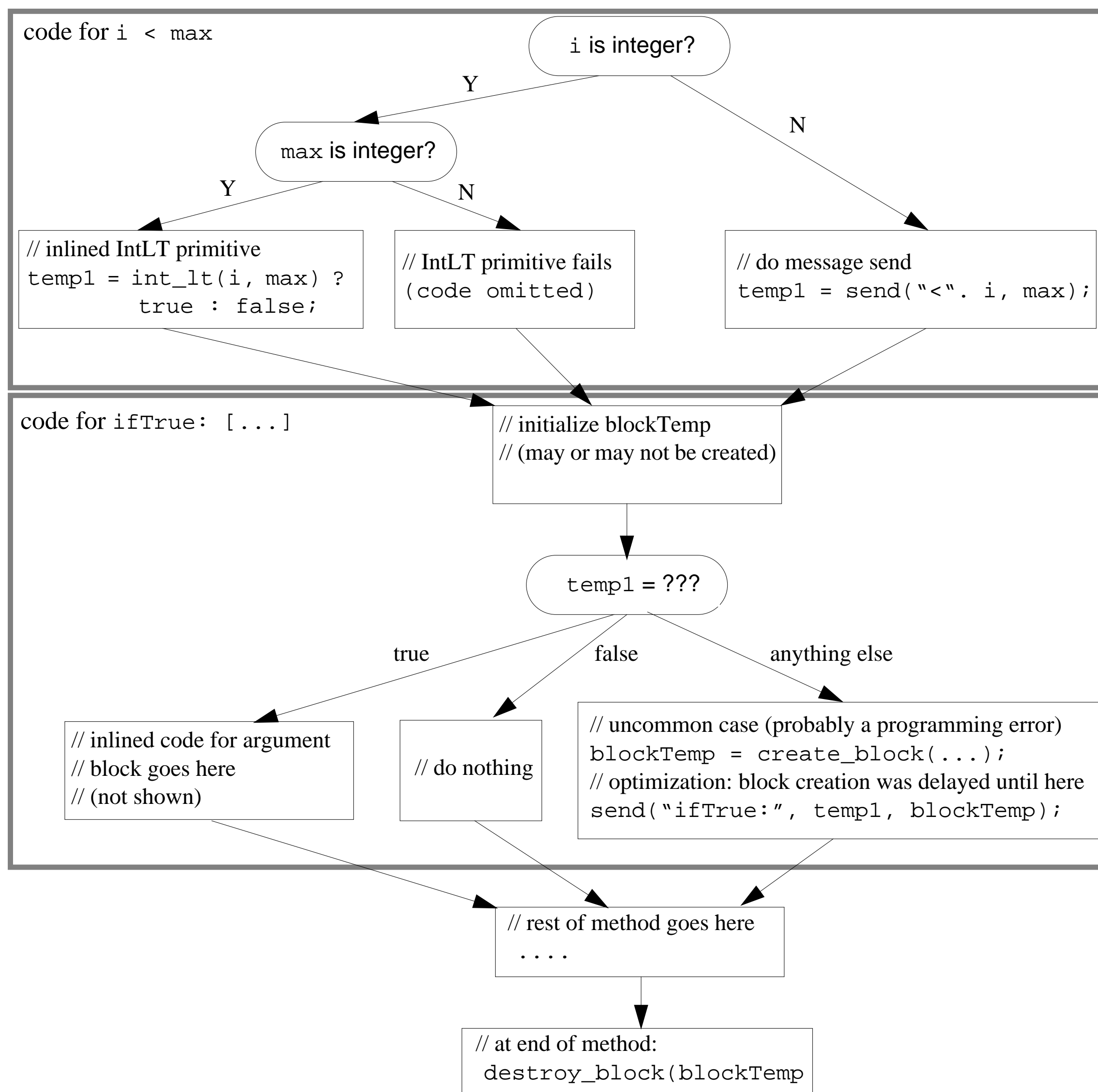


Figure 6-7. Code handling all cases

If the code only had to *detect* uncommon cases but was not required to actually *handle* them, it becomes much simpler. (This optimization was first suggested by John Maloney and first implemented in the SELF-91 compiler.) In

the SELF-93 compiler, code for cases the compiler considers unlikely to happen is replaced by a trap instruction.† The example then compiles to the code shown in Figure 6-7.

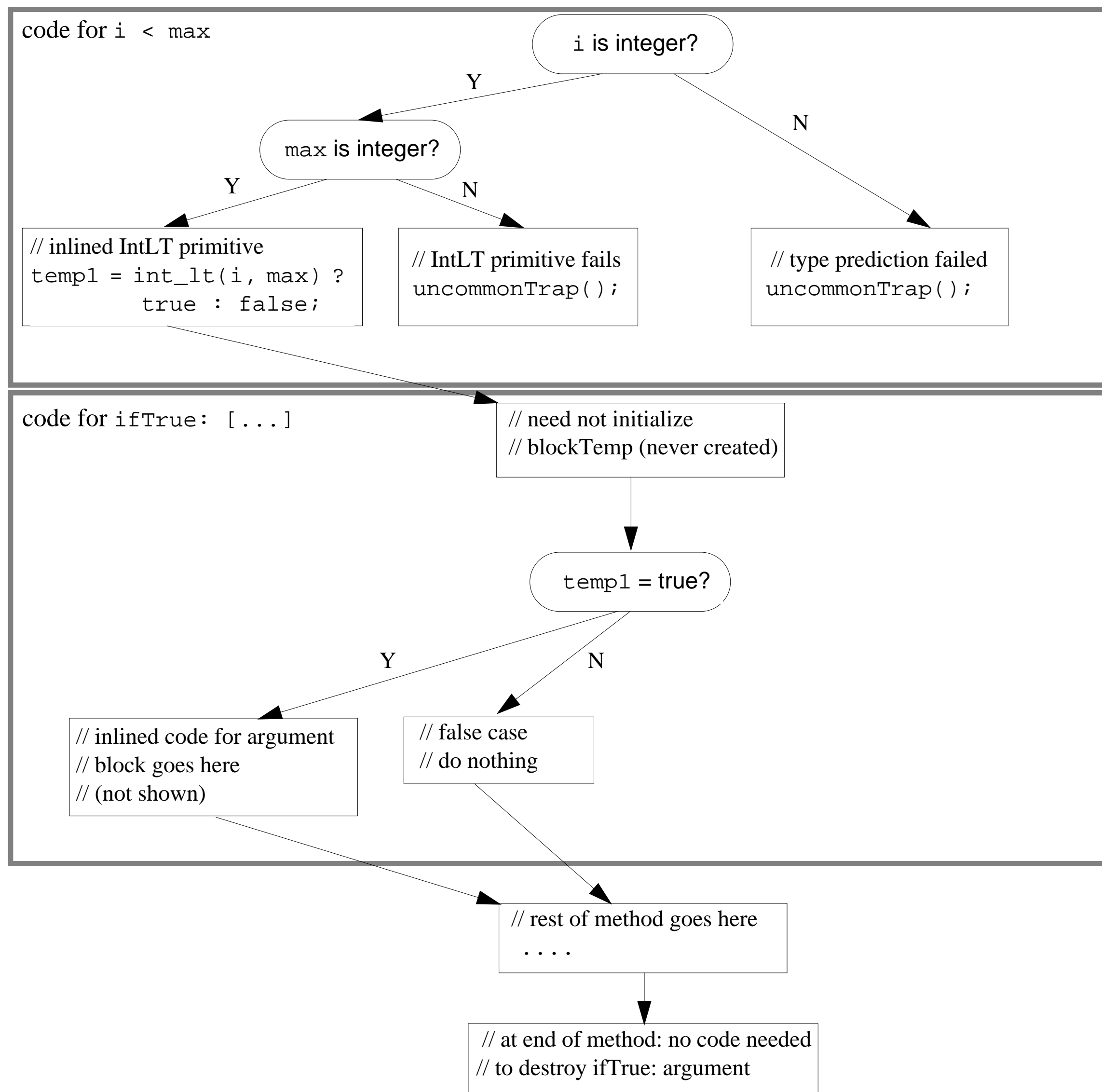


Figure 6-8. Code handling only common cases

This version of the code has several advantages; for example, it eliminates several branches and assignments as well as two sends and a closure creation. As a result, the static code size is significantly reduced, and execution is sped up. Since uncommon cases don't merge back into the common-case code, using uncommon traps also improves the accuracy of the type information. For example, the compiler now knows that `temp1` must be either `true` or `false` (before, it could have been anything since the result types of the "<" send (non-integer case) or of the failed `IntLT` primitive were unknown). Since our compiler does not perform extensive type analysis, this advantage is less pronounced; however, for the previous compiler it was essential [21].

Also, the above code eliminates all overhead related to the argument block since it is never created. In addition to the direct savings (initializing `blockTemp` and destroying the block upon exit from the method), eliminating the block

† Instead of a trap instruction, the compiler could use a call or jump; a trap saves two or three instructions per uncommon case.

has significant benefits for code generation. If the block could possibly be created, the variables accessed by the block (`vector` and `i`) cannot be register-allocated since they may be uplevel-accessed through the block's lexical link and thus have to be in memory. At the very least, the current values of these two variables would have to be written to memory before every non-inlined send, since each such send could invoke the block. By eliminating the block, the compiler therefore also eliminated these memory accesses. (Similarly, if the block assigned a local variable, that variable would have to be stack-allocated, or its value would have to be reloaded from memory after every non-inlined send.)

By now it should be clear that not requiring compiled code to handle uncommon cases can have significant performance advantages. Even though the improved code does not actually handle uncommon cases (such as non-integer `i`), the language semantics are preserved since the code still verifies all type predictions. However, what happens if an uncommon case does happen? Even though such cases may be rare, they are possible and need to be handled somehow.

6.1.4.2 Uncommon branch extensions

The SELF-91 compiler replaced the code of uncommon cases by a call to a runtime routine. If the uncommon case was ever executed, the runtime routine created an "uncommon extension" and patched the call to refer to this extension. The extension was a continuation of the original compiled code, i.e., it contained all the code from the point of failure to the end of the method. Thus, each taken uncommon branch got its own extension code. Unlike the main method, extensions were mostly unoptimized in order not to create further extensions. So, if the compiler predicted `i` to be an integer and the prediction failed, the code for `i + 1` would look as follows:

```
        if (i is integer) {
            int_add(i, 1);
        } else {
            goto uncommon_extension_1;
        }
        // rest of method

    // in a separate method:
    uncommon_extension_1:
        send("+", i, 1);
        // unoptimized code for rest of method
```

Whenever the type prediction for `i` failed, the code would jump to the uncommon extension, send the `+` message, and execute the rest of the extension as mostly unoptimized code. Thus, the common case was fast, but the uncommon case could be fairly slow.

This approach had two problems. First, the compiler had to strike a very careful balance when deciding what "uncommon" meant. If the compiler was too conservative (making only extremely rare cases uncommon), the common-case code would be slowed down. If it treated too many cases as uncommon, these cases would actually occur frequently and result in poor performance. For example, the previous compiler type-predicted `+` sends as always having integer receivers and made the non-integer case uncommon. Therefore, all programs using floating-point arithmetic frequently executed uncommon extensions and thus performed poorly. Thus, the system's performance was fragile: if the compiler's predictions were right, performance was good, but performance could drop dramatically if they were wrong (see Section 7.4.2 on page 77).

Second, this approach was not space-efficient. Since a new extension was created for each uncommon case, a method encountering several "uncommon" cases would need several uncommon extensions, thus duplicating code. Uncommon extensions tended to be large since they were relatively unoptimized; also, since the "uncommon" case often happened near the beginning of the method (e.g., at the first + send), the extensions would include most of the original method.

6.1.4.3 Uncommon traps and recompilation

To avoid these pitfalls, the SELF-93 compiler does not use method extensions for uncommon cases. Instead, it views uncommon cases as another form of runtime feedback; Figure 6-9 shows a summary of the process. The first few times[†] an uncommon trap is executed, it is assumed to be a rare case that need not be optimized. For example, the

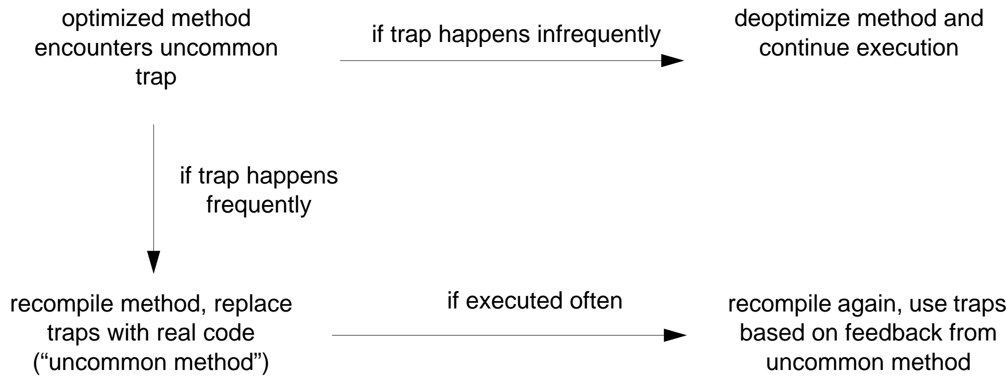


Figure 6-9. Sequence of events when handling uncommon traps

programmer could have passed an object of the wrong type to a method, resulting in a “message not understood” error. Thus, the compiled code remains unchanged, and the stack frame containing the uncommon trap is merely *deoptimized*, i.e., replaced by several unoptimized frames; then, execution continues. Deoptimization is described in detail in Chapter 10; thus, we won’t describe it further here.

If the same uncommon trap is executed more often, it is probably not so rare but the result of a compiler misprediction. For example, a program that until now used only integers could have started using floating-point data, and thus all arithmetic operations start to fail. Thus, if traps occur often, the uncommon trap handler discards the offending compiled method in addition to deoptimizing it, and creates a new version of it. The new version is more conservative in its approach and makes only very few cases uncommon; thus, it will run more slowly. To eventually reoptimize the new method, the system marks it as an “uncommon” method and lets it execute for a while.

If the new compiled method is executed often, it will eventually be selected for recompilation. At that point, the compiler can aggressively use uncommon traps again. While the method was executing, the inline caches corresponding to “uncommon” cases were filled wherever these cases actually occurred. Thus, the status of these inline caches reveals whether they can still be considered uncommon (inline cache is empty) or whether they are not (send was executed at least once). After the recompilation, the method is specialized to the new set of types used by the program.

Thus, the SELF-93 system handles uncommon cases by replacing overly specialized code with less specialized code if necessary, rather that extending the specialized code with a general extension. Handling uncommon cases in this way has two advantages. First, the compiler can be more aggressive when using uncommon traps; if the compiler writer’s “bet” turns out to be wrong, recompilation can correct the mistake. Second, it eliminates (or at least reduces) the potentially large performance difference between “good” programs (where the compiler writer’s bets work out) and “bad” programs (where the bets lose). (Section 7.4 will revisit this topic with some measurements.)

The astute reader may have noticed a pitfall in the above scheme: if the “learning period” between the first and the second (final) recompilation is too short, the uncommon method’s inline caches will not properly reflect the program’s type usage, and thus the recompiled code may soon incur an uncommon trap again because it is too specialized. Therefore, the system keeps track of the compiled code’s version and uses an exponential back-off

[†] The exact limit is user-definable; currently, it is 5.

strategy to determine the minimum number of invocations before reoptimization is allowed. Each time the method has to be recompiled because of uncommon traps, the minimum number of invocations is multiplied by a constant. Thus, the "learning period" will eventually be long enough to capture the method's type usage.

6.2 The back end

The compiler's back end performs some optimizations on the intermediate code and then generates machine code from it. The compiler does not perform full-fledged dataflow analysis or coloring register allocation in order to maximize compilation speed. The remainder of this section discusses the key aspects of the SELF-93 back end.

6.2.1 Intermediate code format

The intermediate code control-flow graph consists of about 30 different kinds of nodes. Most of the node types represent simple RISC-like operations such as assignment, heap loads and stores, or arithmetic. These nodes typically translate to one machine instruction. In contrast, some nodes represent SELF-specific operations and translate into code patterns that are several machine instructions long. Examples of such macro nodes are:

- The *Prologue* node expands into the method prologue code, including the receiver type check, stack overflow test and stack frame allocation.
- *TaggedArithmetic* nodes implement tagged integer arithmetic; the generated code checks the tags of the arguments (if necessary) and also tests the result for overflow. Tagged arithmetic nodes have two successors, one for the success case (both arguments are integers and the result didn't overflow) and the other for the failure cases.
- *ArrayAccess* nodes implement accesses to object and byte arrays, including the bounds check and the value check for byte array stores (only integers between 0 and 255 can be stored in byte arrays). Like tagged arithmetic nodes, array access nodes have two successors.
- The *MessageSend* node implements a non-inlined message send and generates an inline cache for the send. A message node has two successors, one for the normal return and the other for non-local returns.†
- A *TypeTest* node implements an N-way type test and has N+1 successors (for the N types being tested and for the "otherwise" case). Expressing the N-way branch as a single node rather than a sequence of simple two-way branches simplifies the control flow graph and allows for more efficient testing code; for example, it is fairly simple to fill branch delay slots within the testing sequence without implementing a general delay slot filling algorithm.
- *BlockCreation* and *BlockZap* nodes implement the creation and destruction of blocks. Block creation involves calling a primitive whereas block destruction is simpler (see Section 6.3.3).

Generally, we introduced a special node type if its function could not be expressed easily as a combination of simpler nodes, if it allowed the compiler to generate better code for certain constructs (which could otherwise only be generated by introducing more general optimizations that were deemed too expensive), or if it reduced the size of the control flow graph (thus improving compilation speed). High-level nodes sacrifice code quality since a lower-level RTL-style intermediate language can expose more optimization opportunities. We decided to trade compilation speed for runtime performance in this case.

All nodes use pseudo registers as their arguments; these pseudo-registers are later mapped to machine registers or stack locations by the register allocator. The pseudo registers were divided into several kinds to aid the back end's optimizations by incorporating front-end knowledge about the usage of the pseudo-registers. The most important pseudo register types are:

† See the glossary for a definition of non-local returns.

- A *PReg* is the most general form pseudo register. For example, the front end creates PRegs for the local variables of a method.
- A *SAPReg* is a singly-assigned pseudo register. Even though there may be several nodes defining a SAPReg (because of splitting), the front end guarantees that no definition will ever kill another definition. That is, exactly one defining node will be executed along any possible path through the method. Therefore, SAPRegs can be copy-propagated freely without computing reaching definition information.
- A *BlockPReg* is a pseudo register holding a block and additional information needed for the block creation and zapping nodes. Like a SAPReg, it potentially has several definitions but nevertheless can be propagated freely.
- A *ConstantPReg* is a pseudo register holding a constant; each constant value in a method has exactly one ConstantPReg (i.e., if the same constant is used twice, both uses will use the same pseudo register).

As with the intermediate code nodes, we introduced special pseudo-register types whenever they simplified the task of the back end (speeding up compilation) or allowed us to perform optimizations that would otherwise be possible only with more extensive dataflow analysis.

6.2.2 Computing exposed blocks

The first phase of the back end computes the set of exposed blocks, i.e. blocks that may be passed as arguments to other compiled methods or may be stored in the heap. This information is important for several reasons:

- If a block is *not* exposed, it is never needed as an actual object. Therefore, the compiler can omit its creation, saving considerable time (about 30 instructions, not including the garbage collection overhead) and code space (four instructions for the call to the block creation routine and one instruction to "zap" the block[†]).
- If a block is exposed, it may be needed as an actual object at some point. Therefore, the compiler must mark all variables that are uplevel-accessed by the block (i.e., local variables or parameters in lexically enclosing scopes) as uplevel-read or uplevel-written. Every non-inlined call must be considered as a potential use or definition of the uplevel-accessed variables since the block could be invoked by the callee.

Computing the set of exposed blocks is fairly straightforward. First, a pass through all nodes in the control flow graph constructs an initial set. Blocks (more precisely, BlockPRegs) are added to the set if they are assigned to another pseudo register[‡] or if they are the source of a store node. Then, we compute the transitive closure of the initial set by adding all blocks that lexically enclose blocks already in the set.

As a side-effect, this pass also sets the "uplevel-read" or "uplevel-written" flags of all pseudo registers accessed by exposed blocks. Then, it inserts a FlushNode after every definition of an uplevel-read pseudo register to make sure that uplevel reads access the correct value even if the pseudo register is allocated to a machine register. (Currently, uplevel-written variables are allocated to the stack by the register allocator and are never cached in a register.)

6.2.3 Def / use information

The next compiler pass computes the definitions and uses of each pseudo-register in a single pass through the control flow graph. The definitions and uses of a pseudo-register are recorded in a linked list and grouped by basic block so that all definitions and uses within a basic block are readily available for local copy propagation.

[†] See Section 6.3.3 for more details on block zapping.

[‡] BlockPRegs are never assigned to another pseudo-register unless they are used as an argument to a non-inlined send—if they are passed to an inlined method, the front end omits creating a pseudo-register for the callee's argument and directly uses the BlockPReg instead.

6.2.4 Copy propagation

Copy propagation, the most important classical code optimization implemented in the back end, is divided into two phases: local propagation (within basic blocks) and global propagation. During each phase, the compiler tries to propagate SAPRegs or BlockPRegs (both of which are guaranteed to have only one "real" definition) and all other PRegs that have exactly one definition. Since the compiler does not compute "reaching definitions" information, it can detect only a subset of all possible propagations. Fortunately, SAPRegs represent the majority of pseudo registers since incoming parameters and expression stack entries are guaranteed to be singly-assigned, and therefore this limitation is probably not severe. Furthermore, information akin to reaching definitions is readily available within basic blocks so that the local copy propagation phase can also propagate multiply-assigned registers.

If the pseudo-register being propagated is a ConstantPReg, the control flow graph can sometimes be simplified as a result of the propagation. For example, all but one successor branch of a TypeTest node will be eliminated if a constant is propagated into the node. (Recall that the front end does not perform type analysis and thus may generate redundant type tests.)

At the end of the copy propagation process, the compiler eliminates all definitions of unused pseudo registers if the definitions are side-effect free. (To preserve source-level semantics, the compiler cannot eliminate an expression like `i + 1` even if its result is not used, unless the compiler can prove that no integer overflow will occur.)

6.2.5 Register allocation

After performing all other optimizations, the last task remaining before code generation is register allocation. Although it is well-known that algorithms based on graph coloring [18] often produce very good register allocations, we chose not to implement such an allocator because we considered its compile-time cost to be too high.[†] Instead, the register allocator relies on usage counts.

Before performing global register allocation, a local register allocator allocates pseudo registers that are local to basic blocks. First, the local allocator scans all nodes in a basic block to determine which hardwired registers (such as the registers holding outgoing arguments[‡]) are being used in the basic block. Then, for each pseudo register local to the basic block, the allocator then tries to find a scratch register[††] that is completely unused in that basic block. Usually, there are enough free scratch registers to successfully allocate all pseudo registers local to a basic block, and thus local register allocation is very fast.

If all scratch registers are used at least once in a basic block, the local register allocator switches to a slightly more complicated approach. To determine if two pseudo registers can share the same scratch register (because their live ranges do not conflict), the allocator computes the live ranges of all scratch registers for the basic block (the live ranges are represented by bit vectors, using one bit per node in the basic block). Then, for each unallocated local pseudo register, the allocator tries to find a scratch register that is not yet live in the pseudo register's live range. If such a register is found, it is assigned and its live range bit vector is updated.

The combination of these two allocation strategies is very successful: the local register allocator usually succeeds allocating all registers that can be allocated locally. Since these pseudo registers typically represent between 50% and 90% of all pseudo registers, the local allocator significantly reduces the burden of the global allocator.

The global register allocator allocates the remaining unallocated pseudo-registers using a simple usage count algorithm. Each pseudo registers has a "weight" based on the number of times it is used, and the pseudo register with the

---

[†] The overhead includes not only the cost of performing the allocation itself but also the cost of computing the information required by the allocator, such as the register interference graph.

[‡] Our compiler passes the receiver and the first five arguments in registers.

[††] Scratch registers are caller-saved registers or temporary registers used by the compiler in certain hardwired code patterns; the global allocator never uses these registers.

highest weight is allocated first. The usage counts are weighted by loop depth nesting. For simplicity and speed, the live range of a global pseudo register is expressed in source-level terms (“scope A, byte codes 2 to 5”) rather than, say, as a bit vector indexed by intermediate nodes. To test if two live ranges overlap, it suffices to determine if one scope is a child of the other or whether the byte code ranges overlap (if the scopes are identical). The original live ranges are easy to determine: locals are live during the entire method, and expression stack entries are live from the end of the byte code producing them up to the byte code consuming them. To ensure correct allocation, the copy propagation phase updates a pseudo register’s live range to include the new use.

In conclusion, SELF-93’s register allocator is very fast and simple but nevertheless generates reasonably good allocations, as will be demonstrated by the performance measurements in the next chapter. While a more sophisticated (e.g., coloring) register allocator could undoubtedly improve performance, it would also be many times more expensive in terms of compile time. Given the compilation speed demands placed on the compiler by an interactive, exploratory system (see Chapter 9), we believe that the current allocator represents a good compromise between allocator speed and allocation quality.

6.3 Runtime system issues

This section discusses several low-level code generation issues related to the runtime system: how to efficiently test the type of an object, maintain the write barrier for generational garbage collection, and invalidate (“zap”) blocks when their lexically enclosing method returns.

6.3.1 Type tests

Type tests are frequent operations: whenever the compiler predicts a receiver type (either based on information from PICs or based on static type prediction), it has to insert a type test before the inlined body of the respective method. Thus, it is important that type tests be fast. Conceptually, a type test involves loading an object’s type and testing it against a constant. All heap-allocated objects store their map (type) in the second word of the object. Unfortunately, there are two exceptions: integers and floating point numbers are represented as immediate 32-bit values (i.e., are not heap-allocated) and thus do not explicitly contain their map; rather, the type is encoded in the tag bits. Therefore, a type test first has to test the receiver for “heap-ness” before loading the map:

```
    and    obj, #TagMask, temp                 extract object’s tag
    cmp    temp, #MemTag                       compare against pointer tag
    beq,a  isHeapObj                           is it a heap object?
    ld     [obj + #MapOffset], temp            yes (annulled delay slot†); load map
    ; not a heap object—handle immediate case (code not shown)

isHeapObj:                                     continuation of “heap object” case
    set    #Map, temp2                         load map constant
    cmp    temp, temp2                         compare maps
    beq    yes                                 is it the expected map?
    ; code for “yes” and “no” cases omitted
```

With this implementation, a simple type test consumes considerable code space and execution time. Fortunately, we can often improve on this code by speculatively loading the map (without testing the object’s tag). The load will incur an “illegal address” trap if the object is an immediate value since memory addresses must be word-aligned. If such a trap occurs, the trap handler must fill the destination register with the appropriate map (either the integer map or the float map) and continue execution at the next instruction.‡

† SPARC branch instructions have one delay slot; annulled branches (marked by the ,a suffix) execute their delay slot only if the branch is taken. In the above sequence, the load instruction is only executed if the object has the correct tag.

‡ Actually, it will suffice to zero the destination register since the only purpose is to make sure the following comparison with the predicted map will fail.

Speculative map loads significantly speed up type tests; the new code sequence is:

```
ld     [obj + #MapOffset], temp          speculatively load map
set    #Map, temp2                       load map constant
cmp    temp, temp2                       compare maps
beq    yes                               is it the expected map?
; code for "yes" and "no" cases omitted
```

When testing repeatedly for the same map, it is likely that the map constant will be allocated to a register, further reducing the test sequence to just three instructions: load, compare, and branch.

Of course, the type test is very expensive if a trap is taken (costing several thousands of instructions), so the speculative load scheme only makes sense if traps occur very rarely. For this reason, speculative loads have not been used in previous systems except for SOAR. SOAR's trap handler was much faster than a Unix trap handler (very few cycles compared to thousands of cycles) since both the hardware and the software were designed for fast traps. Unfortunately, such fast traps are not available in typical workstations.† Thus, speculative loads are a gamble since performance could degrade significantly if traps occurred often. Fortunately, PICs and type feedback-based recompilation guarantee that traps will be rare in our system. If the speculative map load is part of a PIC, we will encounter at most one trap since the PIC will be extended with the immediate case after the first trap (and, of course, the new PIC will test the tag first before loading the map). If the speculative map load is part of a compiled method, it is unlikely to get a trap because the previous version of the compiled method encountered no immediate values during its many thousand previous invocations. Should a load cause repeated traps anyway, the corresponding compiled method will be recompiled in a way similar to uncommon traps.

In conclusion, adaptive recompilation allows the system to use speculative map loads to speed up type tests. Unlike non-adaptive systems, the SELF-93 system does not have to pay a heavy price when speculation fails, since it can recompile a method (reverting to non-speculative tests) should speculation fail frequently.

6.3.2 Store checks

Generational garbage collectors need to keep track of references from older to younger generations so that younger generations can be garbage-collected without inspecting every object in the older generation(s) [89, 131]. The set of locations potentially containing pointers to newer objects is often called the *remembered set* [132]. At every store, the system must ensure that the updated location is added to the remembered set if the store creates a reference from an older to a newer object. The maintenance of this invariant is usually referred to as *write barrier* or *store check*.

For some stores, the compiler can statically know that no store check is necessary, for example, when storing an integer or when storing an old constant (since old objects never become new again). However, in the general case a store check must be executed for every store operation. Since stores are frequent, an efficient write barrier implementation is essential. SELF-91's write barrier implementation is based on Wilson's card marking scheme [139].‡ In this scheme, the heap is divided into *cards* of size 2^k words (typically, k = 5..7), and every card has an associated byte in a separate vector. A store check simply clears the byte corresponding to the location being updated. At garbage collection time, the collector scans through the byte vector; whenever it finds a marked byte, the collector examines all pointers in the corresponding card in the heap.

The advantage of the byte marking scheme is its speed. In SELF-91, a store check involves just 3 SPARC instructions in addition to the actual store:

† In fact, traps seem to become slower as hardware becomes faster [100]. However, recent designs attempt to reduce trap overhead; for example, the SPARC V9 architecture [119] allows low-overhead user-mode traps.

‡ Similar but less efficient schemes have been proposed by Sobalvarro [119] and Shaw [116].

```
st  [%obj + offset], %ptr                   store ptr into object's field

add %obj, offset, %temp                     calculate address of updated word
sll %temp, k, %temp                         divide by card size 2^k (shift left)
st  %g0, [%byte_map + %temp]                clear byte in byte map
```

This code sequence assumes that the register byte_map holds the adjusted base address of the byte map, i.e. byte_map_start_address - (first_heap_word / 2^k), thus avoiding an extra subtraction when computing the index of the byte to be cleared.

We have reduced the write barrier implementation to only two extra instructions per checked store instead of three as in the previous SELF system. Our new write barrier improves on standard card marking by relaxing the invariant maintained by the card marking scheme. The invariant maintained by standard card marking is

bit or byte i is marked <--> card i may contain a pointer from old to new

Our scheme's relaxed invariant is

bit or byte i is marked <--> cards $i..i+l$ may contain a pointer from old to new

where l is a small constant (1 in our current implementation). Essentially, l gives the store check some *leeway* in choosing which byte to mark—the marked byte may be up to l bytes away (in the direction of lower addresses) from the "correct" byte. The relaxed invariant allows us to omit computing the exact address of the updated word: as long as the offset of the updated word (i.e., its distance from the beginning of the object) is less than $l * 2^k$, we can mark the byte corresponding to the object's address rather than the byte corresponding to the updated field. Thus, the common-case store check is only two instructions:

```
st  [%obj + offset], %ptr                   store ptr into object's field

sll %obj, k, %temp                          calculate "approximate" byte index
st  %g0, [%byte_map + %temp]                clear byte in byte map
```

Usually, a leeway of one ($l = 1$) is sufficient to cover virtually all stores (except for stores into array elements). For example, the card size is 128 bytes in the SELF system and thus all stores into any of the first 30 fields of objects can use the fast code sequence.† Arrays containing pointers are treated specially: array element assignments mark the exact card of the assignment since it may be arbitrarily far away from the start of the array.

In a system that can determine object boundaries, the leeway restriction can be lifted entirely: in addition to scanning the marked card, the collector just scans forward until it finds the next object header. Again, arrays containing pointers have to be treated specially and mark the exact card of the assignment. Then, if a scanned card contains no object headers, or if the last object starting in the card is a pointer array, the collector need not scan beyond the current card.

Even though the overhead of a store check is only two instructions per store, it can still represent a significant fraction of the total garbage collection overhead. For example, store checks slow down `Richards` by about 8% on a SPARCstation-2. A detailed analysis is beyond the scope of this thesis, and we refer the interested reader to [72] for more information.

6.3.3 Block zapping

Consider the following method:

```
foo = ( | localVar | globalVar: [ localVar: 'hello' ] )
```

† The first 2 fields of every object are used by the system, so the first user-defined field has an offset of 8 bytes.

In this example, the method stores a block in the global `globalVar`. If it were possible to invoke this block after `foo` had returned, it would access `localVar` in `foo`'s activation, and thus the activation would have to be heap-allocated to make sure that `localVar` still existed. A block that is invoked after its enclosing method has returned is known as *non-LIFO block* or *upward funarg* in Lisp parlance.

In the current SELF implementation, all method activations are stack-allocated, and thus blocks may not be executed after their enclosing scope has returned. (This restriction currently exists because it is hard to implement full closures without slowing down calls.) To detect (and abort) non-LIFO block invocations, blocks are "zapped" before their enclosing method returns. Each block object contains a pointer to the enclosing scope's stack frame (i.e., a lexical link), and zapping simply clears this link. Most blocks are inlined away and thus don't need to be zapped because no block object is ever created. Some blocks are always created and can be zapped with a simple store instruction. A third category of blocks, *memoized blocks*, may or may not be created depending on the execution path taken. The following pseudo-code for the expression `x do: [ foo ]` illustrates such a situation:

```
if (x is an integer) {
    "The compiler inlined the call of do: and the only use of the block, so that the block is not created"
} else {
    "The type of x is unknown, so we need to perform a real send of do: and create the block in order to
    pass it as an argument"
}
```

Now, before returning from the method containing the above expression, is there a block to zap or not? The previous SELF system solved this problem by initializing the block's location to zero and testing it before executing the zapping store, resulting in the following three-instruction sequence:

```
    cmp    block, #0                      was the block created?
    bne,a  cont                           suppress next instruction if not created
    st     #0, [block + #linkOffset]      zap the block (annulled delay slot)
cont:
```

To eliminate the compare and branch instructions, we initialize the block's location with a "dummy" block object; all locations holding not-yet-created blocks will now point to this dummy block instead of containing zero. As a result, we can unconditionally zap the block with a single store instruction, no matter whether it was created or not. If the block was not created, the store just overwrites the dummy block's lexical link:

```
    st     #0, [block + #linkOffset]      zap the block (either real or dummy)
```

The drawback of reducing block zapping from three instructions to one is an extra instruction to initialize the block's location since 32-bit constants require two instructions on the SPARC (and other RISC architectures). However, the overall cost (initialize + zap) is still lower, especially if branches are expensive. Furthermore, if several blocks need to be initialized, the address of the dummy block is usually allocated to a register, reducing initialization back to one instruction for all but the first block. Thus, SELF-93's way of block zapping is a clear improvement over previous implementations.

6.4 What's missing

After describing what is implemented in the current compiler, it is only fitting to describe what is *not* implemented. Currently, the compiler has several shortcomings that adversely affect code quality; the most important ones are listed in the sections below. Of course, there are many more optimizations that would improve performance, such as common subexpression elimination or loop-invariant code motion. However, we will ignore these optimizations in this discussion since they would require global dataflow analysis and thus slow down the compiler considerably. Therefore, we restrict ourselves to shortcomings that could be removed or at least reduced with relatively little impact on compile time. (Section 8.6 estimates the potential performance impact of these shortcomings.)

6.4.1 Peephole optimization

The generated code contains inefficiencies that could be removed by a simple peephole optimizer. Examples of such inefficiencies are

- *Unfilled delay slots*. Delay slots are only filled within fixed code sequences such as the method prologue; all other delay slots are unfilled.
- *Branch chains*. The code often contains branches that branch to other (unconditional) branch instructions instead of directly branching to the final target.
- *Extra loads*. Values may be repeatedly loaded from memory, even within the same basic block. This is especially inefficient if the loaded value is an uplevel-accessed value since an entire sequence of loads (following the lexical chain) is repeated in these cases.
- *Load stalls*. The compiler does not attempt to avoid load stalls. On the SPARCstation-2, a load takes an extra cycle if its result is used by the very next instruction. This extra cycle could be avoided by inserting an unrelated instruction between the load and the instruction using the loaded value. On the superscalar SPARCstation-10, no load stall occurs, but the instruction after the load starts a new instruction group (i.e., is not executed in the same cycle as the load), incurring a similar performance loss.

6.4.2 Repeated type tests

Since the compiler does not perform type analysis, a value may be tested repeatedly for its type even though all but the first test are unnecessary. For example, consider the following method:

```
foo: i = (| j. k |
    ...
    j: i + 1.
    k: i + 2.
    ...
)
```

Assuming that i is an integer and that the compiler uses uncommon branches, the compiler will produce the following code:

```
...
if (i is integer) {
    j = IntAdd(i, 1)
} else {
    uncommon trap
}
if (i is integer) {
    k = IntAdd(i, 2)
} else {
    uncommon trap
}
...
```

The second type test is redundant since `i` is guaranteed to be an integer at that point; if it weren't, execution would have left the compiled method at the first uncommon trap. The compiler could remove the second test without any dataflow analysis since `i` is a parameter and thus cannot be assigned (parameters are read-only in SELF).

6.4.3 Naive register allocation

The current register allocator, while fast, cannot compete with a high-quality register allocator, especially if register pressure is high. Because it does not use an interference graph, registers are not coalesced effectively, resulting in

extra register moves. Furthermore, since a register's live range is expressed in source-level terms (which are often too coarse), registers may appear to be busy in a basic block even though they are not. Fortunately, the SPARC register windows create a relatively register-rich environment, so that this deficiency usually does not result in many extra memory accesses.

6.5 Summary

The SELF-93 compiler is geared towards producing good code quickly. Three techniques help achieving this goal. *Type feedback* allows the compiler to inline more sends without extensive analysis; the type information can readily be extracted from the runtime system with little overhead. *Code-based inlining heuristics* allow the compiler to make better inlining decisions by inspecting the existing compiled version of the inlining candidate. *Uncommon traps* can be aggressively used to reduce code size and improve code quality by excluding unknown cases; adaptive recompilation allows the compiler to back off from some assumptions should they prove to be overly aggressive.

The compiler's back end is fairly conventional and uses a high-level intermediate code format rather than an RTL-like format in order to retain high compilation speed. By exploiting the single-assignment characteristics of many entities, the compiler can perform some global optimizations without computing full dataflow information. Several innovative solutions in the runtime system further improve code quality: type tests eliminate tag checking overhead by using speculative map loads, store checks for generational garbage collections are reduced to two instructions per heap store, and blocks (closures) are invalidated efficiently.

The next two chapters will evaluate and analyze the performance of the generated code, and Chapter 9 will evaluate its compilation speed and show that it is compatible with an interactive programming environment.

7. Performance evaluation

This chapter evaluates and analyzes the performance of the new SELF system by measuring the execution times of a suite of three small and six large SELF programs (see Table 7-1). With the exception of the `Richards` benchmark, all of these programs are real applications that were written without benchmarking in mind. The applications were written by different programmers and exhibit a range of programming styles; one application (`Mango`) was generated by a parser generator.

	Benchmark	Size (lines)[a]	Description
small benchmarks	DeltaBlue	500	two-way constraint solver [113] developed at the University of Washington
	PrimMaker	1100	program generating "glue" stubs for external primitives callable by SELF
	Richards	400	simple operating system simulator originally written in BCPL by Martin Richards
large benchmarks	CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function (the compiler shares about 80% of its code with the interpreter, CecilInt)
	CecilInt	9,000	interpreter for the Cecil language [27] running a short Cecil test program
	Mango	7,000	automatically generated lexer/parser for ANSI C, parsing a 700-line C file
	Typeinf	8,600	type inferencer for SELF as described in [5]
	UI1	15,200	prototype user interface using animation techniques [28][b]
	UI3	4,000	experimental 3D user interface

Table 7-1. Benchmark programs

a. Excluding blank lines
b. Time for both UI1 and UI3 excludes the time spent in graphics primitives

To evaluate the benefits of type feedback and adaptive reoptimization, we compared three different SELF implementations (Table 7-2). SELF-91 is the previous implementation using iterative type analysis [21]; this system does not use recompilation, nor does it use type feedback.[†] SELF-93 is our current system as described in the previous chapters, using dynamic recompilation and type feedback. Finally, SELF-93-nofeedback is SELF-93 with both recompilation and type feedback turned off. This is not a configuration that is normally used, but since it always optimizes all methods and does not use type feedback, we can use it to estimate the impact of type feedback by comparing it to SELF-93. Furthermore, the impact of type analysis (as used in SELF-91) can be estimated by comparing SELF-91 to SELF-93-nofeedback.

To evaluate the absolute performance of the SELF system, we also compare it to Smalltalk and C++. Smalltalk is the dynamically-typed language that comes closest to SELF in spirit. The implementation we use, ParcPlace Smalltalk-80, is a widely used commercial implementation and is regarded as the fastest Smalltalk implementation available today. To provide another point of reference, we also compare SELF-93 with the most widely used statically-typed object-oriented language, C++. We use the GNU C++ compiler [56] because it is probably the most widely used C++ implementation and generates very good code, and Sun CC, which is typical of C++ implementations based on

[†] SELF-91 could not execute the `CecilComp` benchmark unless it was run using the simulator described in Section 7.1. (We believe that the bug is related to register window flushing, which would explain why the simulated benchmark runs correctly.)

System	Description
SELF-93	The current SELF system using dynamic recompilation and type feedback; methods are compiled by a fast non-optimizing compiler first, then recompiled with the optimizing compiler if necessary.
SELF-93 nofeedback	Same as SELF-93, but without type feedback and recompilation; all methods are always optimized from the beginning.
SELF-91	Chambers' SELF compiler [21] using iterative type analysis; all methods are always optimized from the beginning. This compiler has been shown to achieve excellent performance for smaller programs [21].
Smalltalk	ParcPlace Smalltalk-80, release 4.0 (based on techniques described in [44])
C++	GNU C++ compiler (version 2.5.5), and Sun CC 2.0 (based on AT&T cfront 2.0), both using the highest optimization level (-O2 option)

Table 7-2. Systems used for benchmarking

preprocessing C++ to C. (We compare against both compilers because there are significant performance differences between the two.) Table 7-3 lists the main implementation characteristics of all systems.

System	optimizing compiler	high-quality back end	Type Feedback	Adaptive Recompilation	Customization and Splitting	Type Analysis
SELF-93	x	o	x	x	x	-
SELF-93 nofeedback	x	o	-	-[a]	x	-
SELF-91	x	x	-	-	x	x
Smalltalk	-	-	-	-	-	-
C++	x	x	-	-	-	-[b]

Table 7-3. Implementation characteristics of benchmarked systems
symbols: x = yes, o = some, - = no / none

a. Uncommon cases can trigger recompilations
b. GNU C++ can inline virtual calls in some circumstances, based on data flow analysis

7.1 Methodology

The execution times for the SELF programs reflect the performance of (re-)optimized code, i.e., they do not include compile time.[†] For the recompiling system, the programs were run until performance seemed stable (no recompilations occurred during several runs), and then the next run not involving compilations was used. SELF-91 and SELF-93-nofeedback do not use dynamic recompilation, so we used the second run for our measurements.

[†] Chapter 9 examines aspects related to compilation speed.

Unless mentioned otherwise, we follow established practice by using geometric means when summarizing performance comparisons. For example, “system A is 2 times faster than system B” means that the geometric mean of the benchmarks’ execution time ratios (time taken by B divided by time taken by A) is 2.

To accurately measure execution times, we wrote a SPARC simulator based on the spa [76] and shade [34] tracing tools and the dinero cache simulator [66]. The simulator models the Cypress CY7C601 implementation of the SPARC architecture running at 40 MHz, i.e., the chip used in the SPARCstation-2 (SS-2) workstation. It also accurately models the memory system of a SS-2, with the exception of a different cache organization. We do not use the original SS-2 cache configuration because it suffers from large variations in cache miss ratios caused by small differences in code and data positioning (we have observed variations of up to 15% of total execution time). Cache miss variations are more apparent in SELF because both compiled code and data are dynamically allocated, and small changes to the system can change the relative alignments of both code and the data, thus changing the number of conflict misses. Since none of the systems we measured specifically optimizes programs for better cache behavior, these variations can be considered random, and their presence makes it harder to accurately compare two systems or two versions of the same system.

Instead of the unified direct-mapped 64K cache of the SS-2, we simulate a machine with a 32K 2-way associative instruction cache and a 32K 2-way associative data cache using write-allocate with subblock placement.[†] Also, we assumed a write buffer (or write cache) sufficiently large to absorb all writes.[‡] As in the original SS-2 workstation, cache lines are 32 bytes long, and the cache miss penalty is 26 cycles. With the changed cache configuration, the variations become much smaller (on the order of 2% of execution time) because the caches are two-way set associative (thus reducing the number of conflict misses) and split between I– and D–caches (so there are no cache conflicts between instructions and data). Since the simulated cache configuration reduces the variability of cache misses (i.e., the number of conflict misses), its misses are mostly capacity misses. Thus, larger programs will in general still incur more cache misses and run more slowly, as they would on a real machine. In other words, the simulated cache configuration merely reduces the “noise” in our data but still realistically models a program’s cache behavior. With the reduced noise, we were able to evaluate the performance impact of compiler optimizations much more precisely.

We verified our simulator by comparing real execution times measured on an otherwise idle SPARCstation-2 workstation in single-user mode with the simulated times. When simulating the actual SPARCstation-2 cache organization, the results agreed well, differing by less than 15% in all cases; the simulated times were usually somewhat better than the measured times, probably because the simulator does not model CPU usage and cache pollution by the operating system. The simulated times using the associative cache organization were between 70-90% of the real execution times, depending on the benchmark and the particular measurement run (recall that direct-mapped caches show wide variations in cache effectiveness). We attribute this difference to the better cache organization and the ignored operating system overhead.

7.2 Type analysis does not improve performance of object-oriented programs

First, we measure the baseline speed of the new compiler by comparing SELF-93-nofeedback with the fastest previous implementation, SELF-91. Figure 7-1 shows the results (Table A-3 in Appendix A contains detailed data). Surprisingly, SELF-91 is only slightly faster (11% on average) despite its use of type analysis, more extensive type prediction, and its significantly more ambitious back end. An informal analysis of the generated code leads us to believe that much of the speedup of SELF-91 over SELF-93-nofeedback comes from the differences in the compiler

---

[†] “Write-allocate with subblock placement” allocates a cache line when a store instruction references a location not currently residing in the cache. This organization is common in current workstations (e.g., the DECstation 5000/200) and has been shown to be effective for programs with intensive heap allocation [48].

[‡] The SPARCstation-2’s write buffer organization is somewhat unfortunate and incurs high write costs compared to other common designs. To avoid skewing our data, we chose not to model the SS-2 write buffer and instead assume a perfect write buffer. Previous work (e.g., [14, 81]) has shown that writes can be absorbed with virtually no cost, and Diwan et al. [49] report a write buffer CPI contribution of less than 0.02 for a 6-element deep buffer and a cache organization similar to the DECstation 500/200.

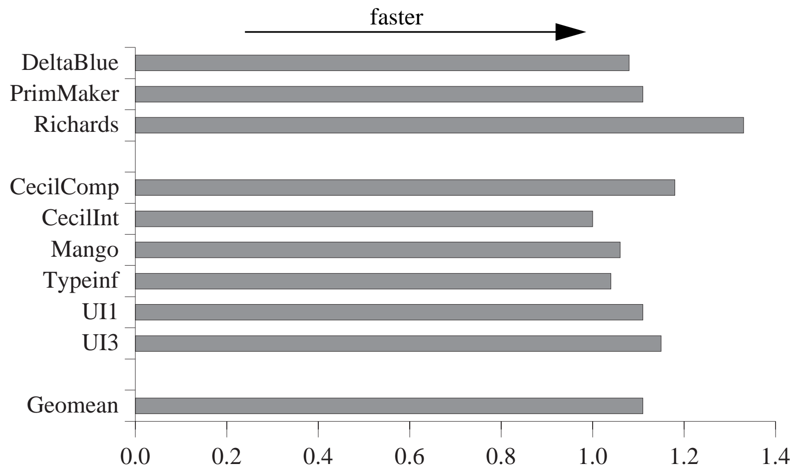


Figure 7-1. Execution speed of SELF-91 relative to SELF-93-nofeedback

back ends. For example, SELF-91 compiles a division by 2 into a shift in `Richards`' inner loop, whereas SELF-93-nofeedback does not perform this optimization. In general, code generated by SELF-91 also performs significantly fewer unnecessary register moves.

If most of the performance difference can be explained by SELF-91's superior back end, does that mean that type analysis has no performance impact? Figure 7-2 confirms that the programs in our benchmark suite do not seem to

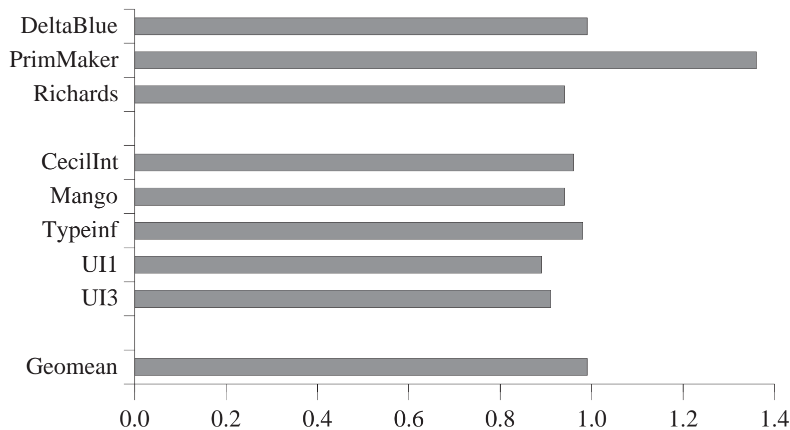


Figure 7-2. Number of sends in SELF-91 relative to SELF-93-nofeedback

benefit from type analysis: on average, SELF-91 could not inline more sends than SELF-93-nofeedback. Apparently, type analysis could not infer the receiver type of most sends.† In general, type analysis cannot infer the result types of non-inlined message sends, of arguments of non-inlined sends, or of assignable slots (i.e., instance variables). Since sends to such values are fairly frequent, a large fraction of message sends does not benefit from type analysis. For the object-oriented programs in our benchmark suite, the benefits of type analysis are very small. (Section 7.5.4 will discuss type analysis in more detail.)

† An alternate explanation for the small impact of type analysis would be that the compiler chose not to inline additional sends even though type analysis did infer their receiver types. However, we have ample evidence that this explanation is not accurate; for example, about half of the non-inlined sends in Richards are access methods that just return (or set) the value of an instance variable [70]. If the compiler had known the receiver type of these sends, it would have inlined them.

7.3 Type feedback works

To evaluate the performance impact of type feedback, we compared SELF-93 with SELF-93-nofeedback. Since SELF-93-nofeedback is competitive with SELF-91, we can be reasonably sure that any speedup obtained by type feedback is not the result of an artificially crippled SELF-93-nofeedback system but a true performance improvement. Figure 7-3 shows the results of our measurements (Table A-3 in Appendix A contains detailed data). Type feedback significantly improves the quality of the generated code, resulting in a speedup of 1.7 (geometric mean) over SELF-93-nofeedback even though SELF-93-nofeedback always optimizes all code whereas SELF-93 optimizes only parts of the code. The recompilation system appears to be highly successful in exploiting the type feedback provided by PICs to generate better code and in finding the time-critical parts of the applications.

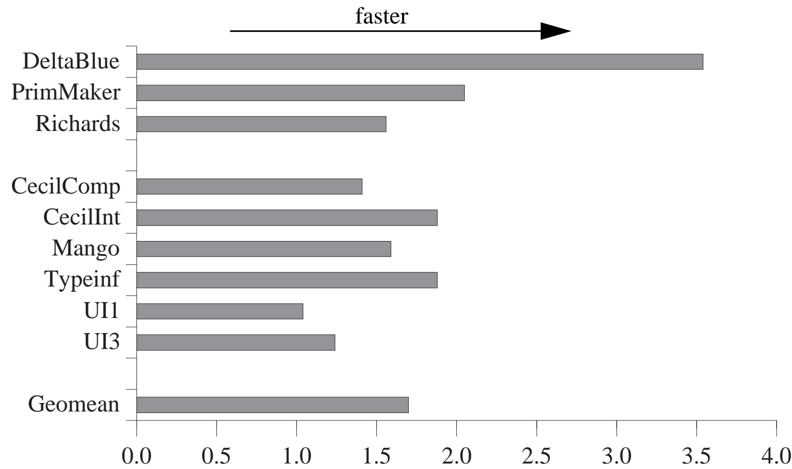


Figure 7-3. Execution speed of SELF-93 relative to SELF-93-nofeedback

Type feedback also drastically reduces the number of calls executed by the benchmark programs (Figure 7-4). On average, type feedback reduces the call frequency by a factor of 3.6.† Apparently, type feedback exposes many new inlining opportunities that SELF-93-nofeedback cannot exploit because it lacks receiver type information. For example, in SELF-93-nofeedback about half of the non-inlined sends in `Richards` are access methods that just return (or set) the value of an instance variable [70]. All of these sends are inlined in SELF-93.

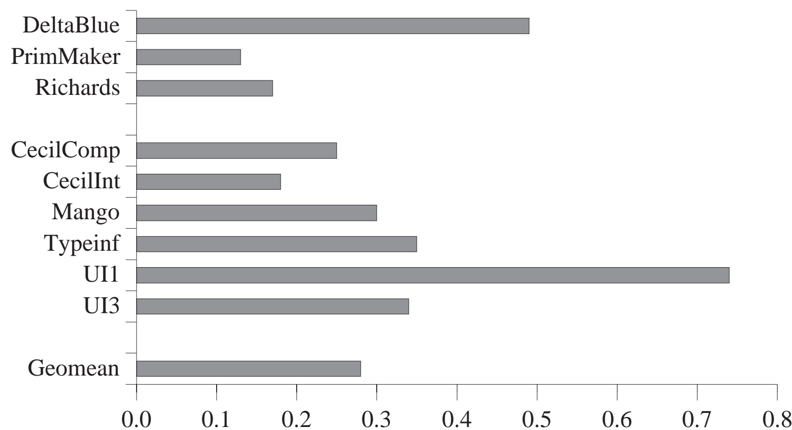


Figure 7-4. Calls performed by SELF-93 relative to SELF-93-nofeedback

† See Table A-4 in Appendix A for detailed data.

It is also interesting to look at the number of calls relative to completely unoptimized SELF. In unoptimized SELF, each message send is implemented as a dynamically-dispatched call, with the exception of accesses to instance variables in the receiver; unoptimized programs run many times slower than both SELF-91 and SELF-93 (see Chapter 4). SELF-93 executes less than 5% as many calls as unoptimized SELF (Figure 7-5). In contrast, 10-25% of the original calls remain in SELF-91 and SELF-93-nofeedback.

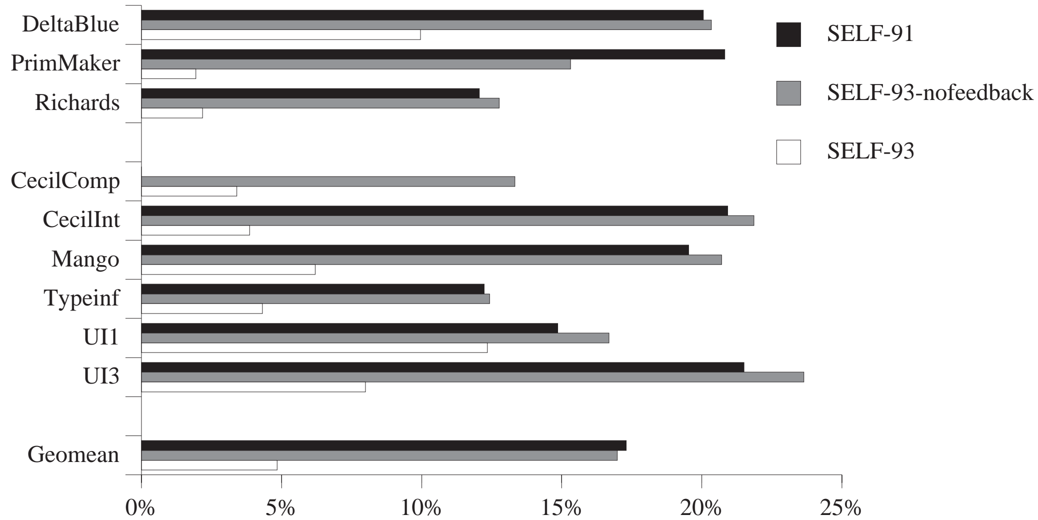


Figure 7-5. Call frequency relative to unoptimized SELF

As can be expected, there is some correlation between eliminated calls and speedup: generally, inlining a call improves performance. However, the correlation is fairly weak (see Figure 7-6). As Table A-4 in Appendix A shows,

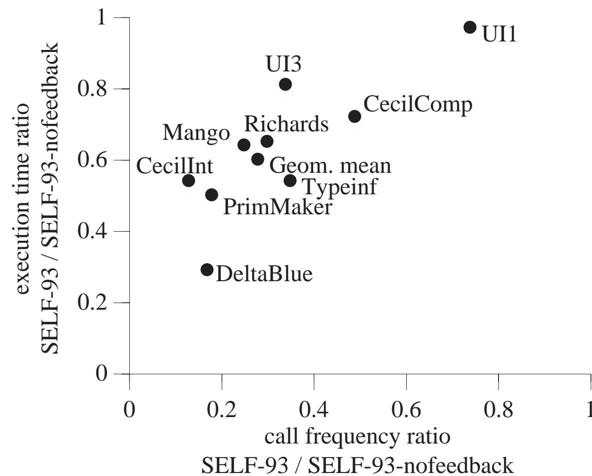


Figure 7-6. Call frequency reduction vs. execution time reduction

the time saved per call (averaged per benchmark) varies by a factor of ten, ranging from about 0.5 to 5 microseconds per call (or about 10 to 100 eliminated instructions per eliminated call). Since these are per-benchmark averages, they probably underestimate the true variance in savings per inlined call: some inlining decisions probably result in negative savings (e.g., because they increase register pressure too much, leading to more spill cycles than cycles saved by eliminating the call overhead), and others may result in much higher savings (e.g., because they allow an expensive primitive to be constant-folded). However, aggressive inlining generally significantly increases the performance of SELF programs. This effect is in marked contrast to studies of other languages (such as C or Fortran) where inlining

was found to have only small benefits, if any [29, 36, 39, 62, 74]. We attribute this discrepancy to a number of significant differences between SELF and conventional languages:

1. In SELF, procedures (methods) are much smaller on average than in C or Fortran because every operation (including very simple ones, like integer arithmetic and array accesses) involves message sends. Also, object-oriented programming encourages code factoring to separate out commonalities among related abstractions, which tends to result in shorter methods as well.

2. C and Fortran programmers may be more performance-conscious when writing their programs, guided by the assumption that "calls are expensive." In essence, the programmer hand-inlines very small (one- or two-line) methods from the beginning.

3. SELF uses closures to implement control structures or user-defined iterators, and inlining a call may enable the compiler to avoid creating a closure, thus saving at least 30 instructions. That is, inlining a single call can potentially result in significant savings even if there are no source-level redundancies (such as common subexpressions) that can be exploited as a result of the inlining. Interestingly, modern computer architectures create similar situations for conventional languages. For example, inlining a single call may dramatically speed up a Fortran program if a loop can be parallelized with the improved data dependence information [62]. The speedup does not result from reducing the execution time of the inlined code but from optimizations enabled in the caller.

Figure 7-7 shows that the sources of improved performance can vary widely from benchmark to benchmark. The x-axis shows the total execution time saving when from type feedback (i.e., the difference between SELF-93-nofeedback and SELF-93). The boxes summarize the importance of each category for the benchmarks, i.e., the contribution to the overall speedup. Depending on the benchmark, the reduced call overhead represents between 6% and 63% of the total savings in execution time, with a median of 13% and an arithmetic mean of 25% (geometric mean: 18%).† The reduced number of type tests contributes almost as much to the speedup, with a median contribution of 17% and a mean of 19%, as does the reduced number of closure creations.

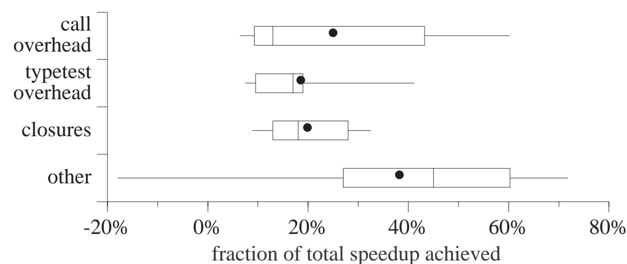


Figure 7-7. Reasons for SELF-93's improved performance

Other effects (such as standard optimizations that perform better with the increased size of compiled methods) make the greatest contribution to the speedup (with a median of 45% and a mean of 38%) but also show the largest variation. For one benchmark, the contribution is actually negative, i.e., slows down execution. Some of the possible reasons for the slowdown are inferior register allocation (because of increased register pressure), or higher instruction cache misses. All of the above measurements include cache effects.

To summarize, the measurements in Figure 7-7 show that the performance improvement obtained by using type feedback is by no means dominated by the decreased call overhead. In most benchmarks, factors other than call overhead dominate the savings in execution time. Inlining based on type feedback is an enabling optimization that allows other optimizations to work better, thus creating indirect performance benefits in addition to the direct benefits obtained by eliminating calls.

† The data assumes a savings of 10 cycles per eliminated call since we could not measure the exact savings per individual call.

7.3.1 Comparison to other systems

To provide some context about SELF's absolute performance, we measured versions of the `Deltablue` and `Richards` benchmarks written in C++ and Smalltalk. (None of the other benchmarks are available in C or Smalltalk.) We measured two C++ versions. The first version was hand-optimized by declaring functions "virtual" (dynamically dispatched) only when absolutely necessary. In the second version, all functions were declared virtual, as they implicitly are in Smalltalk or SELF. This does not mean that all function calls were dynamically dispatched; at least the GNU compiler statically binds calls if the receiver type is known.

Since it was not possible to run Smalltalk with the simulator, we could only obtain elapsed time measurements. As discussed in Section 7.1, we estimate that simulated times are about 10-25% lower than elapsed times since the simulation does not include OS overhead and simulates a better cache organization. For comparison with SELF and C, we estimated the Smalltalk time as being 75% of elapsed time.

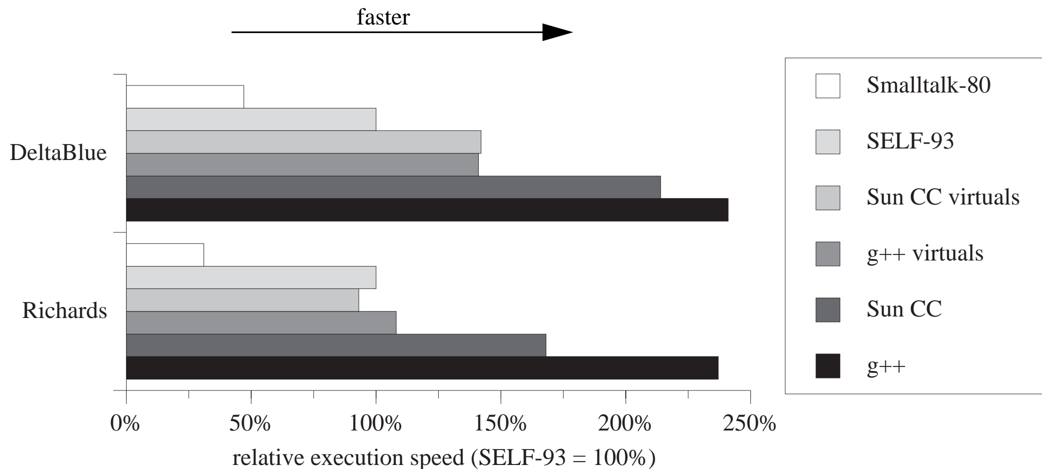


Figure 7-8. SELF-93 execution speed compared to other languages

For these benchmarks, SELF-93 runs about two to three times faster than ParcPlace Smalltalk which is generally regarded as the fastest commercially available Smalltalk system (see Figure 7-8†), even though SELF's language model is purer and thus harder to implement efficiently [21]. Furthermore, SELF-93 is only 1.7 to 2.4 times slower than optimized C++ despite the radically different language models and the fact that SELF-93's back end is clearly inferior to that of the C++ compilers. If all functions are declared virtual to approximate SELF's semantics, C++'s speed advantage is reduced significantly to between 0.9 and 1.4 times the speed of SELF-93. Although the C++ compilers could potentially perform many more classical optimizations than SELF-93, they cannot generate much more efficient code in this case because they cannot optimize dynamically-dispatched calls. Of course, the C++ programs still do not support generic arithmetic, arithmetic overflow checks, array bounds checks, or user-defined control structures.

In summary, the SELF-93 compiler brings SELF's performance to a level that is competitive with other object-oriented languages, without compromising SELF's pure semantics.

7.3.2 Different inputs

A possible objection to the above results is that the performance of SELF-93 was measured using the same inputs on which it was "trained" [137]. Intuitively, this should have a smaller impact for SELF than it does for a conventional system. First, the main benefit of feedback is type information; that is, the program's "type profile" is more important than its time profile. The type profile is much more likely to remain unchanged from input to input than the time

† Table A-6 in Appendix A has detailed data.

profile. (A recent study by Garrett et al. [57] confirmed this hypothesis for Cecil and C++.) Second, the system can dynamically adapt to changes in the time or type profile by reoptimizing new "hot spots" and by recompiling to incorporate new types.

Another potential objection to the performance measurements presented so far is that the benchmark runs are too short. (The execution times of the above benchmarks were kept relatively short to allow easy simulation.) To make sure that the small inputs do not distort the performance figures, we measured three of the benchmarks with large inputs. Table 7-4 shows that the speedups achieved with large inputs are very similar to the speedups with smaller inputs.†

Benchmark	execution time (seconds)		speedup	
	SELF-93 nofeedback	SELF-93	large input	small input[a]
CecilComp-2	97.2	71.5	1.36	1.41
CecilInt-2	38.5	21.9	1.76	1.88
Mango-2	18.5	11.6	1.59	1.59

Table 7-4. Performance of long-running benchmarks

a. computed from the data in Table A-3

7.4 Stability of performance

Besides resulting in high performance, type feedback has an additional advantage: compared to systems using a static analysis of the source, it provides more stable performance. After a minor source change, a system based solely on static analysis may lose a crucial piece of information, thus disabling a key optimization and crippling performance. For example, suppose a program's inner loop adds the elements of a vector:

```
aVector do: [ | :elem | sum: sum + elem ]
```

If a source change prevents the compiler from obtaining the exact type of `aVector`, the loop cannot be inlined and performance will drop significantly. With type feedback, the loop can be inlined anyway (using type information from the runtime system), and the only overhead added is an extra type test. The dynamic nature of type feedback results in more stable performance than static analysis.

7.4.1 Type analysis exhibits unstable performance

To measure the effect of reduced static type information on performance, we took nine small integer programs from the Stanford benchmark suite that was used to evaluate the SELF-91 compiler [21]. The inner loops of these microbenchmarks are typically one to five lines long and perform mostly integer arithmetic and array accesses. As originally written, many of these benchmarks provide static type information to the compiler for many important values. For example, the vector being sorted by the `bubble` benchmark is a constant slot, so that the compiler can inline all array accesses (`at:` and `at:Put:` sends). Also, having a constant array lets the compiler determine the array size and thus allows the compiler to infer that the integer additions performed on the loop index will not overflow. To model "normal" usage, we created a second version of each benchmark, eliminating some sources of static type information, usually by changing constant slots into assignable slots. All changes were minor and did not in any way change the algorithmic character of the benchmark. For example, the only modification to the `bubble` benchmark was to change the array being sorted from a constant slot to an assignable slot. Figure 7-9 shows the perfor-

† It is hard to separate the influence of input size from other input characteristics (i.e., the larger inputs may have exercised different parts of the benchmark applications).

mance of SELF-91 on these benchmarks (performance is shown relative to the performance of equivalent C when compiled with the GNU C compiler using full optimization).

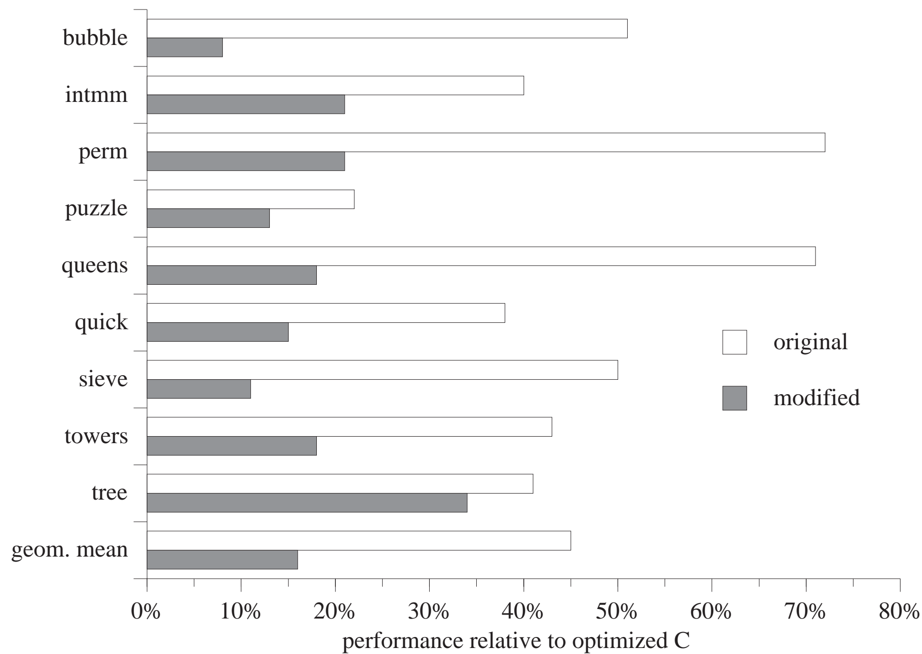


Figure 7-9. Performance of SELF-91 on Stanford benchmarks
("original" = maximum static type information, "modified" = normal type information)

Where SELF-91 has complete type information, it performs very well, reaching 45% of the speed of optimized C. SELF-91's performance drops precipitously when type information is lost; for example, `bubble` slows down by more than a factor of six. On average, the modified benchmarks slow down by a factor of 2.8 in SELF-91. Having lost important sources of static type information, the compiler is not longer able to optimize the benchmarks well.

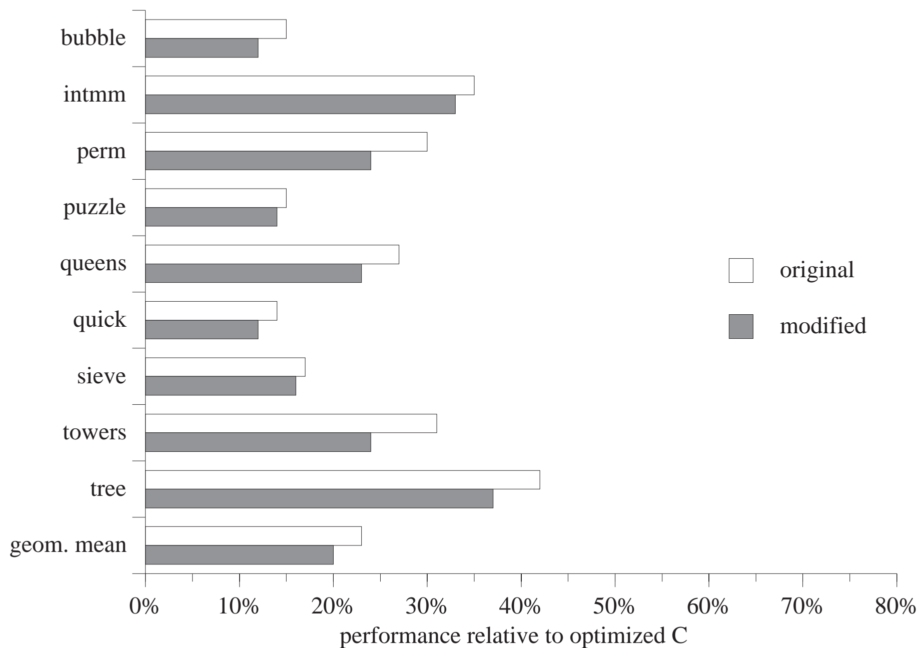


Figure 7-10. Performance of SELF-93 on Stanford benchmarks

SELF-93's performance is much more stable, dropping by only 16% on average (Figure 7-10). Even if an object's type is not known statically, type feedback can provide its type, adding only a small runtime overhead in the form of an additional type test. Type feedback has made SELF's performance much less dependent on static type information, and thus performance varies less if such information is lost.†

7.4.2 Static type prediction can fail

In systems without type feedback, a similar problem exists with static type prediction. Many systems (e.g., most Smalltalk-80 and SELF implementations) statically predict the type of certain messages. For example, the receiver of `ifTrue:` is predicted to be either the `true` or `false` object, and the receiver of + is predicted to be an integer [128, 58, 44, 130]. Static type prediction has obvious performance advantages, but it contributes to unstable performance. First, the specialized messages execute much faster than semantically equivalent code using non-specialized selectors. Second, if the prediction fails, the cost of the message send can be significantly higher than that of a normal send. For example, SELF-91 has to fall back to significantly less optimized code when a type prediction fails, as discussed in Section 6.1.4.

7.4.2.1 Non-predicted messages are much slower

The first example is taken from the Smalltalk world, where a Smalltalk programmer noted on the newsgroup `comp.lang.smalltalk` that the following two code sequences had vastly different performance even though they perform the same amount of work (in source-level terms):

```
"nil1:" x isNil ifTrue: [ 1 ] ifFalse: [ 2 ]
"nil2:" x ifNil: [ 1 ] ifNotNil: [ 2 ]
```

The programmer was puzzled by the fact that the first expression ran almost *ten times* faster on his Smalltalk system than the second expression, even though it involves two sends (`isNil` and `ifTrue:ifFalse:`) instead of just one. The reason for this performance variation is that Smalltalk optimizes the `ifTrue:ifFalse:` message send by type-predicting the receiver and then hard-coding the control structure (in other words, the source code of the `ifTrue:ifFalse:` message is ignored). As a result, the Smalltalk system can not only avoid the actual send but also the creation of actual block objects for the two argument blocks. Since `ifNil:ifNotNil:` does not get such special treatment, its execution involves both the actual message send and the creation of the two argument blocks, and therefore runs much more slowly. Smalltalk's special-casing of some messages creates unstable performance.

Previous SELF implementations had similar problems since they type-predicted `ifTrue:` but not `ifNil:`. We translated the example into SELF and ran each expression 100,000 times (Table 7-5).

	SELF-91	SELF-93-nofeedback	SELF-93
nil1 ("x isNil ifTrue: ..."; type prediction works)	102 ms	167 ms	82 ms
nil2 ("x ifNil: ..."; no static type prediction)	493 ms	594 ms	75 ms
ratio nil2 / nil1	4.8	3.6	0.91

Table 7-5. Performance variation related to static type prediction

On `nil1`, SELF-93 is slightly faster than SELF-91 because type feedback enables it to inline the `isNil` send; SELF-93-nofeedback is considerably slower because of the relatively low-quality compiler back end. However, all systems do relatively well since they correctly predict `ifTrue:`. On `nil2`, the compilers using static type prediction suffer from exactly the same problem as the Smalltalk system and produce dramatically slower code; for example, SELF-91 is almost five times slower. In contrast, SELF-93 generates essentially the same code as for `nil1` since type feedback

† The astute reader will notice that SELF-91 considerably outperforms SELF-93 on the original benchmark suite. Section 7.5.4 will examine this situation in more detail and reveal that SELF-91's performance advantage is due to its better back end.

predicts the receiver of the `ifNil:IfNotNil:` message. Type feedback provides more stable performance since it can predict any message send, not just a few special cases.

7.4.2.2 Mispredictions can be costly

A second example shows the effect of a mispredicted message send in SELF. This example adds two points using the expressions `p1 + p2` and `p1 add: p2`. Both expressions compute exactly the same result.† In the first case, static type prediction will predict `p1` to be an integer since it is the receiver of a `+` message; in the second case, no such misprediction occurs because `add:` is not a type-predicted message name. Table 7-5 shows the results of running each expression 100,000 times.

	SELF-91	SELF-93-nofeedback	SELF-93
p1 + p2 (type prediction fails)	914 ms	609 ms	332 ms
p1 add: p2 (no prediction)	490 ms	550 ms	327 ms
ratio	1.9	1.1	1.02

Table 7-6. Performance variations caused by failing static type prediction

Using neither feedback nor recompilation, SELF-91 shows the largest variation, a factor of 1.9. SELF-93-nofeedback predicts an integer receiver (like SELF-91) but can back out of the misprediction by recompiling the method after encountering an uncommon trap (see Section 6.1.4). Recompiling after encountering uncommon cases reduces the misprediction penalty to 10%. Thus, SELF-93-nofeedback runs faster than SELF-91 in the case of a misprediction; however, without type feedback, the compiler cannot inline the `add:` send and thus the code is still significantly slower than SELF-93. Finally, SELF-93 produces the fastest code in both cases and shows virtually no performance variation. SELF-93 never predicts an integer receiver since the type feedback information from the unoptimized code indicates that the correct receiver type is Point (recall that the unoptimizing compiler performs a send even for `+`). Once more, type feedback and recompilation help to reduce performance variations.

Type feedback helps providing more stable performance because it allows the system to dynamically adapt to the program's actual type usage. Systems without type feedback show higher performance variations because they cannot cope with missing static type information. Such systems also have to rely on static heuristics to speed up common sends but cannot adapt if a new frequently-used message appears or if the heuristics fail.

7.4.2.3 Performance variations in high-level languages

Unstable or unexpected performance has often been quoted as an argument against high-level languages with "expensive" operations whose cost cannot be accurately predicted by the programmer. For example, Hoare [68] argues as follows:

> "The only language which has been optimized with general success is Fortran, which was specifically designed for that purpose. But even in Fortran, optimization has grave disadvantages: [...] A small change in an optimized program may switch off optimization with an unpredictable and unacceptable loss of efficiency. [...]
> The solution to these problems is to produce a programming language for which a simple straightforward "non-pessimizing" compiler will produce straightforward object programs of acceptable compactness and efficiency."

We believe that this argument has lost much of its strength in recent years. With the advent of high-speed RISC processors and cache memories, RAM is no longer "random access" and thus basic operations such as accessing an array element can have widely varying costs even in a relatively low-level language like Fortran. For example, the

† `add:` is not part of the standard SELF system; usually, points are added using `+`.

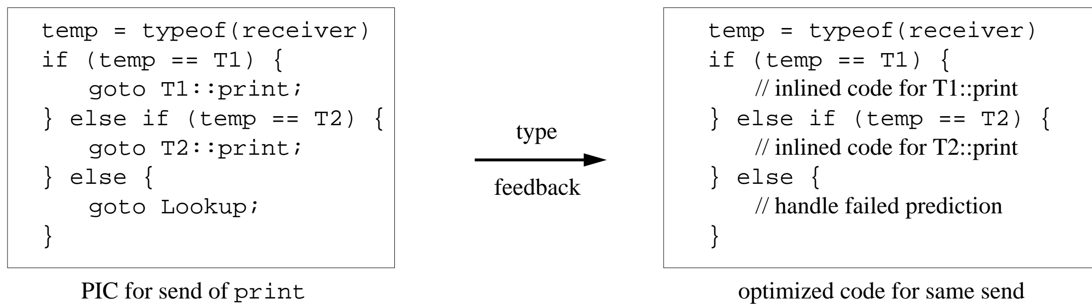
matrix300 benchmark (written in Fortran) that was part of the SPEC89 benchmark suite can be sped up by an order of magnitude with a series of loop transformations that improve cache performance. To the programmer lacking knowledge of the machine architecture, the transformed program appears more complicated and less efficient, rather than radically faster. It appears that every programming language having memory-accessing operations can suffer from significant performance variations that cannot be explained at the source level.

7.5 Detailed performance analysis

We will now analyze some sources of overhead in more detail. First, we look at the overhead of type tests. How much time do optimized programs spend testing receiver types, and how does type feedback influence this overhead?

7.5.1 Type feedback eliminates type tests

Type feedback inlines sends using type test sequences that choose one of several inlined code sequences according to the receiver's type. That is, type feedback replaces a dispatch type test (in a PIC or method prologue) with an in-line type test. For example, assume that `print` is sent to objects of type T1 and T2:



PIC for send of `print` | optimized code for same send

At first sight, it would seem that type feedback replaces each dispatch test with exactly one inlined test for every send that is inlined. Thus, one would expect programs to execute the same number of type tests before and after applying type feedback optimizations. However, Figure 7-11 shows that this is not the case: programs compiled with type feed-

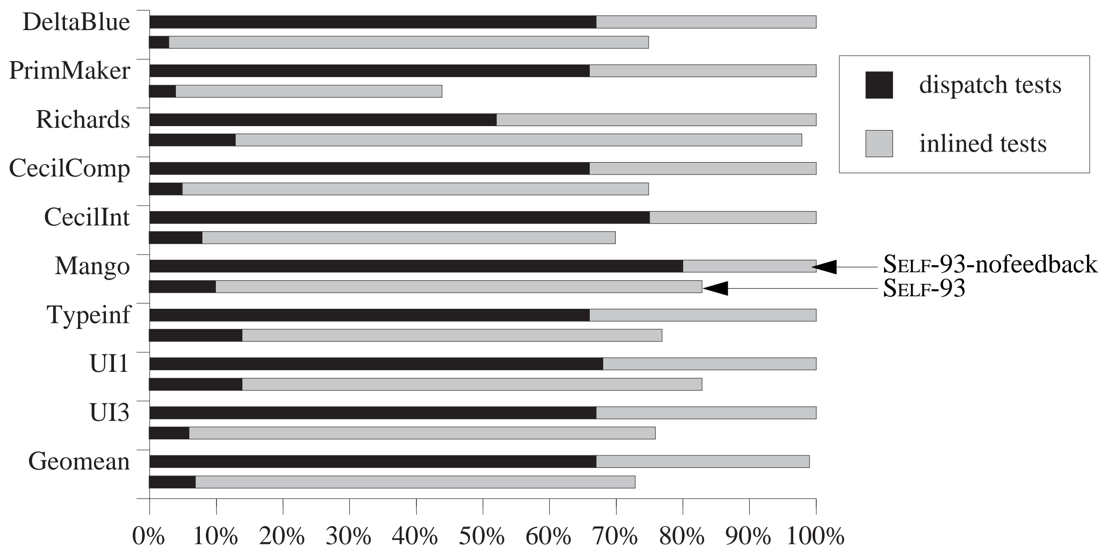


Figure 7-11. Number of type tests executed in SELF-93-nofeedback and SELF-93

back actually execute fewer type tests. For each benchmark, the figure shows the number of dispatch and inlined type tests relative to SELF-93-nofeedback (the upper bar is for SELF-93-nofeedback, the lower for SELF-93).†

In all benchmarks, SELF-93 performs fewer type tests; on average, only 73% of the original tests. Apparently, the simple story about the effect of type feedback is incomplete. However, type feedback does replace dispatch tests with inlined tests: whereas dispatch tests outnumber inlined tests by 2.1 : 1 in SELF-93-nofeedback, the reverse is true in SELF-93 where the ratio is 1 : 10.

The reason for this somewhat surprising result is that inlining a call (using one type test) may result in additional type information that allows other sends to be inlined without any type test. For example, if the type of an argument of an inlined call is known, all sends to that argument can be inlined without a test. (This is a frequent case since the arguments of control structures such as iterators are usually block literals, i.e., constants.) In the non-inlined case, these sends require a type test since the type of the argument is not known. Thus, inlining the send saves type tests.

Similarly, in some situations the compiler may be able to use the outcome of a type test for another send to the same value even though it does not perform general type analysis. Specifically, assume that x is a non-assignable slot (e.g., a parameter‡) that is predicted by type feedback to be of type T, and that the compiler decides to make the "not a T" case unlikely. Then, the statement sequence "`x foo. x bar`" (send `foo` to `x`, then send `bar` to `x`) will be translated into the following code:

```
"x is T or unknown"
if (x is T) {
    "inlined code for foo"
} else {
    unknown_trap();                         "will never return"
}
"now, x is T"
"inlined code for bar—no type test needed"
```

Since the first type test verifies `x`'s type and leads to a trap if it is not T, `x` is known to be of type T in the remainder of the method, and subsequent sends to it can be inlined without any type tests.

We measured the extent of this effect by dividing the number of sends that were inlined by the extra inlined type tests executed (as always, we use dynamic counts). A ratio of 1.0 indicates no "bonus": every newly inlined send requires a type test. Values greater than 1.0 indicate that some sends could be inlined for free. On average, SELF-93 removed 1.88 calls for every new inlined type test it executed (Figure 7-12), so the "bonus" is real even though the compiler does not perform extensive dataflow or type analysis. Programs optimized with type feedback execute fewer type tests.

7.5.2 Type feedback reduces the cost of type tests

Type feedback also reduces the number of types actually tested per type test (its *path length*). For example, a type test that succeeds after the first test (i.e., the object's type was the first type tested) has a path length of one, even if the type test sequence contains several cases. Figure 7-13 shows the path lengths of dispatch and inlined tests for SELF-93-nofeedback and SELF-93.†† The left box within each category represents SELF-93-nofeedback, and the right box

† By "test" we mean a code sequence testing a value against one or more types; that is, a test counts only once even if it tests the value against several types. Inlined tests include type feedback tests, type prediction tests, and tests in inlined primitives (e.g., to verify that the arguments to the integer addition primitive are indeed integers). Thus, the inlined tests in SELF-93-nofeedback are either prediction tests (testing for true/false or integer) or primitive tests. See Tables A-8 to A-9 in Appendix A for detailed data.

The data does not include type tests performed in the virtual machine (e.g., in non-inlined primitives) since these tests are beyond the control of the SELF compiler. However, no benchmark spent much time in type tests in the virtual machine.

‡ SELF methods cannot modify their parameters.

†† The ends of a box correspond to the 75% and 25% quartiles, the horizontal line shows the median. The vertical lines at each end of the box extend to the minimum and maximum value.

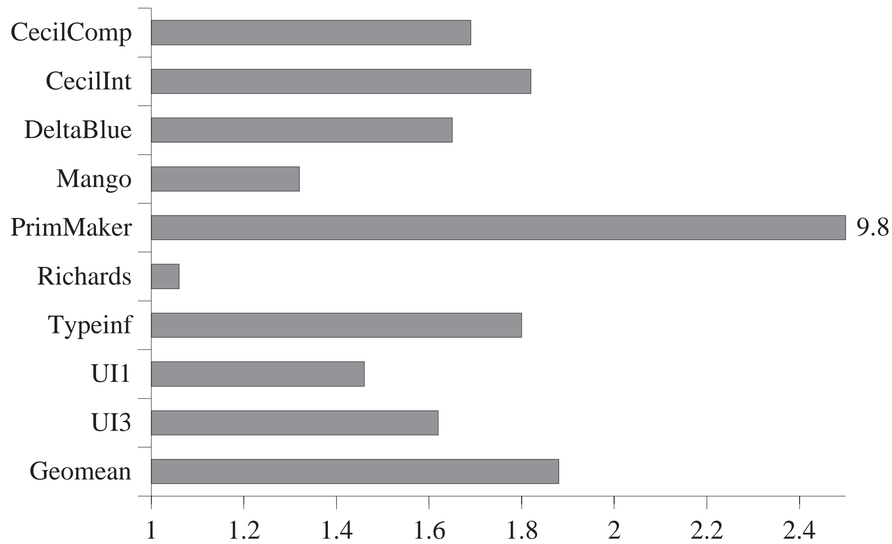


Figure 7-12. Sends inlined per additional inline type test

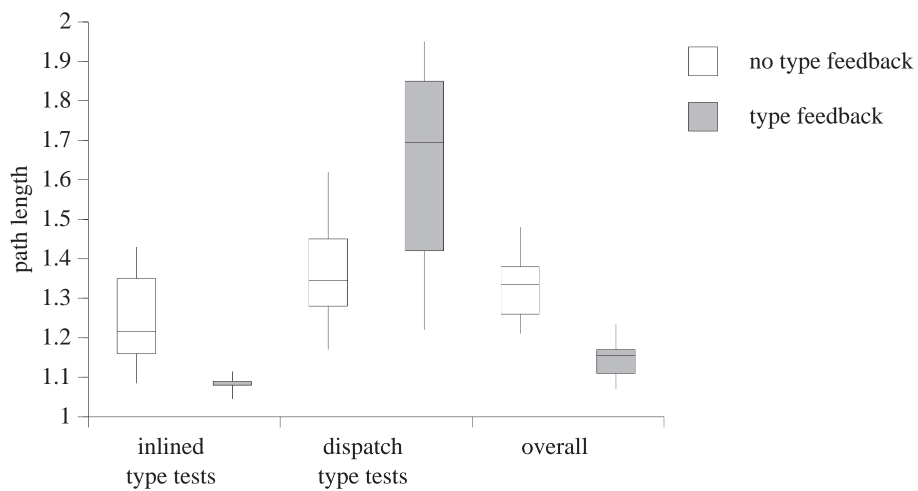


Figure 7-13. Path length of type tests

represents SELF-93. Figure 7-13 highlights two effects. First, inlined type test sequences have a shorter path length than type test sequences used during method dispatch. For example, inlined type tests execute only 1.08 comparisons (geometric mean) per type test sequence in SELF-93, but dispatch tests execute 1.62 comparisons.

Second, type feedback reduces the path length of inlined type tests (from 1.24 to 1.08) but increases the path length of dispatch type tests (from 1.36 to 1.62). The main reason for the reduced path length in inlined tests is that inlining reduces the degree of polymorphism in the code by creating separate copies of the callee for each caller. Thus, the types used in a particular (inlined) callee are just the types used by its caller and not the union of several callers. The remarkably low path length of inlined type tests in SELF-93 (and the equally remarkable low variance) show that the vast majority of inlined sends requiring a type test need to perform only a single comparison to find their target.

The increased path length of dispatch tests is probably a result of not inlining highly polymorphic or megamorphic calls. Since many monomorphic calls are inlined, the remaining calls have a higher degree of polymorphism and thus longer path lengths.

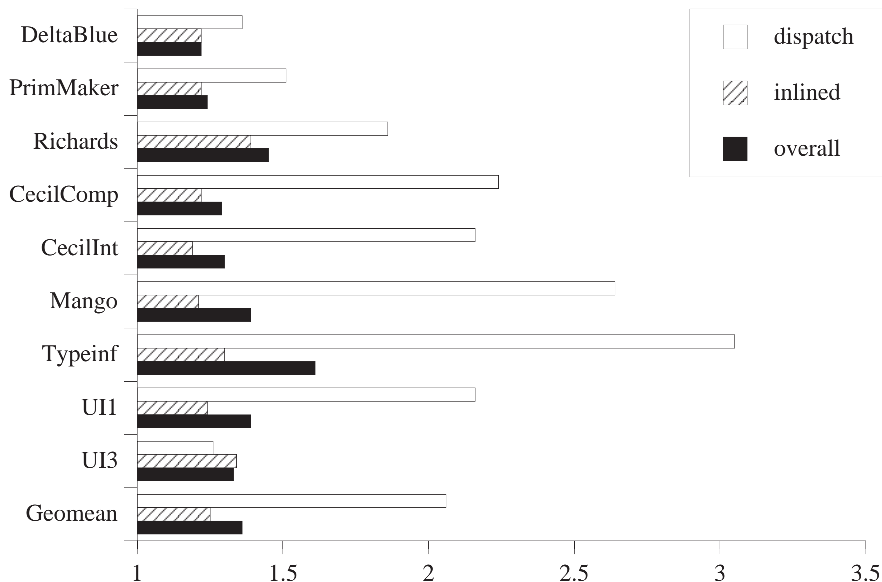


Figure 7-14. Arity of type tests (dynamic averages)

Figure 7-14 shows the average *arity* of type tests, i.e., their degree of polymorphism (number of cases in a type test or dispatch sequence). A type test's arity is always greater than or equal to its path length for every possible execution. For example, a test of arity 2 (e.g., testing for true and false) can have a path length between 1 and 2 if it also contains the unknown case, and a path length of 1 if it doesn't contain the unknown case.† The data show dynamic averages.

As was the case with path length, the arity of dispatched (non-inlined) sends is higher than that of inlined sends. Since inlined sends dominate in frequency, the overall average is closer to the arity of inlined sends. On average, a send's arity is only slightly higher than its path length. For inlined tests, the average arity is 1.25 vs. a path length of 1.08. That is, the path lengths observed in Figure 7-13 are so small because most type tests contain only one or two types. (An arity much higher than the path length would indicate that the first case being tested is much more likely than the others. SELF-93 does not order the cases of a type test according to frequency because it does not have precise edge counts; the low arity shows that this optimization probably would not make much of a difference.)

In conclusion, type feedback reduces the cost of type tests because it decreases their path length, i.e., the number of type tests per send. Both path length and arity of inlined sends are close to 1, indicating that many inlined sends were specialized for just one type.

7.5.3 Type feedback reduces the time spent in type tests

Type feedback reduces the number of type tests and the work performed per test, as we have seen in the previous two sections. Together, these two factors help to reduce the overhead of type tests in SELF-93. Figure 7-15 shows the execution time consumed by type tests in SELF-93-nofeedback (upper bars) and SELF-93 (lower bars). In all benchmarks, SELF-93 spends less time in type tests; on average, 1.62 times less (geometric mean). This reduction is not a result of different code sequences for dispatched and inlined tests (i.e., faster inlined tests)—both sequences are virtually identical.

Figure 7-16 shows the percentage of execution time spent in type tests for SELF-93, split up into dispatch and inlined tests. On average, SELF-93 spends only 15% of its time in type tests‡ when executing our benchmark programs; also, the large benchmarks tend to spend less time in type testing than the smaller ones. In comparison with other systems,

† One test suffices for two types if there is no "unknown" case; a frequent case occurs when testing the result of a primitive known to return either true or false.

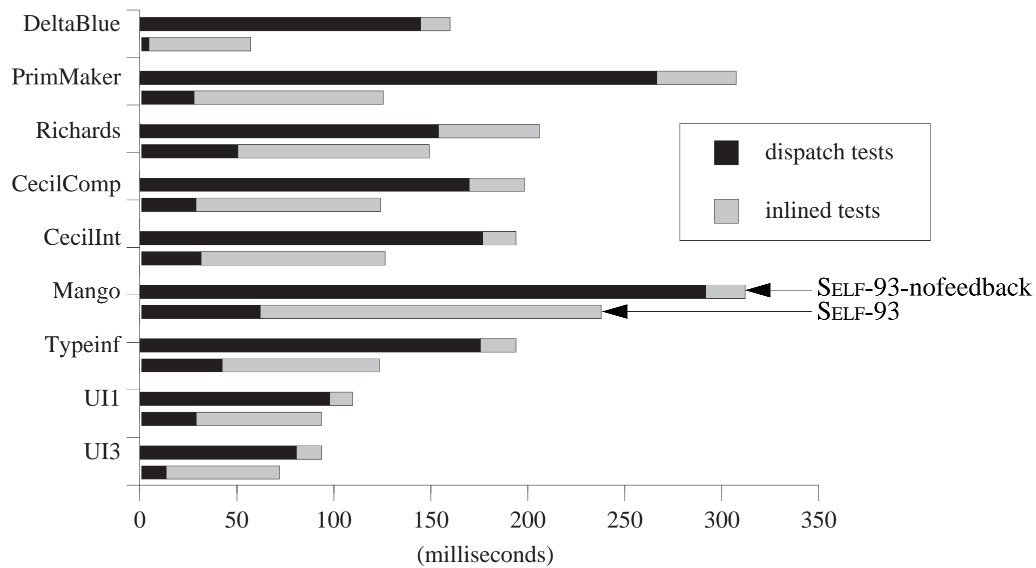


Figure 7-15. Execution time consumed by type tests

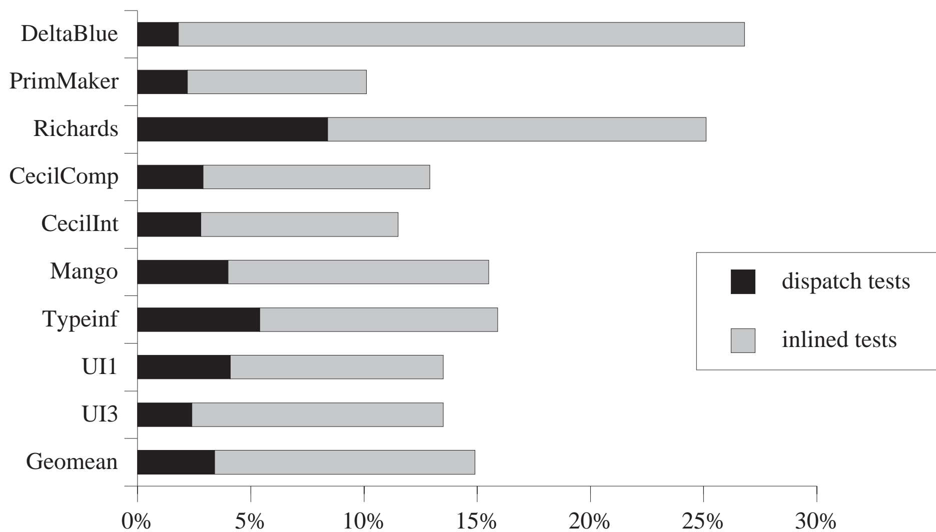


Figure 7-16. Percentage of execution time spent in type tests for SELF-93

the type testing overhead in SELF-93 appears modest, especially when considering the absolute speed of systems (since the relative overhead increases as other parts of the system become faster). In comparison, SOAR, a Smalltalk implementation on a RISC processor with special hardware support, spent 23% of its time in method dispatch and would have spent another 24% for integer tag checking had it not included special hardware [132].[†] In Lisp systems, tag checking is the closest equivalent to type tests since Lisp does not have dynamic dispatch. Steenkiste reported

‡ Recall that the category "inlined tests" includes not only type feedback tests but also tests required to verify the arguments of primitive operations, i.e., that the index of an array indexing operation is an integer.

† The SPARC architecture contains most of this support, so SOAR would not incur additional overhead on a SPARCstation. However, virtually no other RISC architecture includes such support; also, as we discuss in Chapter 8, the current SELF system does not profit much from these instructions. Thus, on another RISC architecture, it is likely that a SOAR-like system would be slowed down more than SELF.

11% to 24% of execution time being spent in tag handling for Lisp on the MIPS-X [121], and Taylor reported that 12% to 35% of all instructions involved tag checks on the SPUR machine (which was a tagged architecture) [126].

In conclusion, type feedback reduces the number of type tests, the amount of work (path length) per type test, and the total execution time spend in type tests. On average, programs spend less than 15% of their time in type tests, i.e., message dispatch.

7.5.4 Type analysis vs. type feedback

As we have seen in Section 7.3, a system using only type feedback (i.e., SELF-93) outperforms a system using only type analysis (SELF-91 on the large object-oriented benchmarks we measured). How valuable, then, is type analysis? Comparing SELF-91 to SELF-93-nofeedback, we saw that SELF-91 is only marginally faster on the benchmarks and inlines approximately the same number of sends. Although it is hard to separate the effects of the individual factors contributing to this performance difference (type analysis, static type prediction, code generation), this result is a strong indication that type analysis in SELF-91 does not pay off for these programs.

But how valuable would type analysis be for SELF-93, i.e., combined with type feedback? Since SELF-93 spends about 10% of its time in type tests (not including type tests required for method dispatch) for the large, object-oriented applications we measured, it is likely that type analysis would not improve performance by much more than that.† However, type analysis may be more valuable in one particular class of programs, those with very tight loops. If a program's inner loop consists of very few instructions, any extra type test brings a significant performance hit with it. For example, the inner loop of `bubblesort` consists of four instructions in C. A single test to verify the type of the array adds another four instructions to the loop, thus halving performance. In addition, the impact of code generation weaknesses (unfilled delay slots, redundant register moves) tends to be amplified in very small loops.

Figure 7-17 compares SELF-93 and SELF-91 on the original Stanford integer benchmarks, a set of small integer benchmarks [21], most of which are only a few lines long. Most programs come in two versions, plain and "oo" (object-oriented). The oo versions have been rewritten to make the main data structure the receiver. For example, `bubble` takes the array to be sorted as an argument (the receiver being the bubble benchmark object), but in `bubble_oo` the array is the receiver of the sort message. This difference can have a large performance impact (at least for SELF-91) because the receiver type is known due to customization. SELF-91 performs much better on these small benchmarks than it did on the applications in our main benchmark suite, outperforming SELF-93 by a factor of 1.9. (Recall that SELF-93 was 1.5 times faster on the large benchmarks.)

There are two main reasons for SELF-91's good performance on the original Stanford benchmarks. First, in most of the benchmarks the compiler can statically know *all* receiver types, either because they are constants or through static type prediction. (As discussed in Section 7.4.1, SELF-91's performance drops precipitously if this static type information is removed.) Therefore, the SELF-91 compiler can inline all sends and isn't at a disadvantage relative to SELF-93. Second, SELF-91 performs a variety of optimizations that work very well on these benchmarks, such as loop splitting, range analysis (to eliminate array index bounds checks and arithmetic overflow tests), and common subexpression elimination. Furthermore, its back end is considerable more sophisticated. Since most of the benchmarks are very small, the weaknesses in SELF-93's code generator (such as extra register moves or repeated loads) have a large performance impact.

Figure 7-18 shows the number of cycles executed in the inner loop of `sieve`‡ which consumes 32 cycles in SELF-93 vs. 8 cycles in SELF-91. Back end differences are responsible for most of the performance difference. Unnecessary register moves and unfilled delay slots account for 14 of the 32 cycles, and missing array optimizations consume another 6 cycles (SELF-91 uses a derived pointer to step through the array, whereas SELF-93 indexes into the array in

† Eliminating type tests could conceivably save additional time through secondary effects such as better cache behavior or larger basic blocks.

‡ In C, the loop is `while (k <= size) { flags[k] = 0; k += prime; }`.

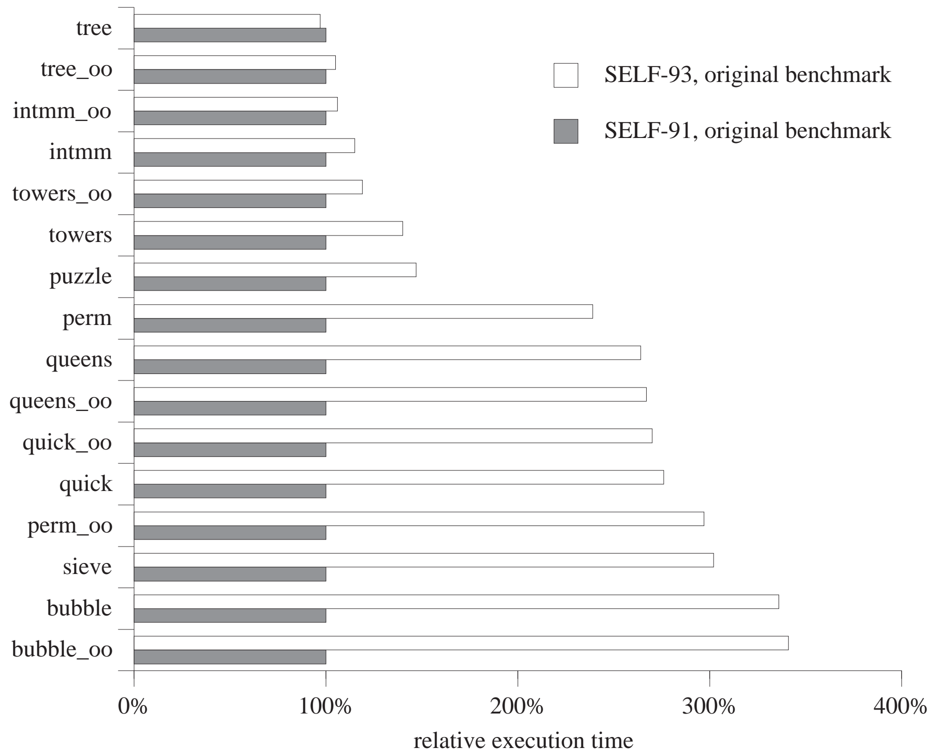


Figure 7-17. Performance on the Stanford integer benchmarks

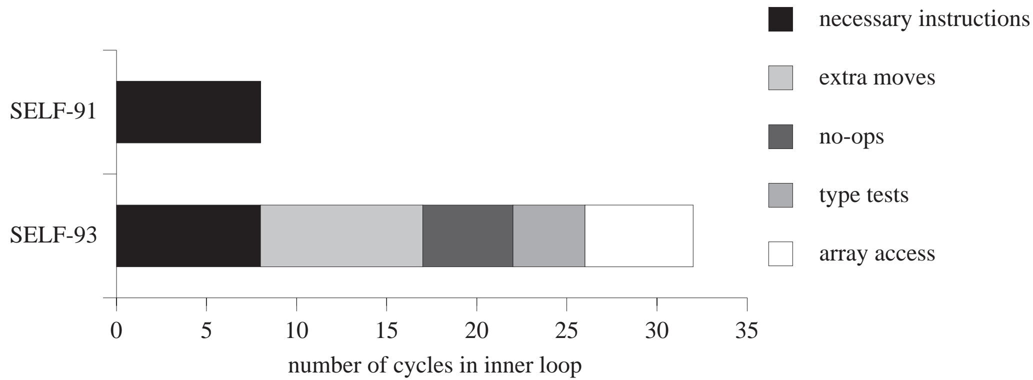


Figure 7-18. Analysis of inner loop of `sieve` benchmark

each iteration). Type analysis would eliminate only 4 instructions, two integer type tests for `k`. That is, if SELF-93 had SELF-91's back end optimizer but did not perform type analysis, its performance for `sieve` would be much more competitive.

Figure 7-19 shows that adding type analysis to SELF-93 is unlikely to significantly speed up any of the other benchmarks since the average type testing overhead is below 10% of total execution time. For example, `bubble` is 6 times slower in SELF-93, yet spends only 10% in type tests.

In conclusion, the relatively poor performance of SELF-93 on these benchmarks results from our initial decision to keep the compiler small and fast. With a better back end using global dataflow analysis and standard optimizations, SELF-93 could probably reach a performance level competitive with SELF-91 even on these benchmarks. Standard dataflow techniques could also reduce the type testing overhead: for each branch *i* of a type test of value *v*, define the

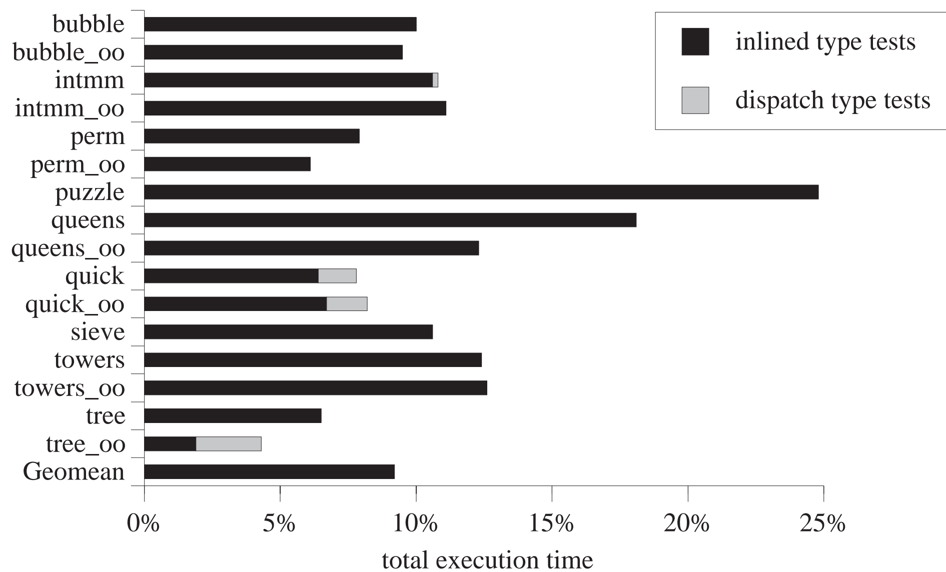


Figure 7-19. SELF-93 type testing overhead in the Stanford benchmarks

auxiliary value $type_v$ to represent the result of the type test, add the assignment $type_v = T_i$ (where T_i is the type being tested against) at the beginning of branch i, and add $kill(v)$ to $kill(type_v)$. Then, type tests we be can optimized using standard value propagation: if a single definition of $type_v$ reaches a type test of v, the type test can be eliminated. Using standard dataflow techniques rather than full-fledged type analysis sacrifices some precision since value propagation "loses" the value of $type_v$ if more than one definition reaches some point, whereas type analysis keeps track of the exact union of the values. However, recall that the vast majority of type tests involves a single type and makes the "otherwise" branch uncommon (Figure 7-14 on page 82). Thus, at most one definition of $type_v$ will reach other type tests, and the loss of information relative to type analysis will not occur. Therefore, we believe that adding standard dataflow analysis to the compiler would eliminate redundant type tests almost as well as a full type analysis system.

7.5.5 Reoptimization effectiveness

All of the performance numbers for SELF-93 presented so far included a certain amount of unoptimized code since the recompilation system does not optimize every single method in the program. How much faster would programs run if the system optimized everything, i.e., how much time is spent in unoptimized code? Table 7-7 answers these questions. The time spent in optimized and unoptimized code was measured using the Unix kernel profiling facility; we repeated each benchmark 100 times (with recompilation turned off) to get accurate data.

For most of the programs, unoptimized code represents less than 5% of total execution time. The two programs spending the most time in unoptimized code, `PrimMaker` and `UI1`, both use *dynamic inheritance* (DI) which reduces the effectiveness of all optimizing compilers used in our comparison. With dynamic inheritance, an object can change its inheritance structure on the fly, thereby changing the set of methods it inherits. Current SELF compilers cannot inline any sends that may be affected by dynamic inheritance because lookup results aren't compile-time constants with DI. DI also poses problems to the recompilation system; as a result, it will never recompile a method affected by DI.†

† We do not discuss the exact reason because it would require a detailed explanation of the way dynamic inheritance is currently implemented. The problem could be fixed, but we chose to keep our implementation simple and to wait for a reorganization of the implementation of DI (which would most likely make DI much simpler to handle for the recompilation system).

	% time in unoptimized code
CecilComp	6.1%
CecilInt	4.6%
DeltaBlue	0.2%
Mango	2.0%
PrimMaker	8.6%
Richards	0.0%
Typeinf	3.1%
UI1	14%
UI3	4.7%
median	3.3%

Table 7-7. Time taken up by unoptimized code

The two benchmarks with almost no time spent in unoptimized code, `Richards` and `DeltaBlue`, are the two smallest benchmarks and spent most of their time in very few compiled methods. For example, `Richards` spends 95% of its time in only 7 compiled methods. (For the larger programs, the top 20 compiled methods combined usually represent 40-60% of execution time.)

In conclusion, the speed of unoptimized code is largely irrelevant for the benchmarks measured, since they do not spend much time in unoptimized code. For example, it might be possible to replace the non-optimizing compiler with an interpreter without losing too much performance.

7.5.6 Size of compiled code

Despite the additional inlining, programs generated SELF-93's code are not much larger than those generated by the other compilers. Figure 7-20 shows the size of compiled code for SELF-93 (upper bar in each benchmark) and SELF-91 (lower bar) relative to SELF-93-nofeedback. On average, SELF-91's code is smallest, 9% smaller than SELF-93-nofeedback's code. This difference is probably a result of SELF-91's better back end—SELF-93-nofeedback generates more register moves and redundant loads and stores. SELF-93's code is biggest on average, 14% bigger than SELF-93-nofeedback and 25% bigger than SELF-91.

The additional inlining enabled by type feedback can sometimes lead to larger code. For example, the optimized SELF-93 code for `Typeinf` is more than 40% larger than that of SELF-93-nofeedback. However, besides `Typeinf` only `CecilComp` shows a substantial increase in optimized code, and additional inlining can also reduce code size. For example, `Richards` is substantially smaller when compiled by SELF-93.

Second, unoptimized code is much less dense than optimized code (see Section 4.3.5). Thus, the unoptimized parts of a program take up more space than they would if everything was optimized. The compiler configuration used for our measurements recompiles aggressively, so that only a relatively small portion of the program remains unoptimized. With less aggressive recompilation, more code would remain unoptimized, and overall code size would most likely grow.

7.6 Summary

Type feedback is very successful in optimizing SELF programs: in the SELF-93 compiler, it improves performance by a factor of 1.7 for a suite of large SELF applications. Compared to SELF-91, the previous SELF compiler, SELF-93 is 1.52 times faster even though SELF-91 is significantly more complicated and performs more back end optimizations. Type feedback also reduces the call frequency of the programs by a factor of 3.6. The counter-based recompilation

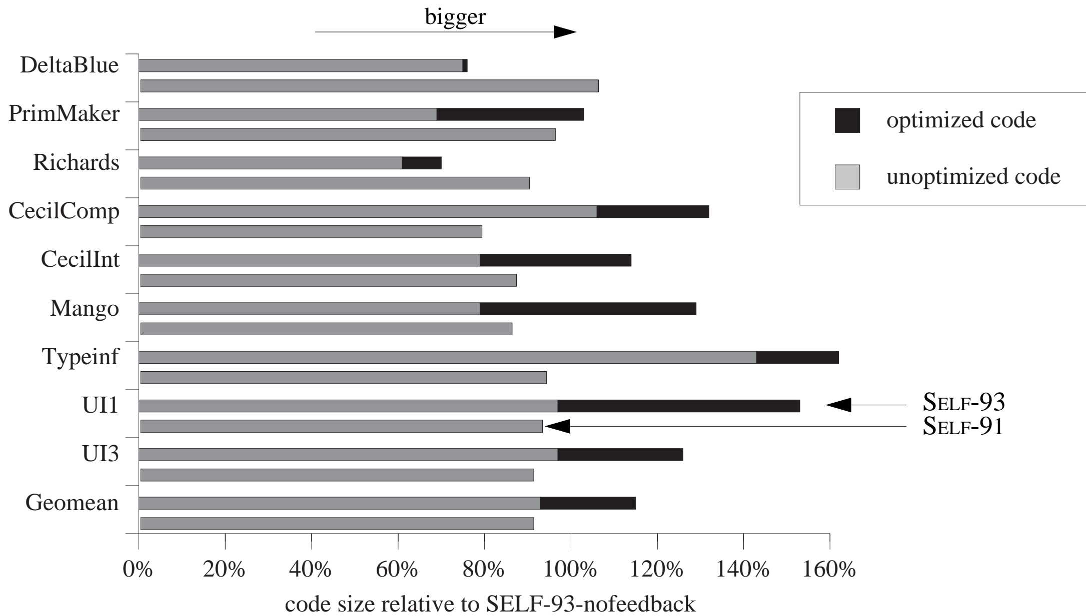


Figure 7-20. Code size of SELF-93 and SELF-91

system is effective in finding the time-critical parts of our applications; most applications spend less that 5% of their time in unoptimized code.

Although every send in SELF is dynamically-dispatched, the dispatch overhead of optimized programs is modest. SELF-93 implements message dispatch with type test sequences, either in PICs (Chapter 3) or in inlined code. Even though the SELF-93 compiler does not perform extensive optimizations aimed at eliminating type tests, the execution time overhead of type tests is small, about 15% for our benchmark suite (including tests used for method dispatch). Type feedback actually reduces the type testing overhead compared to a system not using feedback. Inlined type tests have a very short path length: on average, a type test sequence performs only 1.08 comparisons before branching to the target code.

Type feedback can help reduce the performance variability experienced in systems using static type prediction or type analysis. In those systems, a small change can have a large performance impact if the change removes an important piece of type information, or if it causes type prediction to fail. Since our implementation of type feedback obtains its information from the running program itself, it is less dependent on static information, and thus a small change usually does not fundamentally change the set of optimizations the compiler can perform.

Type feedback is a simple technique that significantly speeds up object-oriented programs. We hope that the results presented here will lead implementors of other object-oriented languages (or other systems with similar implementation characteristics) to consider type feedback as an attractive implementation technique.[†]

[†] If you should use type feedback in your system, I would like to hear from you. Please drop me a line at urs@cs.stanford.edu.

8. Hardware's impact on performance

How much could dedicated hardware improve the performance of object-oriented programs? Many previous studies have indicated that various hardware features could improve performance, and some implementations of object-oriented systems have relied heavily on hardware support. For example, Ungar reported that the SOAR system would have been 46% slower without register windows and 26% slower without instructions for tagged arithmetic [132]. Williams and Wolczko argue that software-controlled caching improved the performance and locality of object-oriented systems [142]. Xerox Dorado Smalltalk, for a long time the fastest Smalltalk implementation available, contained microcode support for large portions of the Smalltalk virtual machine [43]. However, none of these systems used an optimizing compiler comparable to SELF-93, and so the results of previous studies may not be valid for the current SELF implementation.

In order to evaluate which architectural or implementation features (if any) would benefit its execution, we have analyzed the instruction usage of programs compiled with SELF-93. First, we compare SELF's instruction mix against the SPECInt89 benchmark suite which consists of programs written in C. Then, we evaluate two features present in the SPARC architecture (register windows and tagged arithmetic) that are commonly believed to improve the performance of object-oriented programs. Finally, we examine the cache behavior of the benchmark programs.

The results indicate that programs compiled with SELF-93 differ remarkably little from optimized C programs, less so than a suite of C++ programs. Unlike previous studies, we could not find any "object-oriented" hardware feature that would improve performance by more than 10%.

8.1 Instruction usage

When designing an architecture, one of the first tasks is to identify frequent operations and then to optimize them. Many previous studies have found the execution characteristics of object-oriented languages to be very different from C and have argued the need for object-oriented architectures. For example, Smalltalk studies [86, 132] have shown calls to be much more frequent than in other languages. Even for a hybrid language like C++ (which has C at its core and thus shouldn't be too different), significant differences were found. Table 8-1 shows data from a study by Calder et al. [16] that measured several large C++ applications and compared them to the SPECint92 suite and other C programs. The study used the GNU C and C++ compilers on the MIPS architecture. Even though C++ is similar in spirit to C, the differences in execution behavior are pronounced: for example, C++ executes almost 7 times more calls and executes less than half as many conditional branches.

	C++	SPECint92	ratio C++ / SPEC
basic block size	8.0	4.9	1.6
call/return frequency	4.6%	0.7%	6.7
instructions per conditional branch	15.9	6.4	2.5

Table 8-1. Differences between C++ and C (from [16])

Based on these numbers, one would expect a pure object-oriented language like SELF to be much further away from C, because the language is radically different from C. (Recall that, for example, even integer addition or `if` statements involve message sends.) However, execution behavior is a function not only of the source language characteristics but also of compiler technology, and the latter can make a big difference. Figure 8-1 shows the dynamic instruction usage of the four SPECint89 integer benchmarks (written in C) and the nine SELF programs we measured. The data is summarized using box plots† (the appendix contains detailed data). The leftmost box in each category represents SELF-only, i.e., the execution of compiled SELF code, excluding any time spent in the runtime system (which is written in C++). The middle box in each category (filled gray) represents complete SELF programs (i.e., all

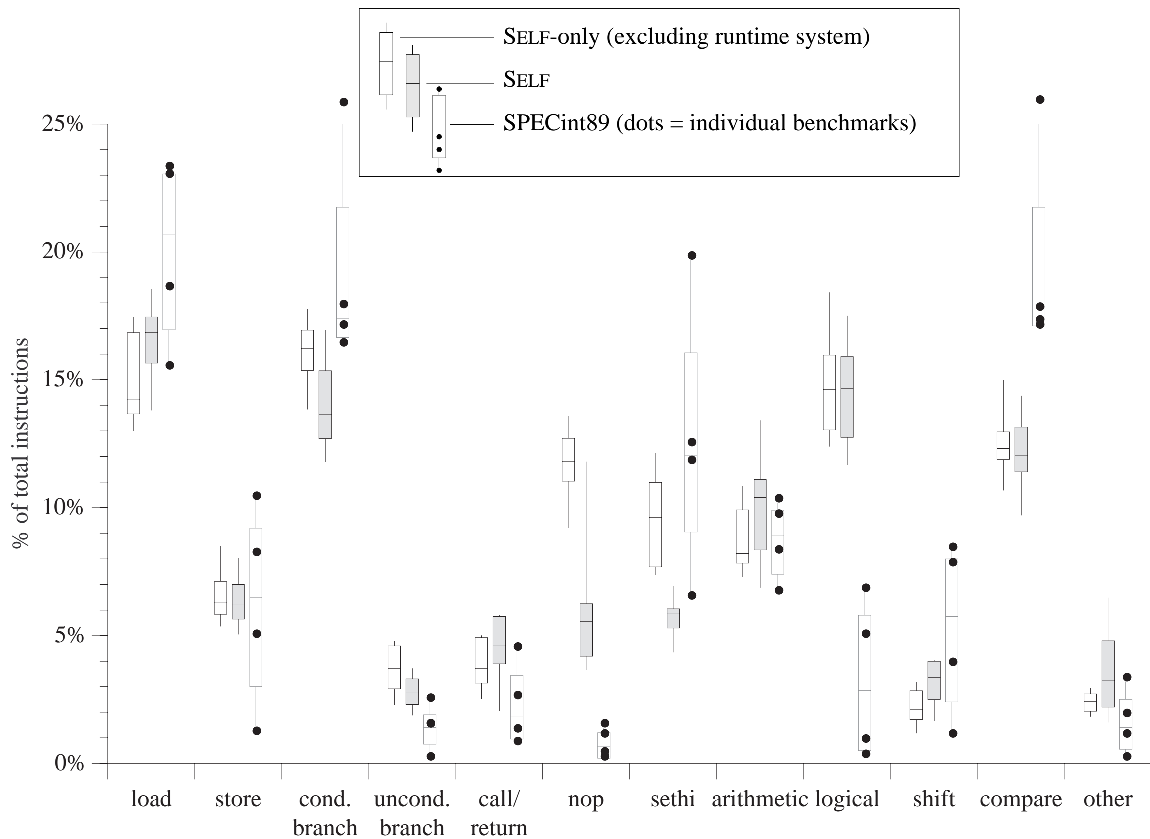


Figure 8-1. Dynamic instruction usage of SPECint89 vs. SELF-93 benchmarks

user-mode instructions). We have measured both since SELF programs often call routines in the runtime system, for example, to allocate objects or call functions defined in C libraries. Some programs spend one-third of their time in such routines, and we wanted to make sure that our data was not biased by the execution behavior of non-SELF code. On the other hand, showing SELF-only could be misleading as well, since this data may not be representative of the instructions the processor actually executes.

Since there are only four benchmarks in the SPECint89 suite, the individual data points for C are shown directly, and the box plots (dotted) are given for reference only. The SPECint89 data are taken from Cmelik et al [33].

Figure 8-1 reveals several interesting points:

1. Overall, there are few differences between the SPEC benchmarks and SELF. Often, the differences between the individual SPEC benchmarks are bigger than the difference between SELF and C (in the graph, the SPEC boxes are usually larger than the corresponding SELF boxes).

2. On average, the SELF programs execute more no-ops, more unconditional branches, and more logical instructions. However, most of these differences can be explained by the simple back end of the current SELF compiler, as explained in more detail below. Thus, they are an artifact of the current compiler back end and are not linked to the object-oriented nature of SELF.

† A box plot summarizes a distribution by showing the median (horizontal line in the box), 25% / 75% percentiles (end of the box), and 5% / 95% percentiles (end of vertical lines).

3. SELF’s basic blocks are of similar size than that of the SPEC programs, 4.5 instructions vs. 4.4 for the SPEC benchmarks (geometric means). However, calls and returns† are more frequent, occurring every 33 instructions (SELF-only) and 25 instructions (SELF) vs. every 57 instructions in SPECint89. Interestingly, SELF’s runtime system (written in C++ and making heavy use of `inline` functions) has a higher call/return frequency than SELF code.

For a more accurate comparison, we exclude two instruction categories that are distorted by deficiencies in the back end of the current SELF compiler. The back end does not fill delay slots (except in fixed code patterns), and thus SELF programs contain many no-ops (6.2% vs. 0.45% for C‡). Also, it does not optimize branch chains or rearrange code blocks to avoid unconditional branches; thus, the code contains many unconditional branches (2.7% vs. 0.8% in C) that would be eliminated by a better back end. Since the presence of these extra instructions distorts the frequencies of other instruction categories, we will exclude no-ops and unconditional branches in future graphs.

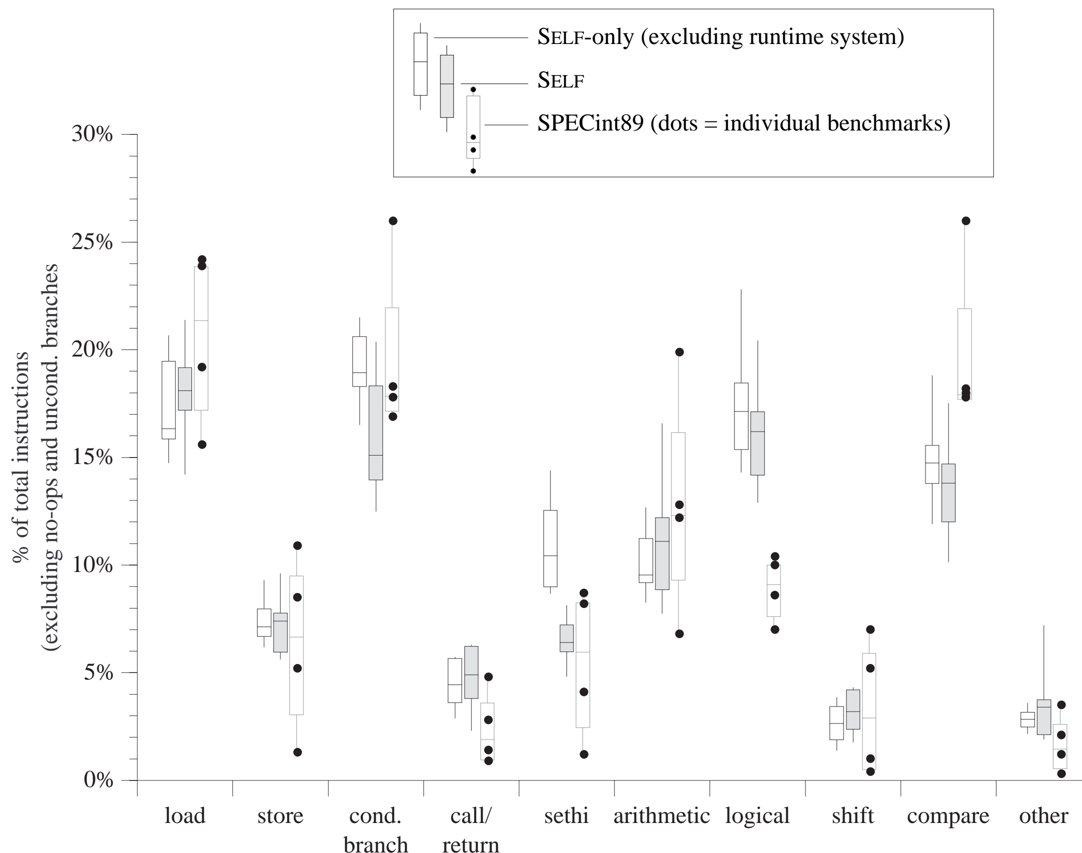


Figure 8-2. Dynamic instruction usage of SPECint89 vs. SELF-93 benchmarks (excluding no-ops and unconditional branches)

Figure 8-2 shows the adjusted execution frequencies. Comparing SELF and SELF-only, it appears that the runtime system does not influence the overall behavior much. The only two instruction groups showing a significant difference are conditional branches and `sethi` instructions. Conditional branches are more frequent in SELF-only than in

† Since SELF uses a non-standard calling convention and sometimes performs both a call and a (direct) jump to perform message dispatch, we are currently unable to measure the call frequency directly and use the call/return frequency instead. The call/return frequency is the frequency of all `jmpl` and `call` instructions combined. For C/C++, the call frequency is half the call/return frequency; for SELF, this is not the case (i.e., the call frequency is less than half of the call/return frequency since some calls involve both a `call` and a `jmpl`.)

‡ The percentages are the geometric means of the frequencies.

SELF, but not unusually high compared to the SPEC programs. `sethi` instructions are used to load 32-bit constants; they are more frequent in compiled SELF code for two reasons: first, the values `true`, `false`, and `nil` are objects in SELF, represented by 32-bit code constants; in contrast, C programs can use the short immediates 0 and 1. Second, object types are represented by the address of the type descriptor, and the implementation of message dispatch compares the receiver's type against the expected type(s). Since message dispatch is frequent, so are 32-bit constants.

Compared to the SPEC benchmarks, SELF shows few differences. Besides the differences in conditional branches and `sethi` instructions already mentioned above, only logical instructions and comparisons show a significant difference. Logical instructions are more frequent in SELF for two reasons. First, the compiler's back end uses only a simple register allocator, and as a result generates many unnecessary register moves.† (Move instructions account for roughly half of all logical instructions in SELF.) Second, SELF uses a tagged object representation with the two lower bits as the tag. Dispatch tests involving integers, as well as integer arithmetic operations, test their argument's tag using an `and` instruction; similarly, the runtime system (e.g., the garbage collector) often extracts an object's tag. Together, these `and` instructions account for about 25% of the logical instructions. We are unable to explain the remaining difference between SELF and the C programs because no detailed data on C's use of logical instructions was available.

SELF executes fewer comparisons than the SPEC integer benchmarks. This result surprised us; we expected SELF to execute *more* comparisons since message dispatch involves comparisons and is quite frequent. If SELF used indirect function calls to implement message dispatch, one could explain the lower frequency of conditional branches with the object-oriented programming style which typically replaces `if` or `switch` statements with dynamic dispatch; Calder et al. observed this effect when comparing C++ programs to the SPEC C programs [16]. However, since the SELF implementation does not use indirect function calls, we cannot explain the difference in this way. It is possible that SELF's optimizer eliminates enough dispatch type tests to lower the overall frequency of comparisons.

Instruction category	Frequency relative to SPEC	Reasons for difference
call/return	1.5x higher (SELF-only)	different programming styles; possible compiler deficiencies in SELF 3.0
logical instructions	about 1.8x higher (SELF-only)	extra register moves (inferior back end); integer tag tests (`and` instructions); unknown third factor (see text)
sethi (load 32-bit constant)	about 1.5x higher	`true`/`false`/`nil` are 32-bit constants; message dispatch involves comparisons with 32-bit constants
comparison instructions	about 1.5x lower	unknown (see text)

Table 8-2. Summary of main differences between SELF and SPECint89

Table 8-2 summarizes the main differences in instruction usage. Overall, there are few differences between SELF programs and the SPEC C programs, a surprising result considering that the two languages are very different. What is even more surprising is that SELF is closer to C than is C++. Figure 8-3 summarizes the C++ data from [16] and our own measurements. In each category, SELF is closer to SPEC than is C++. For example, the basic block size in C++ is 1.6 times higher than SPEC, but in SELF it is only 1.1 times higher than SPEC. (Some of the difference could be due to differences between the SPARC and MIPS compilers, but Calder et al. used the same compiler technology (GNU) for both C and C++.)

Apparently, execution behavior is more a function of compiler technology than of the source language characteristics. We believe that much of the difference between C++ (or other object-oriented languages) and C would disappear if compilers used OO-specific optimizations as well. Of course, until such systems are actually implemented, one

† Since moves are implemented using the `or` instruction, they are included as logical instructions.

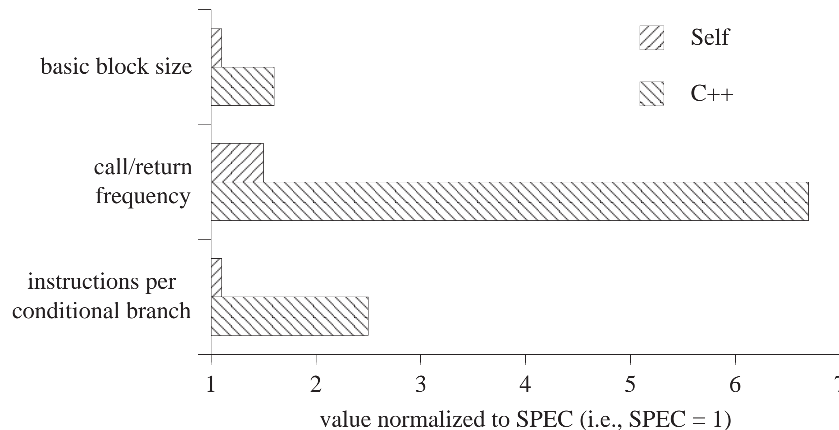


Figure 8-3. Differences between SELF/C++ and the SPEC C programs

cannot be certain that better optimization would indeed bring other object-oriented languages closer to C. However, in light of our data, any claims regarding the necessity of architectural support for an object-oriented language should be regarded with caution unless the system used for the study employs state-of-the-art optimization techniques, or unless such optimizations have been shown to be ineffective for that particular language.

8.2 Register windows

The SPARC architecture [118] defines a set of overlapping register windows which allows a procedure to save the caller's state by switching to a new set of registers. As long as the call depth does not exceed the number of available register sets, such a save can be performed in one cycle without any memory accesses. If there is no free register set when a save instruction is executed, a "register window overflow" trap occurs and the trap handler transparently frees a set by saving its contents in memory. Similarly, a "window underflow" trap is used to reload such a flushed register set from memory if it is needed again.

Register windows were originally proposed as part of the Berkeley RISC architecture, with the intention of making procedure calls as fast as possible [102]. If the variation in the procedure call depth does not significantly exceed the number of available register windows (usually, 7 or 8 on current SPARC implementations), most calls are executed without any memory accesses.

Are register windows good for SELF? Figure 8-4 shows box plots of the relative overhead of register window overflows and underflows for unoptimized SELF, SELF-93, and C++ (using Richards and DeltaBlue) as a function of the number of register windows available. The overhead is measured as a percentage of the time spent in a program's user-mode instructions. That is, the base time excludes any cache overhead and time spent in the window trap handlers. An overhead of 100% therefore means that execution time is twice as long as it would be with an infinitely large set of register windows. We truncated the graph at y = 100% in order not to lose resolution at the interesting points corresponding to the actual system (SELF-93 on a machine with 8 windows, 7 of which are available to user programs).

As expected, the window overhead decreases with more available register windows because there are fewer overflows and underflows. For 7 windows (the number of windows available in most SPARC implementations) the median overhead for SELF-93 is only 11%. The overhead for the C++ programs varies widely. For most versions of `Richards`, the overhead is zero (because it never exceeds a call depth of 7), but if compiled with Sun CC the overhead is 74% (apparently, the cfront-based compiler performs less inlining than GNU C++). Similarly, the overhead for `DeltaBlue` varies between 16% and 74% depending on the compiler and the version (virtual or not). Not surprisingly, unoptimized SELF programs have an extremely high overhead because they routinely have call depths much deeper than 8.† With 7 register windows, the median overhead for unoptimized SELF code is 140%.

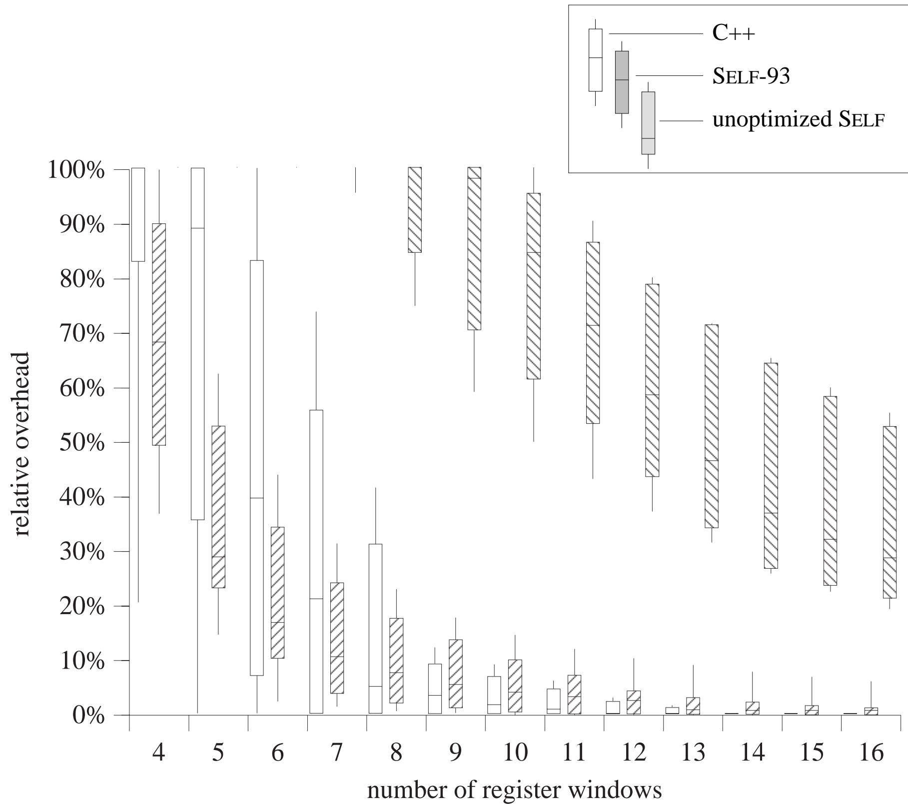


Figure 8-4. Register window overhead

For comparison, the SOAR architecture had 8 windows with 8 registers each; Ungar reports that only about 2.5% of all calls caused a register window overflow [132]. For unoptimized SELF this ratio is 9.4%. The reason for the significant difference is most probably that SELF doesn't hardwire common control structures but Smalltalk-80 does. Thus, `if` statements or loops contribute to the call depth in SELF but not in Smalltalk.

The above numbers are based on an overflow cost of 205 cycles and an underflow cost of 152 cycles, as measured empirically on a SPARCstation-2 running SunOS 4.1.3. The window overflow and underflow handlers are so expensive because they execute in supervisor mode, which leads to many complications. For example, the trap handler has to make sure that the loads and stores are executed with the memory access permissions of the user process, not of the kernel. The SPARC V9 architecture [119] corrects this flaw so that window trap handlers can execute in user mode. This modification reduces the window handling overhead by more than a factor of 10 (!) to 19 instructions per trap [47]. With this change, the register window handling overhead should become very small. Even with a reduction by a factor of only 5, the remaining overhead for SELF-93 would be negligible, and even the overhead for unoptimized code would be a modest 28% despite the extremely high call frequency and call chain depth. That is, register windows are cheap—even for programs with very high call depths—if the overflow and underflow traps are reasonably fast.

Register windows have two potential benefits. First, they simplify the compiler somewhat, since it does not have to keep track of which registers need to be saved across function calls, and how to minimize the resulting register spills. However, this effect is likely to be small, especially in an optimizing compiler. Second, register windows potentially save load and store instructions for such register spills. Do register windows increase SELF's performance? Since the

---

† For example, the equivalent of a `for` loop has a nesting depth of 13 since it is implemented with messages and closures. (The optimizing SELF compilers can usually eliminate all 13 levels.)

SELF-93 compiler lacks a register allocator that would consider register spills, we could not measure the number of memory references eliminated by register windows. However, we can get a rough estimate by assuming that without register windows each call would store 1.5 values on the stack, namely the receiver and 0.5 arguments (the average number of arguments is taken from Smalltalk [86]).† Non-leaf calls must save the return PC and frame pointer (2 words); we also assume they save two locals.‡ Lacking any data, we assume that 50% of the calls are non-leaf calls. Together, the average call stores and reloads 3.5 words. With calls representing roughly 1.5% of all instructions in SELF-93, this would increase the number of instructions by 10.5% (3.5 loads and 3.5 stores per call). On our reference machine, loads take 2 cycles and stores take 3, so the overhead in terms of cycles would be 26% (ignoring cache effects). For unoptimized code, the relative overhead would be roughly twice as high since calls are twice as frequent.

Based on this rough estimate, it appears that SPARC V8 register windows (with a high trap overhead) improve SELF-93's performance (11% vs. about 26% overhead for storing and restoring values) but significantly increase the execution time of unoptimized code (140% vs. about 52% overhead). With SPARC V9 register windows (at least five times lower trap overhead), performance is improved in both cases, by about 24% (26% - 11/5%) for optimized code and about the same 24% (52% - 140/5%) for unoptimized code. However, such a machine would probably also offer one-cycle loads and stores, reducing the performance advantage of register windows to about 8% (10.5% - 11/5%) for optimized code and -7% (21% - 140/5%, i.e., a disadvantage) for unoptimized code.

In contrast, Ungar reports that SOAR would have been 46% slower without register windows [132]. This discrepancy stems from the non-optimizing compiler used in SOAR which did not perform any inlining, resulting in a higher call frequency. Furthermore, Ungar's numbers overestimated the benefits of register windows because the number of registers to be saved per call was assumed to be the number of non-nil (i.e., used) registers at the method return.††

Thus, although these estimates are fairly rough, it appears that register windows do not dramatically improve performance for SELF.‡‡ In the most likely scenario, they speed up optimized code by 8% and slow down unoptimized code by 7%.

8.3 Hardware support for tagged arithmetic

Tagged integer addition and subtraction instructions are a unique SPARC feature that was motivated by the results of the SOAR project. SOAR Smalltalk would have been 26% slower without them [132]. But how useful are they for a system with an optimizing compiler such as SELF-93? To answer this question, we first need to look at the code generated for integer addition (or subtraction) in SELF-93. Assuming `x` and `y` are integers, the expression `x + y` is compiled into the following code:

† The values to be saved are the receiver and arguments of the caller, not the callee; we assume that arguments are passed in registers so that argument passing itself involves no memory operations. Also, since we have no data on the average depth of the expression stack (i.e., the number of arguments to future sends that have already been evaluated but not consumed), we ignore the possible contribution of this category of values.

‡ The number of locals probably is the most uncertain factor in this estimate. On one hand, optimized methods are likely to have more than two locals or not-yet-consumed outgoing arguments. On the other hand, only the variables actually updated (used) between two calls need to be saved (restored).

†† Some of these non-nil registers could have held expression stack temporaries (evaluated but not yet consumed arguments of a message send), which are live during only part of the method and thus need not be saved at every call.

‡‡ Although the estimates are rough, the estimates using the most likely machine configuration (fast traps and one-cycle loads and stores) are fairly stable. Had we underestimated the save/restore memory traffic by a factor of two, the advantage of register windows would be 20% (21% - 11/5%) for optimized code and 14% (42% - 140/5%) for unoptimized code, which is still a long way from the 46% reported by Ungar.

```
if (x is integer) {                          // 3 instructions (test & branch, unfilled delay slot)
    tagged_add(x, y, temp);                  // 1 instruction
    if (overflow flag set) {                 // 1 instruction (cond. branch)
          // handle error case: overflow or non-integer argument
    } else {
          result = temp;                     // assign result (1 instruction, in delay slot of
                                             // above branch)
    }
} else {
    // handle non-integer case
}
```

The tagged_add instruction sets the overflow bit if x or y have the wrong tags (nonzero lowest two bits) or if the addition overflows. The type test for x is the type feedback test for the "+" message and may be omitted if x's type is known; in general, there may be other instructions between the type test and the tagged add, depending on the definition of the "+" method for integers. (Currently, all these instructions can be optimized away for the standard definition.) Thus, in general, an integer addition takes 6 instructions in SELF-93.

The ideal code sequence for addition, given the tagged addition instruction, is one instruction:

```
tagged_add_trapIfError(x, y, result);
```

This variant of the tagged add instruction (taddcctv in SPARC syntax) assigns the result register only if both x and y have integer tags and if the addition does not overflow. Otherwise, a trap occurs. SELF-93 currently does not use this variant because the runtime system would need some additional mechanisms to handle the traps and recompile the offending compiled code if necessary (to avoid taking frequent repeated traps if the integer type prediction is no longer valid). However, this support would not be hard to add. Therefore, to get an upper bound on the benefits of tagged instructions, we will assume that the ideal system would make maximal use of the tagged arithmetic support and execute all integer additions and subtractions in one cycle.

Finally, here is the code sequence SELF-93 could use if SPARC did not have tagged instructions:

```
if (x is integer) {                          // 3 instructions (test, cond. branch, unfilled delay slot)
    if (y is integer) {                      // 2 instructions (fill delay slot with add)
          add(x, y, temp);                   // 1 instruction
          if (overflow flag set) {           // 1 instruction (cond. branch)
                // handle error case: overflow or non-integer argument
          } else {
                result = temp;               // assign result (1 instr., delay slot of above branch)
          }
    } else {
          // handle non-integer case
    }
} else {
    // handle non-integer case
}
```

This sequence needs only two additional instructions for the extra type test, for a total of 8 instructions. Tagged arithmetic instructions therefore save a maximum of 7 instructions per arithmetic operation. All of the above code sequences assume an unsophisticated back end like that of SELF-93; with better optimization, the code sequences could be shortened. For example, a better code for the sequence not using tagged instructions would fill delay slots, use only one tag test (by first ORing x and y), and eliminate the extra assignment, for a savings of 3 or 4 instructions. For our comparison, we assume a 6-instruction sequence per integer addition or subtraction.[†]

In addition to integer additions and subtractions, tagged instructions are also useful in integer comparisons[‡] and other integer tag tests (e.g., to verify that the index in an array indexing operation is an integer). For each integer tag test in these operations, we assume an overhead of 2 cycles (test + branch + unfilled delay slot – tagged add).

Therefore, the number of cycles saved by tagged arithmetic instructions is 5 * number_of_adds + 5 * number_of_-subtracts + 2 * number_of_integer_tag_tests. Table 8-3 shows the frequencies of these operations in SELF-93, assuming they would take one cycle each.† Integer arithmetic instructions are used rarely and represent only about 0.6% of all instructions on average; integer tag tests are more frequent, with a median of 2.2%. Without hardware support for tagged integers, our estimate predicts that SELF would be 7% slower than a system making maximal use of tagged instructions.

	% integer additions	% integer subtractions	% integer tag tests	estimated slowdown without hardware support
CecilComp	0.48%	0.10%	2.20%	7.3%
CecilInt	0.10%	0.05%	0.58%	1.9%
DeltaBlue	0.64%	0.33%	4.63%	14.1%
Mango	0.29%	0.16%	1.74%	5.7%
PrimMaker	0.57%	0.04%	1.92%	6.9%
Richards	0.40%	0.36%	2.46%	8.7%
Typeinf	0.36%	0.17%	2.01%	6.7%
UI1	0.56%	0.09%	2.63%	8.5%
UI3	0.67%	0.28%	2.86%	10.5%
Median	0.48%	0.16%	2.20%	7.3%

Table 8-3. Arithmetic operation frequency and estimated benefit of tagged instructions

However, the true savings are likely to be smaller since our estimate makes several simplifying assumptions that all tend to inflate the benefits of tagged arithmetic instructions:

- Counting instructions rather than cycles overestimate the savings because the code sequences are free of memory accesses and could thus execute at close to 1.0 CPI (cycles per instruction), whereas the overall CPI is closer to 2 for our reference machine. Thus, the sequences replacing tagged instructions consume about 7% of all instructions but only about 3.5% of all cycles. The exact extent of this overestimation is of course machine-dependent, but we believe the non-tagged instructions could execute with a below-average CPI even on superscalar architectures with high branch penalties because the branches are extremely predictable (the integer type test will almost always succeed, and overflows are rare). Furthermore, existing superscalar SPARC implementations cannot execute the tagged instructions in parallel with other instructions [123] because they can trap, and so one tagged instruction consumes as much time as several other instructions (up to three for SuperSPARC [123]).
- We assumed that the compiler has no type information on the arguments of integer additions or subtractions, and thus has to explicitly test both tags for all integer operations. This overestimates the cost of explicit tag checking since one argument may be a constant (e.g., when incrementing a loop index by 1) or otherwise known.

For these reasons, we consider our estimate of 7% to be an upper bound of the benefits of tagged instruction support for the programs in our benchmark suite.‡ In our reference system, a more accurate upper bound is 4% of execution

† Replacing the whole integer addition sequence with a trapping tagged add requires the same optimization as folding the two tag test into one does (using an `or`): the compiler has to recognize that there are no instructions between the dispatch test (first integer tag test) and the addition (with the argument tag test). However, the fact that there are no instructions between the two tests is a result of the current definition of the SELF method for smallInteger addition; since the definition can be changed by the user, the compiler could not hardwire the code sequence. Thus, if we assume a one-instruction sequence for the system using tagged instructions, we also have to assume a 6-instruction sequence (folded tag tests) for the system not using tagged instructions.

‡ Comparisons differ from subtractions in that they do not have to check for arithmetic overflow.

† I.e., the frequencies relative to a system making maximum use of tagged instructions.

time. Thus, it appears that removing the instructions for tagged addition and subtraction from the SPARC architecture would not significantly reduce SELF-93's performance.

This result stands in marked contrast to Ungar's measurements showing that SOAR would have been 26% slower without instructions for tagged arithmetic [132]. Why the big difference? By analyzing Ungar's data, we could find several reasons for the higher estimate:

- Ungar's data includes speedups from several tagged instructions (such as "load" and "load class") that are not present on the SPARC or not needed in SELF-93.[†] Including only the tagged arithmetic instructions, Ungar's estimate would be 12%.
- The SOAR code sequences for integer arithmetic without tagged instructions are slowed down because SOAR does not have a "branch on overflow" instruction. With such an instruction, the code sequences would become much shorter, reducing the estimated slowdown to 8%.
- Integer arithmetic is more frequent in SOAR than in SELF-93. In Ungar's benchmarks, they represent 2.26% of all instructions vs. 0.64% in SELF-93. We do not know whether this difference is a result of compiler differences or of differences in the benchmarks.

Together, these factors explain much of the difference between our estimates and the SOAR estimates.

SELF-93 does not significantly benefit from the tagged addition and subtraction instructions present in the SPARC architecture: a SELF system making ideal use of these instructions would save only 4% of total execution time.

8.4 Instruction cache behavior

As part of this performance analysis, we measured the cache behavior of SELF-93 for a variety of cache configurations. All caches have the same organization as our reference machine (two-way associative, 32-byte lines, write-allocate with subblock placement for the data cache, 26 cycles miss penalty). The most interesting result of our analysis is that for medium-sized caches SELF-93 spends a significant amount of time in instruction cache misses and relatively little time in data cache misses. Figure 8-5 shows the execution time overhead relative to a system with an infinitely fast memory system; for example, an overhead of 40% means that the system runs 1.4 times slower than the ideal system.[‡] For the 32K instruction cache used in the reference system, instruction cache overheads range from virtually zero (for the two small benchmarks, `Richards` and `DeltaBlue`) to almost 70% for `CecilInt`, with a median of 40%. For all sizes measured, doubling the cache size roughly halves the median overhead. By comparing the results with another simulation using a 32-way associative cache, we determined that the misses are dominated by capacity misses and not by conflict misses, so increasing cache associativity would not help. Most of the programs in the benchmark suite are large (500 Kbytes of code or more), and it appears that a 32K cache is not big enough to hold the programs' working sets. A 128K cache, however, would reduce the miss overhead to modest levels (median 10%).

SELF-93's code is about 25% larger than that of SELF-91 or SELF-93-nofeedback, as we have seen in Section 7.5.6. How much does the increased code space contribute to the cache overhead? To obtain an approximate answer, we measured the cache behavior of SELF-91, the system generating the smallest code (Figure 8-6). The instruction cache overhead in SELF-91 is about half that of SELF-93 up to a cache size of 128K, and even lower for larger caches. For

[‡] Of course, programs with a higher frequency of integer arithmetic (such as the Stanford integer benchmarks) could benefit more from tagged arithmetic instructions. However, this class of programs is also more amenable to optimizations that eliminate type checks (see Section 7.5.4).

[†] For example, the "load class" instruction is handled equally fast in SELF-93 with a normal load instruction (see Section 6.3.1).

[‡] We use time overhead rather than miss ratios because comparing miss ratios can be misleading. First, write misses are free since we assume they can be absorbed by a write buffer or write cache [14, 81]. Furthermore, data miss ratios are not directly related to execution time if (as is customary) they are given relative to the number of data references since the density of loads may vary from program to program. For reference, Appendix A (starting on page 152) contains miss ratios for all programs.

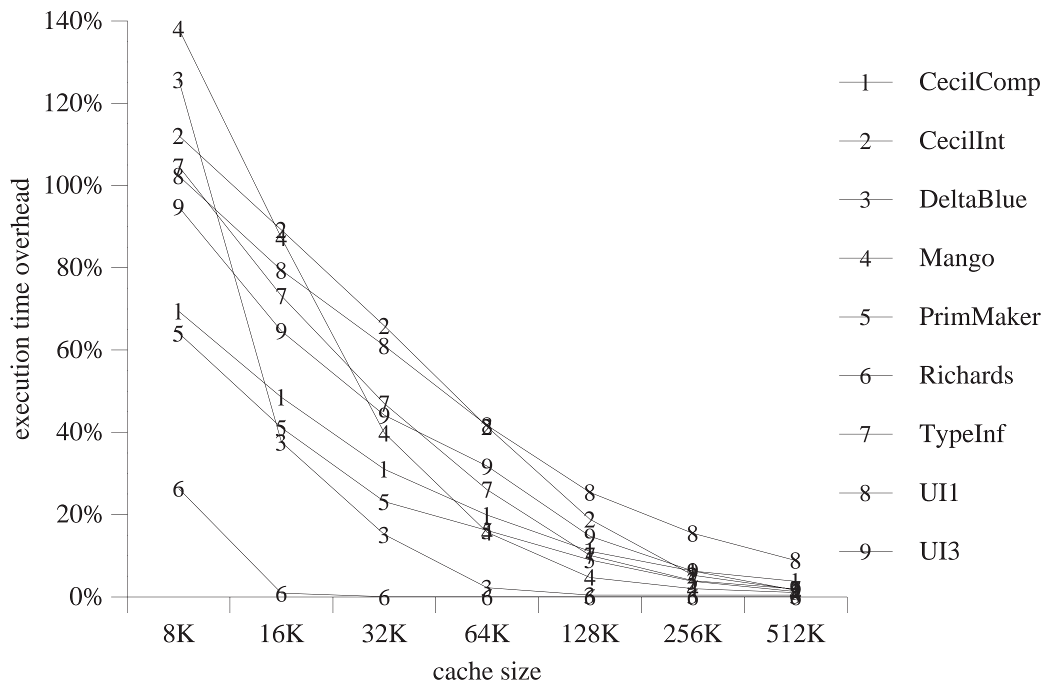


Figure 8-5. Time overhead of instruction cache misses in SELF-93

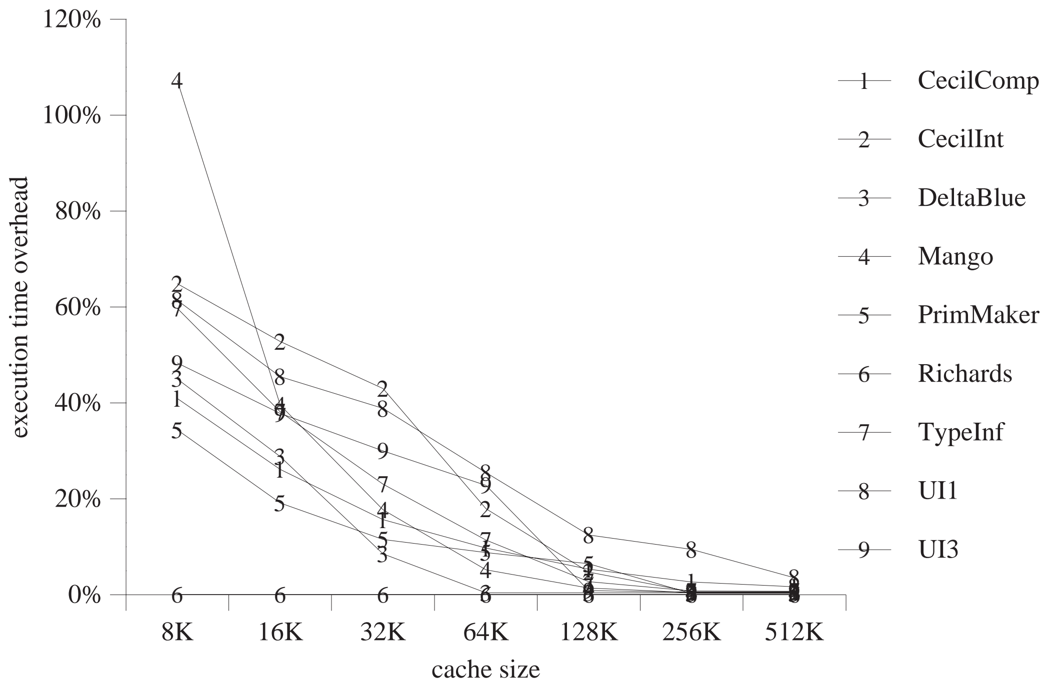


Figure 8-6. Time overhead of instruction cache misses in SELF-91

the 32K cache in the reference system, SELF-91 is only 20% slower than a system with an ideal memory system as opposed to a slowdown of 40% in SELF-93. Despite the increased I-cache overhead, SELF-93 runs considerably faster.

To summarize, instruction cache misses have a substantial performance impact on SELF-93. With the 32K instruction cache of our base system, SELF-93 runs 40% slower than it would with an infinitely large cache. Doubling the cache size to 64K halves the performance penalty, and halving the cache size doubles the penalty. Thus, the size of the instruction cache can have a larger influence on SELF-93's performance than register windows and tagged instructions combined.

Machine	Cache Parameters		Memory latency (cycles)
	Instruction	Data	
Reference machine	32K, 2-way	32K, 2-way	26
SPARCstation-2	64K unified, 1-way		26
SPARCstation-10/41	20K, 5-way	16K, 4-way	5 (40[a])
DECstation 5000/240	64K, 1-way	64K, 1-way	15
HP 9000/720	128K, 1-way	256K, 1-way	16-18

Table 8-4. Cache parameters of current workstations

a. 5 cycles to the 1 MByte external cache, approximately 40 cycles to main memory

However, we should emphasize that the large cache overhead is in part a result of the conservative parameters for the cache design of our reference machine. In particular, the main memory latency of 26 cycles, while consistent with the SPARCstation-2 cache, is on the upper end of typical caches in current workstations (Table 8-4). However, we felt that conservative cache parameters would be more helpful in predicting performance on future systems since the gap between processor (cache) and main memory speeds is expected to widen in the future. Other studies have shown that the I-cache overhead of scientific and commercial workloads can be similar to the overhead measured for SELF. For example, Cvetanovic and Bhandarkar [38] report an I-cache miss overhead of 30-40% of total execution time for some SPECint92 benchmarks and the TCP benchmarks running on a DEC Alpha system.

8.5 Data cache behavior

Figure 8-7 shows that the data cache overhead is much smaller. For a 32K data cache, the median overhead relative to an infinite cache is only 6.6%. Doubling the cache size usually reduces the median overhead by about a factor of 1.5. Even with an 8K data cache, the overhead would still be very modest, with a median of 12%. This is quite surprising since most of the programs allocate several MBytes of data.† However, our data is consistent with that of Diwan et al. who have measured allocation-intensive ML programs [48] and found very low data cache overheads for the same cache organization (write-allocate, subblock placement). Our data also confirms that of Reinhold [108] who investigated the cache performance of large Lisp programs.

Diwan et al also measured that the data cache overhead of their ML programs increases substantially with a write-noallocate policy, i.e., with a cache that does not allocate a cache line on a write miss. This is also true for SELF-93 (see Figure 8-8). For a 32K cache, write-noallocate increases cache overhead by a median factor of 1.6, and this factor increases with larger caches up to a factor of 2.1 for a 512K cache. With write-noallocate, the write miss ratio is truly staggering, ranging from 22% to 70% (!) for a 32K cache and barely decreasing with larger caches (see Figure 8-9). Although there is no cost associated with write misses in our model, it is interesting to look at these extremely high miss ratios because they help explain why write-noallocate is an inferior cache policy for allocation-intensive systems like SELF-93.

The write miss ratio is so high because of the way objects are allocated in a system with a copying generational garbage collector (see Figure 8-10). Typically, such systems allocate objects consecutively in a "creation space"

† See Table A-16 in Appendix A.

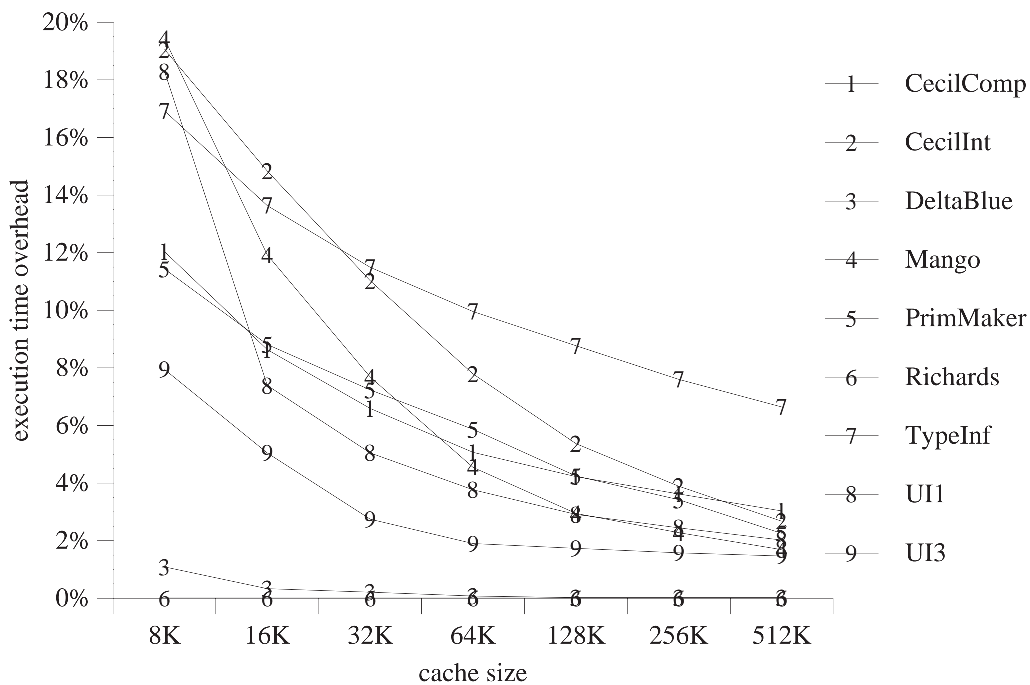


Figure 8-7. Time overhead of data cache misses in SELF-93
(write-allocate with subblock placement)

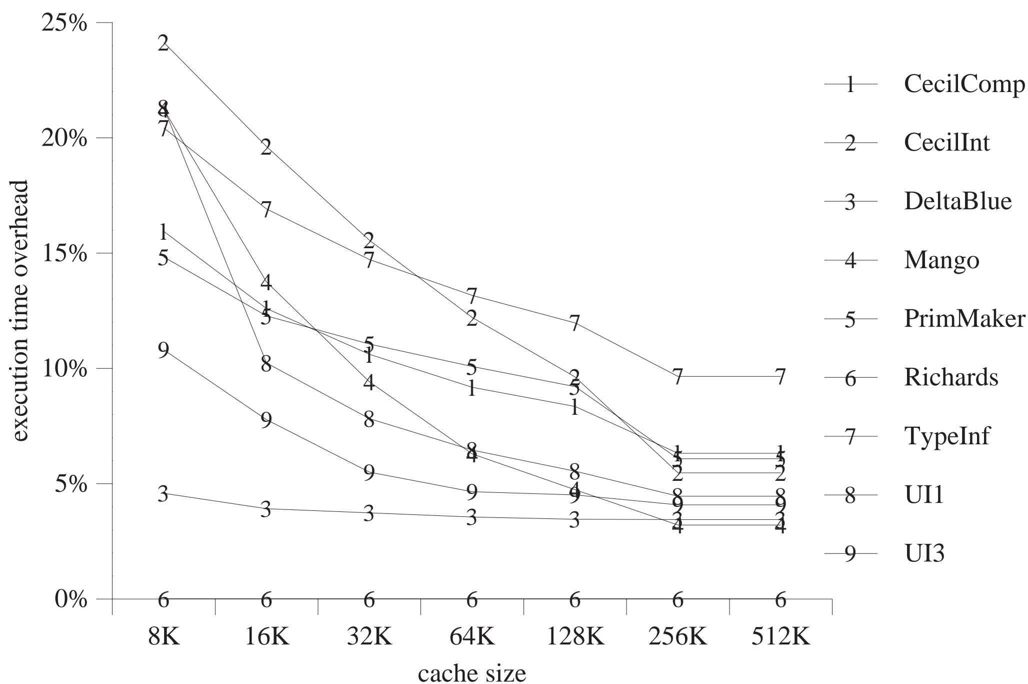


Figure 8-8. Time overhead of data cache misses in SELF-93
(write-noallocate cache)

[131]. After every scavenge, this creation space is empty. Therefore, all the system has to do to allocate an object is to

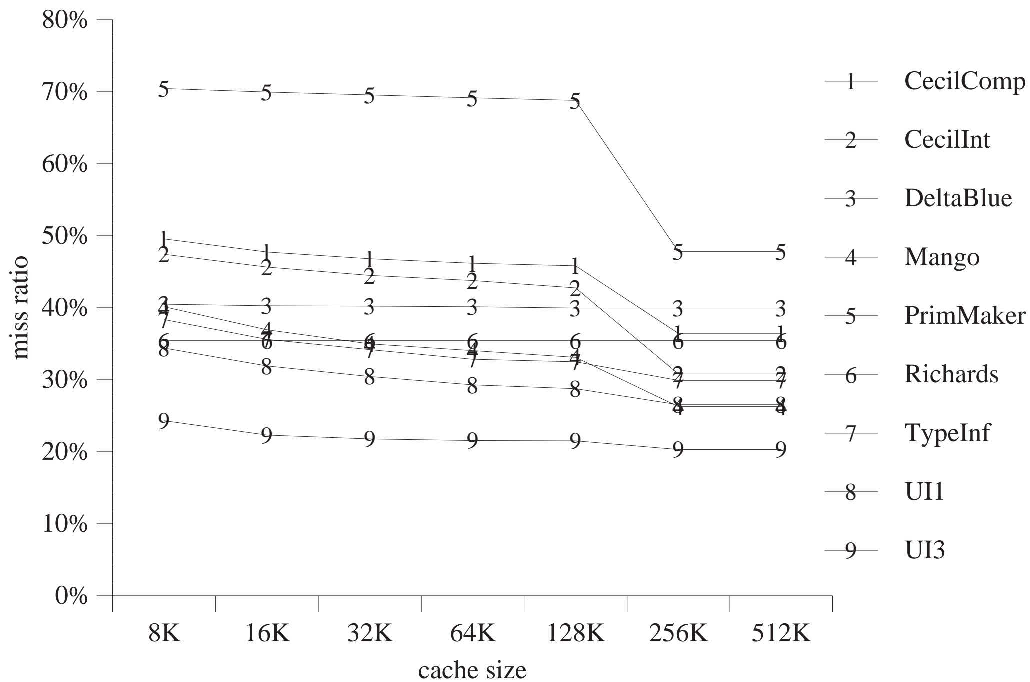


Figure 8-9. Write miss ratio for SELF-93 (write-noallocate cache)

increment the pointer marking the start of the available space; if the pointer points beyond the end of creation space, a scavenge is initiated, which evacuates all live objects from the creation space to another area of memory.

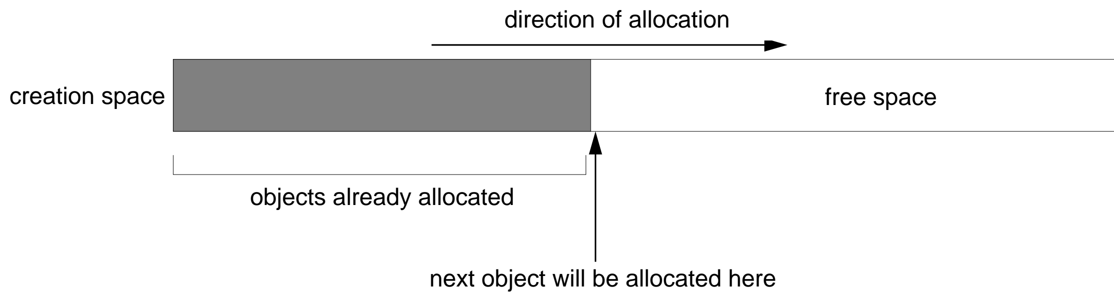


Figure 8-10. Object allocation in SELF

The creation space is much larger than most caches—400 Kbytes in SELF. The memory locations of a newly allocated object are virtually guaranteed not to be in the cache, since the allocator has touched a contiguous block of 400 Kbytes since the last time an object was allocated at this address. Thus, the stores initializing the object will all miss the cache in a write-noallocate cache since all words of the object are initialized before they are read for the first time; the first read will cause an additional cache miss which finally allocates a cache line.†

In contrast, a write-allocate cache will allocate a cache line on the first write miss so that all subsequent accesses to the object are cached. If the cache also uses subblock placement, no cache penalty is incurred since the current contents corresponding to the newly allocated line need not be read from memory (the subblocks containing invalid

† Obviously, this cache behavior could be improved with prefetching, i.e., by executing the load first to bring the line into the cache. With non-blocking loads, prefetching could possibly allow write-noallocate caches to approach the miss ratios of write-allocate caches.

data are marked as such). Without subblock placement, the old contents of memory would be read unnecessarily, only to be overwritten by the initializing stores.

Wilson et al. [140] argue that this allocation behavior will result in especially high cache overheads for direct-mapped caches since the allocation pointer will sweep through the entire cache and evict every single cache line. Therefore, caches should be associative, and the size of creation space should be smaller than the cache size. To test this hypothesis, we measured a version of SELF-93 using a 50Kbyte creation space to determine the cache impact. The result showed that is not desirable to run SELF with such a small creation space: overall execution time increased by roughly a factor of 1.5, mostly because of the increased garbage collection overhead. (With a 50Kbyte creation space, scavenges become eight times more frequent but not eight times less expensive (per scavenge) since the root processing time remains constant; also, object survival rates increase because of the shorter time intervals between scavenges.)

Furthermore, our experiment did not confirm the basic assumptions of Wilson et al. The data read miss ratio was substantially *higher* than in the reference system for almost all benchmarks and cache sizes. For example, for caches between 64K and 256K, the median miss ratio doubled.[†] We attribute this increase to the increased data references made by the scavenger which pollute the program's cache, although we have no data to prove this hypothesis. However, it is consistent with the fact that the instruction cache miss ratio is only half that of the reference system even though we did not change any of the I-cache parameters. The reduced I-cache misses could be a result of the frequent scavenges—if scavenges occur often enough, the scavenger's instructions may remain cache-resident. Also, the scavenger is fairly small and is thus likely to have better cache behavior than the overall system.

In conclusion, the data cache behavior of SELF-93 is fairly good despite its high allocation rates. Because most data accesses go to objects recently created, and because of the sequential nature of allocation, many accesses hit the cache. With a 32Kbyte data cache, the median D-cache overhead is less than 7%.

8.6 Possible improvements

Given the results of our performance evaluation, what are possible improvements that could speed up SELF-93? Table 8-5 shows a list of such improvements for SELF-93.

Source of improvement		see section(s)…	estimated speedup
software improvements	Better back end (register allocation, peephole optimization)	6.4, 7.3, 8.1	10-25%
	Eliminating unnecessary type tests	7.5.4	5%
	Reducing code size (better inlining decisions, interpreter)	8.4, 5.4, 7.5.5, 4.5	0-20%
hardware improvements	64K instruction cache (instead of 32K)	8.4	20%
	SPARC V9 register windows	8.2	6%

Table 8-5. Sources of possible performance improvements

Except for the gains from hardware improvements (last two entries), the speedups listed in the table are rough estimates based on the measured overheads and an estimate of what fraction could be eliminated. The two main areas for improvement are:

- *A better back end incorporating standard, well-known code optimizations.* Our estimate is based on inspecting the code generated by the current compiler and the speedup that such optimizations typically bring in compilers for other languages. Based on the frequency of unfilled delay slots, extra register moves, and branch chains, we estimate that a relatively simple postpass over the generated code could already bring a speedup of 5-10%.

[†] See Appendix A for detailed figures (page 154).

However, extensive optimizations may not be compatible with an interactive system since compile pauses could increase too much (see next chapter).

- *Reducing instruction cache misses.* Since SELF-93 spends 40% of its time in I-cache misses with a 32K cache, reducing these misses could bring significant performance benefits. One technique that could help reducing code size is a system making better-informed inlining decisions (similar to those proposed by Dean and Chambers [40]), taking into account not only the size of the inlining candidate but also the potential speedup. Such a system could possibly reduce code size without significantly increasing the number of instructions executed, and thus reduce overall execution time because of the reduced cache overhead. However, we do not have enough data to judge whether such an approach would be successful.

Taken together, the speedup estimates reveal that the speed of SELF-93 could be improved significantly at the expense of a more complicated compiler and slower compilation. Of course, the estimated speedups are not additive; for example, if the software techniques succeed in halving code size, a 64K cache would bring only an additional 10% speedup. Nevertheless, the combined speedup of the software techniques is substantial and appears to be a promising area for future research.

8.7 Conclusions

The SELF-93 compiler reduces SELF's pure message-passing language model to programs whose instruction usage is very similar to that of the SPEC'89 integer benchmarks. After factoring out the effects of the relatively naive code generator (generating more no-ops and register moves), the most significant difference is that SELF-93's call frequency is about two times higher. The basic block sizes, however, are very similar. Surprisingly, programs compiled by SELF-93 differ much less from the C programs in the SPEC'89 suite than do C++ programs. We believe that much of the difference between C++ (or other object-oriented languages) and C would disappear if compilers used OO-specific optimizations as well.

The data cache behavior of SELF-93 is very good even for small caches; with a 32K data cache and a memory latency of 26 cycles (a relatively high latency), the median overhead compared to an ideal system with infinitely fast memory is only 6.6%. Reducing the size of the allocation area to fit the cache size does not improve performance since scavenges occur too often with the smaller allocation area. For the same reason, data cache misses do not decrease with a smaller allocation area. The instruction cache overhead is substantially higher, with a median overhead of 40% for a 32K instruction cache. Doubling (halving) the cache size roughly halves (doubles) the overhead.

Other hardware features have a smaller impact on performance. Register windows simplify the compiler (and probably speed up compilation), but they do not improve performance on the SPARCstation-2 because window handler traps are expensive to handle. With cheaper window traps—as specified in the upcoming SPARC V9 implementations—register windows would improve SELF-93's performance by an estimated 8%. The performance benefit of tagged integer arithmetic instructions is even smaller; compared to a system making ideal use of these instructions (which would require a more sophisticated compiler back end), SELF-93 without the special instructions would be at most 4% slower.

Since compiled SELF-93 code is similar to C code, it should run efficiently on standard hardware which usually was designed for C-like programs. In light of these results, we believe that any arguments in favor of architectural support specific to an object-oriented language should be regarded with caution unless the system used for the study employs state-of-the-art optimization techniques, or unless such optimizations have been shown to be ineffective or impractical for that particular language.

9. Responsiveness

The previous chapter examined SELF's performance in terms of the speed of compiled code. But in an interactive environment intended for rapid prototyping, raw runtime performance is only one aspect of performance. The other aspect is responsiveness: how quickly does the system react to programming changes (i.e., how quickly can the programmer try out a new piece of code after typing it in)?[†] Since SELF uses runtime compilation (interpretation would be too slow), compile pauses impact the interactiveness of the system. For example, the first time a menu is popped up, the code drawing the menu has to be compiled. Therefore, runtime compilation may create distracting pauses in such situations. Similarly, adaptive recompilation may introduce pauses during later runs of that code as it gets optimized. This chapter explores the severity of compilation pauses with a variety of measurements such as compilation speed, the distribution of compile pauses, and the pauses experienced in an actual interactive session. Part of this work is a definition of *pause clustering*, a new method of characterizing pauses that takes into account the way pauses are experienced by the user. In addition, we measure the influence of several parameters of the recompilation system on performance variations and on final performance.

Unlike the measurements of previous chapters, the performance numbers in this chapter were not obtained with a simulator because many measurements involve interactive use of the system. Instead, we measured CPU times on an otherwise idle SPARCstation-2 with 64 MBytes of memory. Due to the cache-related performance fluctuations discussed in Chapter 7, measurements are probably only accurate to within 10-15%.

To obtain our measurements, we ran instrumented versions of SELF. Individual compilation times were measured by comparing process CPU time before and after a compilation. The data in Section 9.4 were obtained by PC sampling, i.e., by interrupting the program 100 times per second and inspecting the program counter to determine whether the system was compiling or executing code at the time of the interrupt. The results of these samples were written to a log file. Instrumentation slowed down the system by about 10%.

9.1 Pause clustering

One of the main goals of this chapter is to evaluate SELF-93's interactive performance by measuring compile pauses. But what constitutes a compile pause? It is tempting to measure the duration of *individual* compilations; however, such measurements would lead to an overly optimistic picture since compilations tend to occur in clusters. When two compilations occur back-to-back, they are perceived as one pause by the user and thus should be counted as a single long pause rather than two shorter pauses. That is, even if individual pauses are short, the user may notice distracting pauses if many compilations occur in quick succession. Since the goal is to characterize the pause behavior *as experienced by the user*, "pause" must be defined in a way that correctly handles the non-uniform distribution of pauses in time.

Pause clustering attempts to define pauses in such an interaction-centered (physiological) way. A pause cluster is any time period satisfying the following three criteria:

1. A cluster starts and ends with a pause.
2. Pauses consume at least 50% of the cluster's time. Thus, if many small pauses occur in short succession, they are lumped together into one long pause cluster as long as the pauses consume more than half of CPU time during the interval. We believe that a limit of 50% is conservative since the system is still making progress at half the normal speed, so that the user may not even notice the temporary slowdown.

[†] Of course, such "latency vs. throughput" problems are well-known in other areas, e.g., operating systems.

3. A cluster contains no pause-free interval longer than 0.5 seconds. If two groups of pauses that would be grouped together by the first two rules are separated by more than half a second, we assume that they are perceived as two distinct pauses and therefore do not lump them together. (It seemed clear to us that two events separated by half a second could be distinguished.)

Figure 9-1 shows an example. The first four pauses are clustered together because they use more than 50% of total execution time during that time period (rule 2). Similarly, the next three short pauses are grouped with the next (long) pause, forming a long pause cluster of more than a second. The two clusters won't be fused into one big 2.5-second cluster (even if the resulting cluster still satisfied rule 2, which it does not in the example) because they are separated by a pause-free period of more than 0.5 seconds (rule 3).

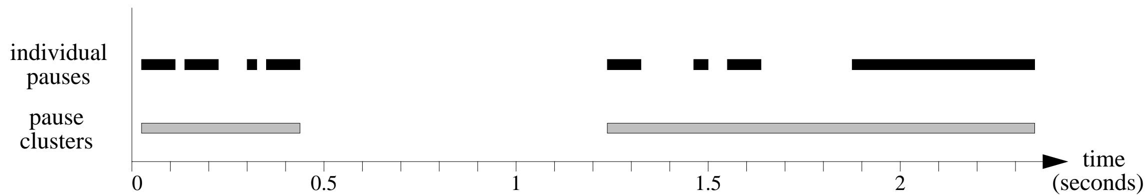


Figure 9-1. Individual pauses and the resulting pause clusters

This example illustrates that pause clustering is quite conservative and may overestimate the true pauses experienced by the user.† However, we believe that pause clustering is more realistic than measuring individual pauses. Furthermore, we hope that this approach will strengthen our results since the measured pause behavior is still good despite the conservative methodology. We also hope that this work will inspire others (for example, implementors of incremental garbage collectors) to use similar approaches when characterizing pause times.

Figure 9-2 shows the effect of pause clustering when measuring compile pauses. The graph shows the number of compile pauses that exceed a certain length on a SPARCstation-2. By ignoring pause clustering we could have reported that only 5% of the pauses in SELF-93 exceed 10 milliseconds, and that less than 2% exceed 0.1 seconds. However, with pause clustering 37% of the combined pauses exceed 0.1 seconds. *Clustering pauses makes an order-of-magnitude difference.* Reporting only individual pauses would result in a distorted picture.

Of course, the parameter values of pause clustering (CPU percentage and intergroup time) will affect the results. For example, increasing the pause percentage towards 100% will make the results more optimistic. However, the results presented here are fairly insensitive to changes in the parameter values. In particular, varying the pause percentage between 35% and 70% does not qualitatively change the results, nor does doubling the intergroup time to one second.

9.2 Compile pauses

This section measures the pauses occurring during an interactive session of the SELF user interface. All pauses measurements use pause clustering unless specifically mentioned otherwise. That is, the term "pause" always means "clustered pause."

9.2.1 Pauses during an interactive session

We measured the compilation pauses occurring during a 50-minute session of the SELF user interface [28]. The session involved completing a SELF tutorial, which includes browsing, editing, and making small programming changes. During the tutorial, we discovered a bug in the cut-and-paste code, so that the session also includes some "real-life" debugging.

† Pause clustering may also be too conservative for compilation pauses because it ignores execution speed; a SELF interpreter could be so slow that it causes distracting interaction pauses. For the sake of simplicity, we assume that an interpreter would be fast enough for interactive use.

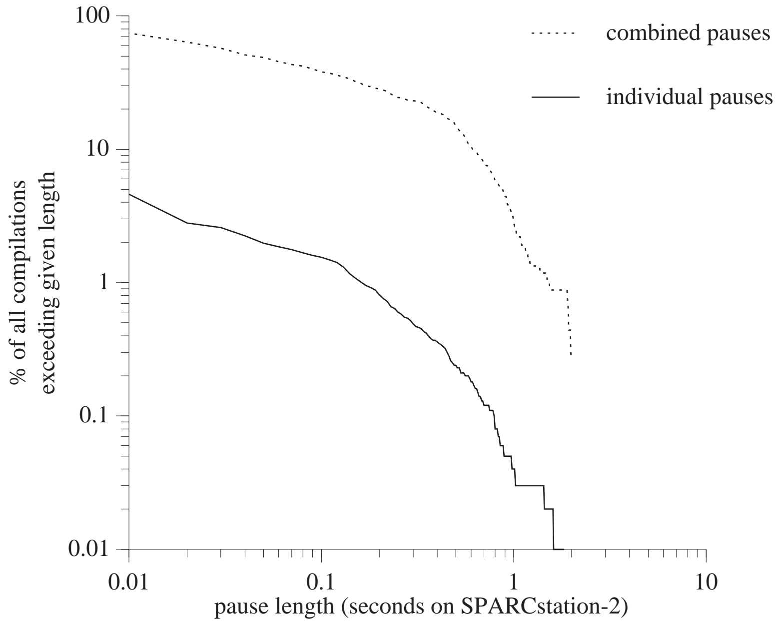


Figure 9-2. Distribution of individual vs. combined compile pauses

Figure 9-3 shows the distribution of compile pauses during the experiment.[†] Almost two thirds of the measurable pauses[‡] are below a tenth of a second, and 97% are below a second. Figure 9-4 shows the same data in absolute terms; the first two bars were truncated at y = 50 to show more detail in the rest of the histogram. If we use 200 ms as a lower threshold for perceptible pauses, then only 195 pauses exceeded this threshold. Similarly, if we use one second as the lower threshold for distracting pauses, there were 21 such pauses during the 50-minute run.

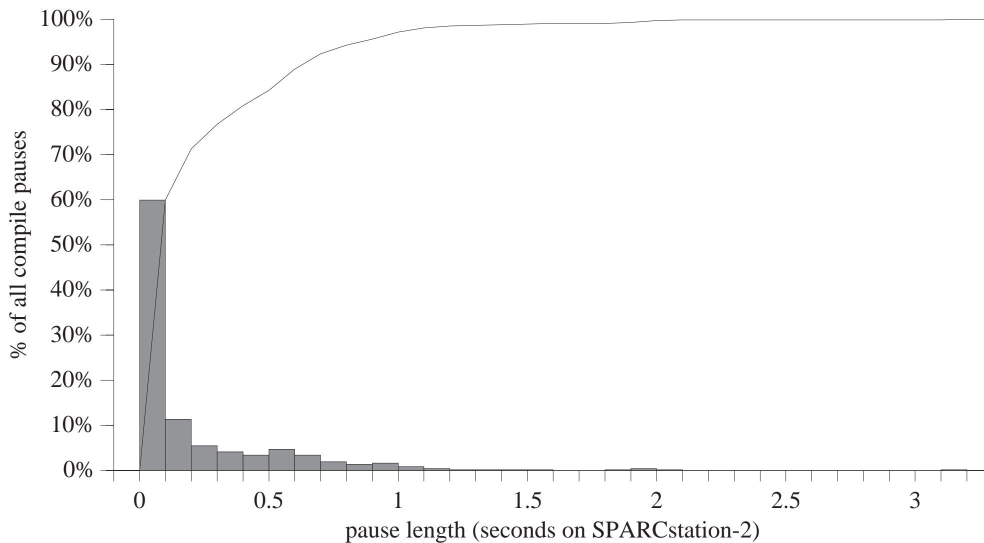


Figure 9-3. Distribution of compile pause length

[†] See Table A-18 in Appendix A for details.

[‡] Since we obtained the data by sampling the system at 100 Hz, very short compilations are either omitted or counted as a pause of 1/100 second.

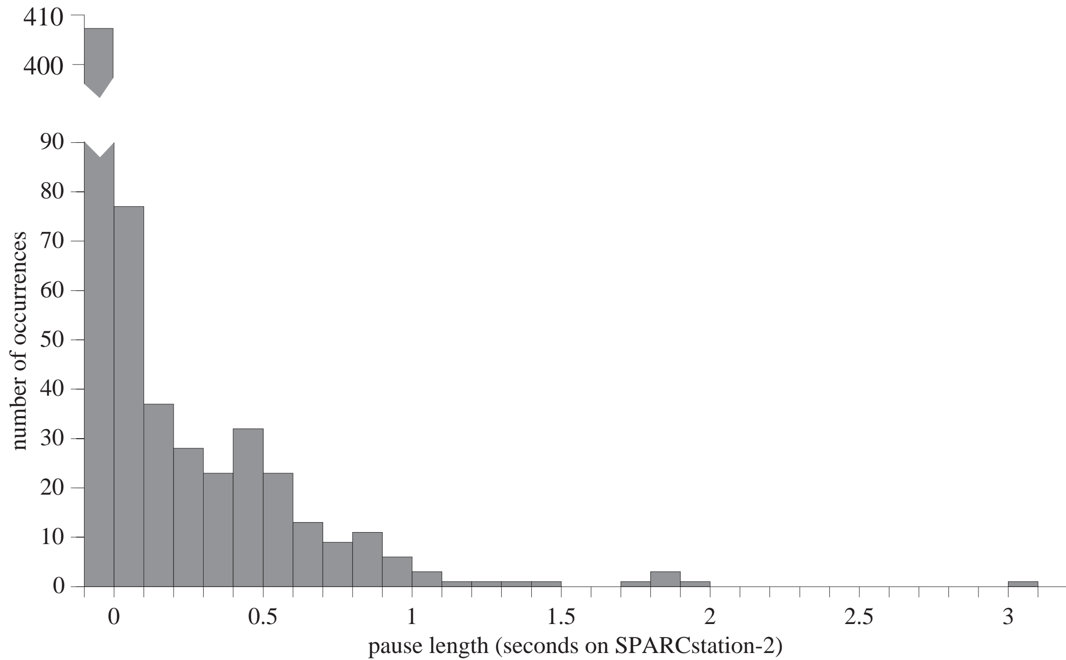


Figure 9-4. Compile pauses during 50-minute interaction

Pause clustering addresses the short-term clustering of compile pauses. However, pauses are also non-uniformly distributed on a larger time scale. Figure 9-5 (on page 108) shows how the same pauses as in Figure 9-4 are distributed over the 50-minute interaction. Each pause is represented as a spike whose height corresponds to the (clustered) pause length; the x-axis shows elapsed time. Note that the x-axis' range is much larger than the y-axis' range (by three orders of magnitude) so that the graph visually exaggerates both the spikes' height and proximity.

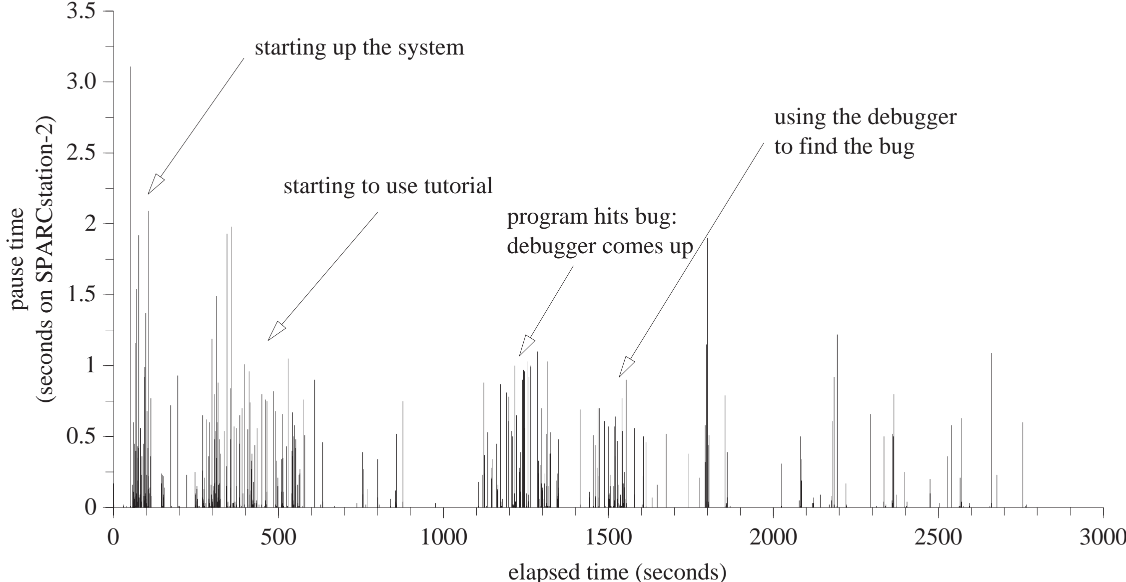


Figure 9-5. Long-term clustering of compilation pauses

During the run, several substantial programs were started from scratch (i.e., without precompiled code). Each of these working set changes caused a flurry of compilations, visible in Figure 9-5 as groups of spikes. The initial group that includes the longest pause corresponds to starting up the user interface; the next group represents the first phase of the

tutorial, where much of the user interface code is exercised for the first time. The last two groups correspond to invoking the debugger after discovering a bug, and inspecting the stack to find the cause of the error.

The system's pause behavior on the SPARCstation-2 seems adequate, especially when considering that this workstation is considered "low-end" today (Summer 1994). Although the interactive behavior isn't as good as that of the (non-optimizing) Deutsch-Schiffman Smalltalk compiler, pauses are much shorter than in traditional programming environments using batch-style compilation.

9.2.2 Pauses on faster systems

The practicality of optimizing compilation in an interactive system is strongly dependent on CPU speed. A system significantly slower than the 20-SPECInt92 SPARCstation-2 would probably make pause times too distracting. That is, our system would have been impractical on the machines commonly in use when the Deutsch-Schiffman Smalltalk compiler was developed, since they were at least an order of magnitude slower.

On the other hand, the system's interactive behavior will improve with faster CPUs. Today's workstations and high-end PCs are already significantly faster than the SPARCstation-2 used for our measurements (see Table 9-1). To investigate the effect of faster CPUs, we re-analyzed our trace with parameters chosen to represent a current-generation workstation or PC (three times faster than a SPARCstation-2) and a future workstation[†] (ten times faster).

System	SPECInt92 (higher is faster)	
	absolute	relative to SS-2
SPARCstation-2	22	1.0
66MHz Pentium PC	65	3.0
80 MHz PowerPC Macintosh	63	2.9
SPARCstation-20/61	89	4.0
1994 high-end workstation	135	6.1
expected in 1995	>200	>9.1

Table 9-1. Speed of workstation and PCs

Figure 9-6 compares the SPARCstation-2 pauses with those on the two simulated systems.[‡] For each pause length, the graph shows the number of pauses exceeding that length. Note that the graph for a faster machine is not just the original graph shifted to the left, because it may combine compilations that were not combined in the slower system. For example, if two groups of compilations are 1 second apart on the SPARCstation-2, they are only 0.33 seconds apart on the "current" workstation and thus must be combined into a single pause (see the rules in Section 9.1). However, the overall impact of this effect is quite small, which confirms our earlier observation that pause clustering is fairly insensitive to the value of the interpause time parameter.

On the current-generation workstation, only 13 pauses exceed 0.4 seconds. These numbers confirm our informal experience of SELF running on current-generation SPARCstation-10 machines: pauses sometimes are still noticeable, but they are rarely distracting. The even faster next-generation workstation will eliminate virtually all noticeable pauses: only 4 pauses will be longer than 0.2 seconds. On such a machine, the current SELF system should feel like an

[†] Workstation vendors are expected to announce 200-SPECint workstations by the end of this year, so that even these "future" workstations should be available in 1995.

[‡] See Table A-18 in Appendix A for details.

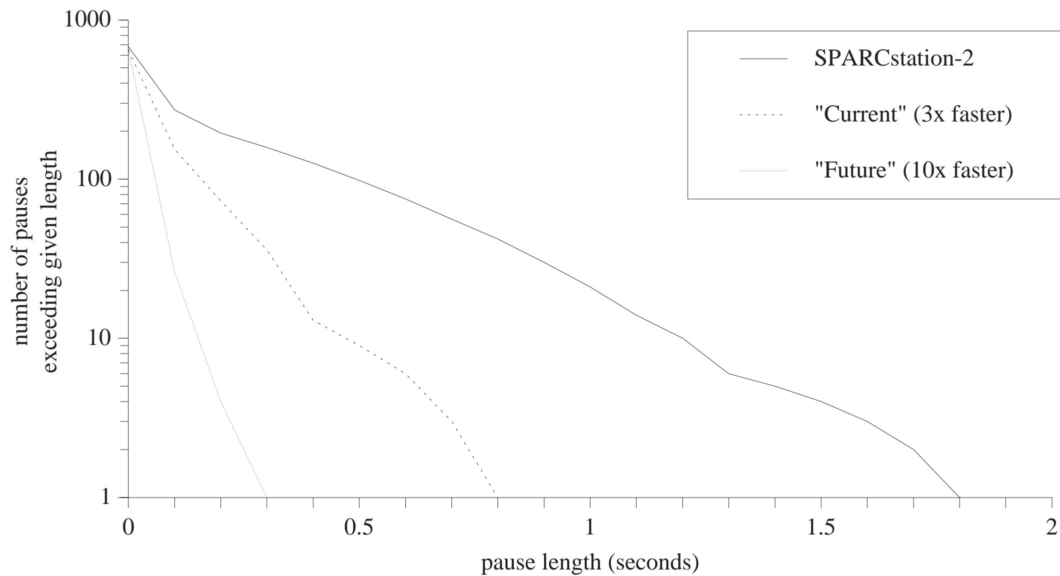


Figure 9-6. Compilation pauses on faster CPUs

"optimizing interpreter." (However, pauses may still not be short enough for real-time animation or video, since the human eye can spot pauses of less than 0.1 seconds in such situations.)

Arguments of the kind "with a faster machine, everything will be better" are often misleading, especially if they assume that everything else (e.g., the problem size or program size) remains constant. We believe that our argument escapes this fallacy because the length of individual compilations does not depend on program size or program input but on the size of a compilation unit, i.e., the size of a method (and any methods inlined into it during optimization). Unless people's programming styles change, the average size of methods will remain the same, and thus individual pauses will become shorter with faster processors. Furthermore, as already noted above, we did not simply divide pause times by the speedup factor but reanalyzed the trace, combining individual pauses into longer pauses by correctly accounting for the shorter interpause times. Larger programs may prolong the time needed for recompilation to settle down (see Section 9.4.1 below), but our experience is that program size does not influence the clustering of individual compilations. In other words, while larger programs may cause more pauses, they do not lengthen the pauses themselves. Therefore, we believe that it is safe to predict the interactiveness of the system as perceived by the user will improve with faster processors, as shown in Figure 9-6.

To summarize, one could characterize the compilation pauses of the SELF-93 system as noticeable but only occasionally distracting on previous-generation systems, sometimes noticeable but almost never distracting on current-generation systems, and virtually unnoticeable on next-generation systems. When combined with adaptive recompilation, an optimizing compiler can indeed coexist with an interactive programming environment.

9.3 Starting new code

A responsive system should be able to quickly start up new (as yet uncompiled) programs or program parts. For example, the first time the user positions the cursor in an editor, the corresponding editor code has to be compiled. Starting without precompiled code is similar to continuing after a programming change, since such a change invalidates previously-compiled code. For example, if the programmer changes some methods related to pop-up menus and then tries to test the change, the corresponding compiled code must be generated first. Thus, by measuring the time needed to complete small program parts (e.g., user interface interactions) without precompiled code, one can characterize the behavior of the system during a typical debugging session where the programmer changes source code and then tests the changed code.

In order to evaluate the benefits of combining a fast non-optimizing compiler with a slower optimizing compiler, we measured the time taken by the execution of common user interactions such as displaying an object or opening an editor. These are typical actions where long compilation pauses are especially distracting. At the beginning of the sequence, the system was started with an empty code cache. Table 9-2 shows the individual interactions of the sequence. For the measurements, the interactions were executed in the sequence shown in the table, with no other activities in-between except for trivial actions such as placing the cursor in the text editor. Thus, although the sequence starts with an empty code cache, interactions may reuse code compiled for previous interactions. Interactions 13 to 15 measure how quickly the system reacts to changes in the definition of methods, using the "worst case" of redefining the integer addition method.

No.	Description	execution time (s on SPARCstation-2)		
		SELF-93	SELF-93-nofeedback	SELF-91
1	start user interface, display initial objects	26.3	42.2	91.9
2	dismiss standard editor window	1.5	4.7	11.5
3	dismiss lobby object	0.5	0.7	0.8
4	show "thisObjectPrints" slot of a point object	0.6	0.7	2.3
5	open editor on slot's comment	2.4	3.2	8.1
6	dismiss editor	0.5	0.4	0.2
7	sprout x coordinate (the integer 3)	2.0	4.4	11.4
8	sprout 3's parent object (traits integer)	2.4	1.2	2.5
9	display "+" slot	1.5	2.6	7.5
10	sprout "+" method	2.9	4.2	11.8
11	dismiss "+" method	1.0	1.5	3.0
12	open editor on "+" method	1.7	2.2	4.6
13	change "+" method (changing the definition of integer addition)	13.9	31.0	82.4
14	reopen editor on "+" method	6.9	12.0	28.2
15	undo the previous change (changing the definition of integer addition back to the original definition)	15.0	30.7	82.0
16	dismiss traits smallInteger object	1.0	4.0	9.8
	geometric mean of ratios relative to SELF-93	1.0	1.6	3.4
	median	1.0	1.6	4.0

Table 9-2. UI interaction sequence

Several points are worth noting. First, SELF-93 usually executes the interactions fastest; on average, it is 3.45 times faster than SELF-91 and 1.6 times faster than SELF-93-nofeedback (geometric means; the medians are 4.0 and 1.6). However, there are fairly large variations: some tests run much faster with SELF-93 (e.g., number 16 is 9.6 and 3.9 times faster, respectively), but a few tests (e.g., number 6) run slower than in the systems without recompilation. Adaptive recompilation introduces a certain variability in running times, slowing some interactions down with recompilations.

Several factors contribute to these results. SELF-93 is usually fastest because the non-optimizing compiler saves compilation time. However, dynamic recompilation introduces a certain variability in running times, slowing down some interactions. This can happen when recompilation is too timid (so that too much time is spent in unoptimized code) or when it is too aggressive (so that some methods are recompiled too early and then have to be recompiled

again). SELF-93-norecomp is faster than SELF-91 because type feedback allowed its design to be kept simpler without compromising the performance of the compiled code.

Rows 13 to 15 show how quickly the system can recover from a massive change: in both cases, a large amount (300 Kbytes) of compiled user interface code needed to be regenerated after the definition of integer addition had changed. Since the integer addition method is small and frequently used, it was inlined into many compiled methods, and all such compiled methods were discarded after the change. Adaptive recompilation allowed the system to recover in less than 15 seconds (recall from Table 7-1 on page 67 that the user interface consists of about 15,000 lines of code, not counting general system code such as collections, lists, etc.). This time included the time to accept (parse) the changed method, dismiss the editor, update the screen, and react to the next mouse click. Compared to SELF-93-norecomp, dynamic optimization buys a speedup of 2 in this case; compared to SELF-91, the speedup is a factor of 5 to 6. Of course, subsequent interactions may also be slower as a result of the change because code needs to be recreated. For example, opening an editor (row 14) takes four to six times longer than before the change (row 12).

With adaptive optimization, small pieces of code are compiled quickly, and thus small changes to a program can be handled quickly. Compared to the previous SELF system, SELF-93 incurs significantly shorter pauses; on average, the above interactions run three to four times faster.

9.4 Performance variations

If a program (or part of a program) is run repeatedly, its performance will vary over time. Initially, it will run more slowly, both because the code is unoptimized and because much compilation and recompilation occurs. With time, performance should stabilize at some asymptotic value. In this section, we investigate how the execution behavior of our benchmark programs changes over time and how different compiler configuration parameters affect this behavior.

Three questions are addressed:

- Does performance stabilize, and how do programs differ in this respect?
- How do configuration parameters influence how quickly performance stabilizes (i.e., the slope of the performance curve)?
- How do the configuration parameters influence the final performance (i.e., the asymptote of the performance curve)?

Table 9-3 shows the main configuration parameters and their function.

Parameter	Function
max. compiled method size	limits the maximum size of a method, and therefore limits the maximum length of a compilation
inlining limits	used in determining which calls to inline (see Section 6.1.2); smaller values reduce individual compilation times (because the individual compiled methods become smaller) and overall compilation time (less inlining usually means less code duplication)
recompilation limit	determines when methods trigger a recompilation (Section 5.3)
half-life time	determines how quickly invocation counts decay over time (Section 5.3)
max. compilation percentage	limits how much of total CPU time compilation may use (as a percentage of a sliding one-second interval); low values limit recompilation during start-up situations

Table 9-3. Configuration parameters impacting compilation pauses in SELF-93

9.4.1 Start-up behavior of large programs

The previous section characterized the pauses caused by (re-)compiling small pieces of code. This section investigates what happens when large programs must be compiled from scratch. To see how performance develops over time, each benchmark was started with an empty code cache and then repeatedly executed 100 times. Although the benchmarks are fairly large, the test runs were kept short, at about 2 seconds for optimized code. Thus, the first few runs are dominated by compilation time since a large body of code is compiled and optimized (Figure 9-7). For example, `Typeinf`'s first run takes more than a minute, whereas the tenth run takes less than three seconds. After a few runs, the compilations die down and execution time becomes stable for all programs except `UI1` which experiences another flurry of compilation around run 15. However, the initial peak is substantial: typically, the first execution runs 20-50 times slower than the 100th run. Why this initial peak in execution time?

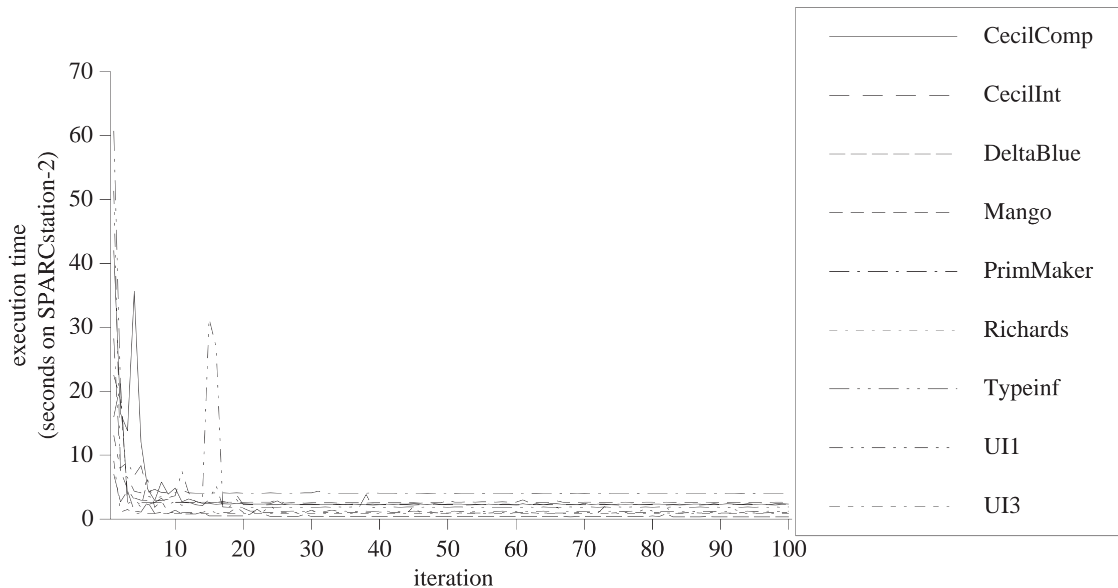


Figure 9-7. Overview of performance development

Figure 9-8 (page 114) breaks down the start-up phase of several benchmarks into compilation and execution. Most of the initial execution time of `UI1` is consumed by non-optimizing compilations and slow-running unoptimized code; in `UI1`, optimizing compilation never dominates execution time. To reduce the initial peak in execution time for `UI1`, the non-optimizing compiler would have to be substantially faster and generate better code. In contrast, optimizing compilation dominates the start-up phase of `Typeinf` and (to a lesser extent) `CecilComp`. `CecilInt` lies somewhere in-between—optimizing compilation consumes a minor portion of the first run but a major portion of the second run. In summary, the reasons for the high initial execution time vary from benchmark to benchmark, and there is no single bottleneck.

The programs' start-up time should correlate with program size: one would expect larger programs to take longer to reach stable performance since more code has to be (re)compiled. Figure 9-9 (page 115) shows the "stabilization time" of the benchmarks, plotted against the programs' sizes. (The stabilization time is the compilation time incurred by a program until it reaches the "knee" of the initial start-up peak.[†]) As expected, some correlation does exist: in general, larger programs take longer to start. However, the correlation is not perfect, nor can it expected to be. For example, a large program that spends most of its time in a small inner loop will quickly reach good performance since only a small portion needs to be optimized.

[†] The knee was determined informally since we were interested in a qualitative picture and not in precise quantitative values.

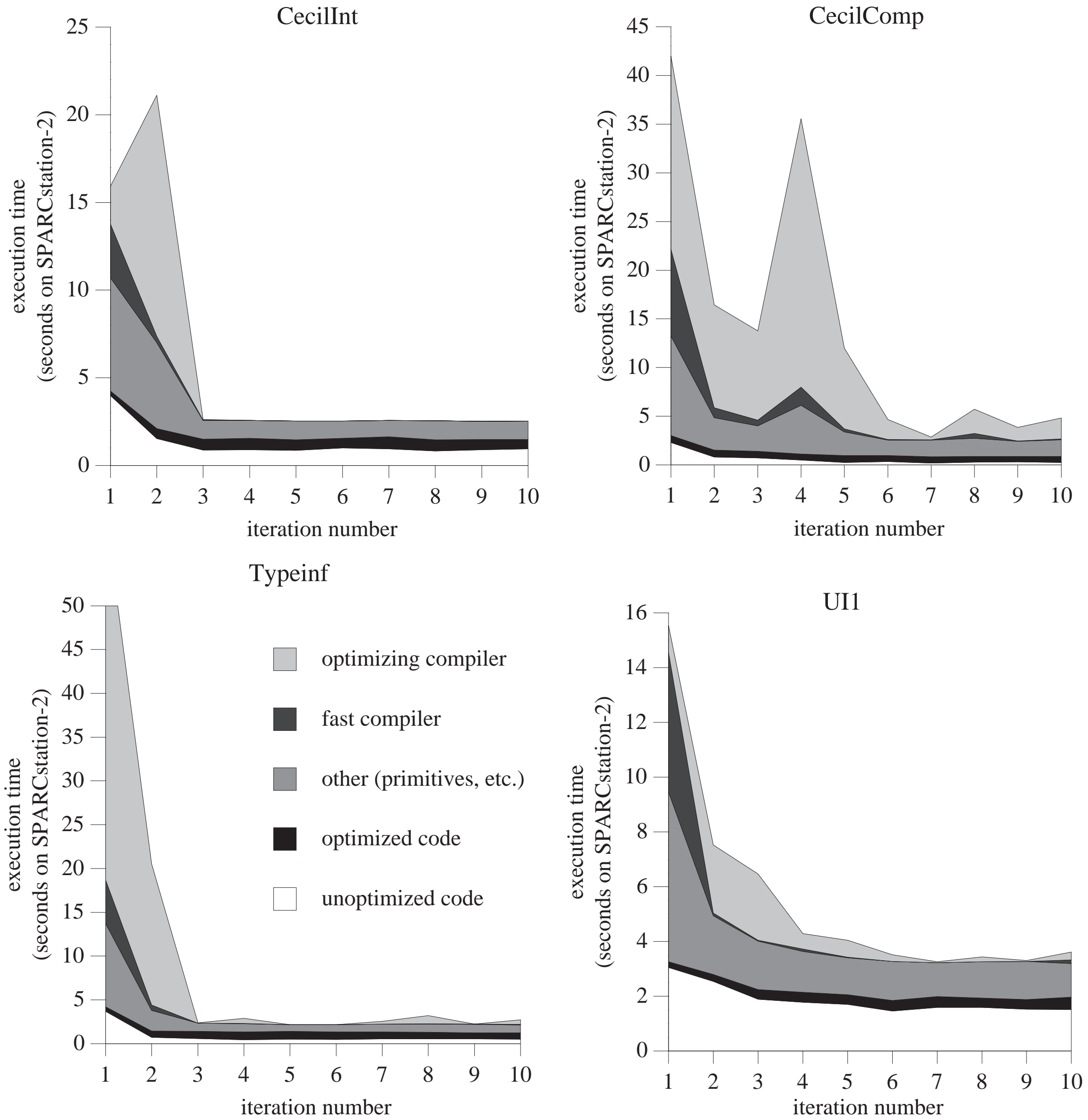


Figure 9-8. Start-up phase of selected benchmarks

If start-up compilation is correlated with program size, we can use the above measurements to characterize the start-up behavior of the system depending on program size and execution time (Table 9-4). If programs are small, so is the

		program size	
		small	big
execution time	short	good	not good
	long	good	good

Table 9-4. Start-up behavior of dynamic compilation

start-up time, and thus the start-up behavior is good for small programs. If programs run for a long time, start-up behavior is good as well, since the initial compilations are hidden by the long execution time. However, if large

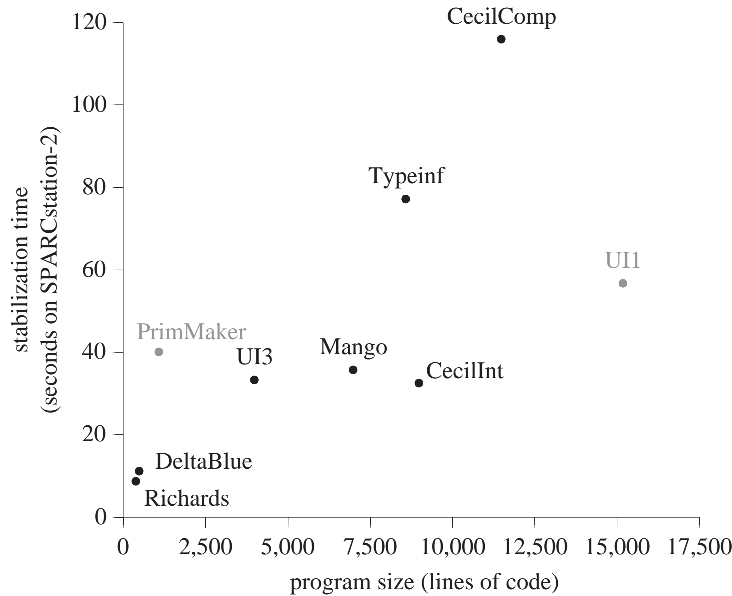


Figure 9-9. Correlation between program size and time to stabilize performance

programs execute only for a short time, the start-up costs of dynamic compilation cannot be hidden in the current SELF system. Our benchmarks all fall in this category because their inputs were chosen to keep execution times short, so that they could be simulated with reasonable effort. In real life, one might expect large programs to run longer, and thus start-up behavior would be better than with the benchmarks.

9.4.2 Performance stability

Even after the initial flurry of compilations has tailed off, programs still experience some performance variations caused by compilations. For the remainder of this section, we will clip the Y axis of the graphs at y = 10 seconds in order to show more detail on the performance variations in later runs. Figure 9-10 shows the development of performance for all benchmarks when run with the standard system. All benchmarks show little variation in execution time after run 20, and many of them converge before run 10. Three benchmarks converge especially quickly to a stable performance level and show no subsequent variations. This group consists of `Richards` and `DeltaBlue` (the two smallest benchmarks), and `Mango`. Because they are well-behaved in all configurations we measured, these benchmarks were excluded in future graphs to reduce the visual clutter.

The remaining group of benchmarks shows some peaks in execution time even after the start-up phase (Figure 9-11), but the variations are usually small. In almost all cases, these peaks correspond to recompilation overhead.† After 10 iterations, only `UI1` and `CecilComp` have not yet reached stable performance; after 20 iterations, all benchmarks have reached stability. The remaining peaks are fairly small, on the order of a second or less; for example, `UI3` shows three peaks of about one second each at iterations 20, 45, and 73.

Overall, performance is fairly stable, except during the start-up phase. Our graphs tends to exaggerate the performance volatility of the system because the individual execution times (i.e., the time taken for a single run) are short, and thus even a small absolute compilation overhead registers as a significant peak. Figure 9-12 shows how much of a difference the measurement interval makes; it presents the same data as Figure 9-7 but groups five runs into one, simulating the effect of longer benchmarks. Had we used longer benchmark runs, performance would seem to be very stable without exception, and the initial slowdown would appear smaller as well. However, we believe that keeping

† To improve the graphs' legibility, we do not separately show execution and compilation time. Some small variations are caused by sources other than compilation, but do not significantly alter the appearance of the graphs.

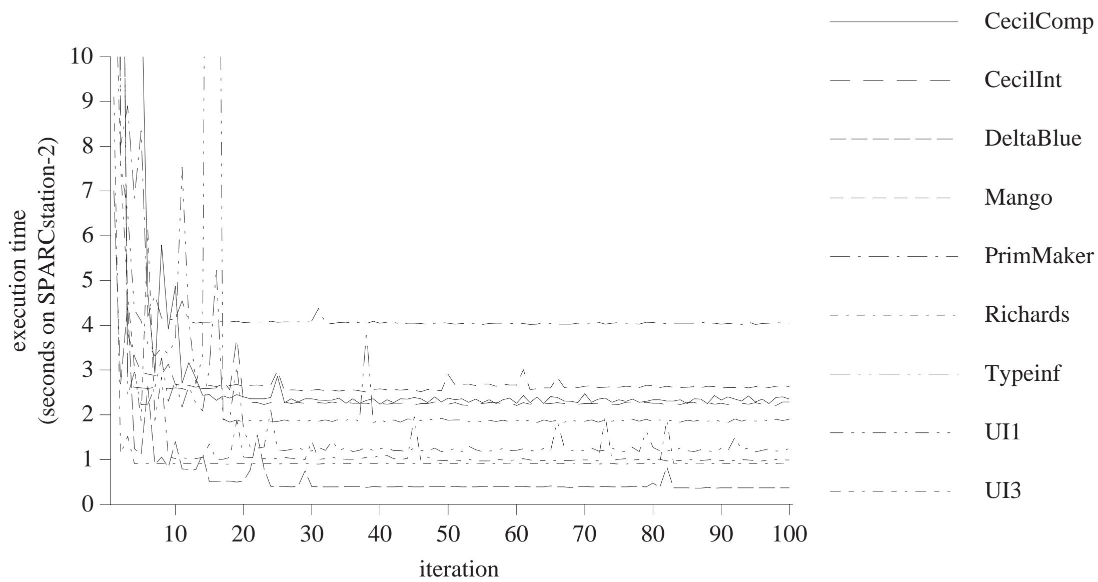


Figure 9-10. Performance variations of SELF-93

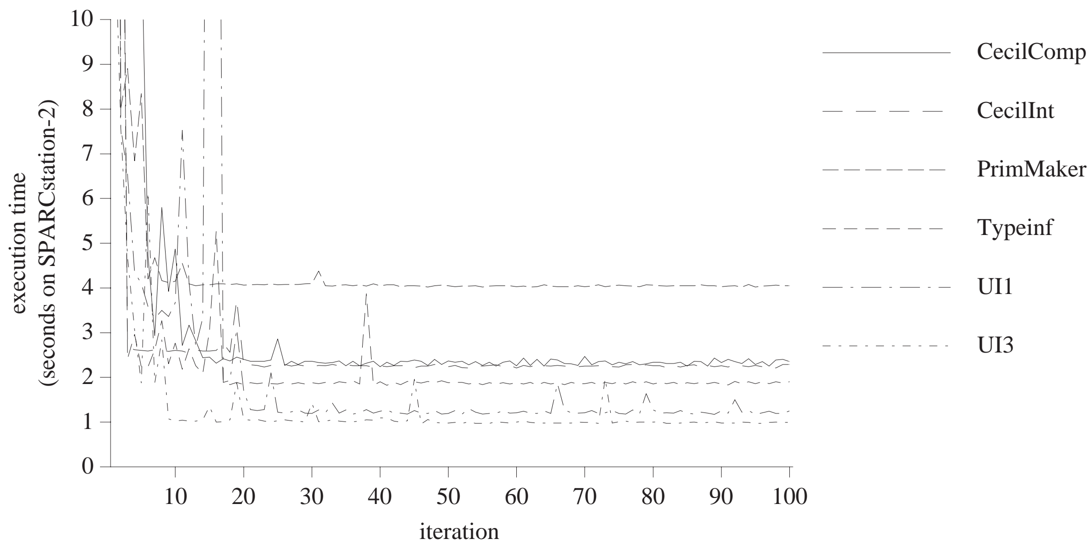


Figure 9-11. Performance variations on subset of benchmarks (SELF-93)

the individual measurement intervals short is the right way to measure variability because it reflects the time taken by typical actions in an interactive system. Intuitively, the “visual height” of the peaks indicates how disturbing the peak would be in interactive use: a small peak of 0.1 seconds is barely noticeable, but a peak of two seconds certainly is.

The measurements shown in this chapter are not entirely reproducible since the exact event sequence varies from run to run. For example, invocation counter decay is implemented by a process that wakes up every four CPU seconds. Since clock interrupts do not arrive reproducibly, invocation counter values (and thus recompilation decisions) may vary from run to run. However, the shape of different runs is usually similar. For the graphs in this section, we repeated each measurement five times, ranked the resulting graphs and used the third-ranked (middle) graph. Figure 9-13 shows the graph ranked worst for the same configuration used in Figure 9-11. Except for `UI1`, the two graphs are very similar. However, `UI1` behaves very differently and has three large recompilations around run 30, 50,

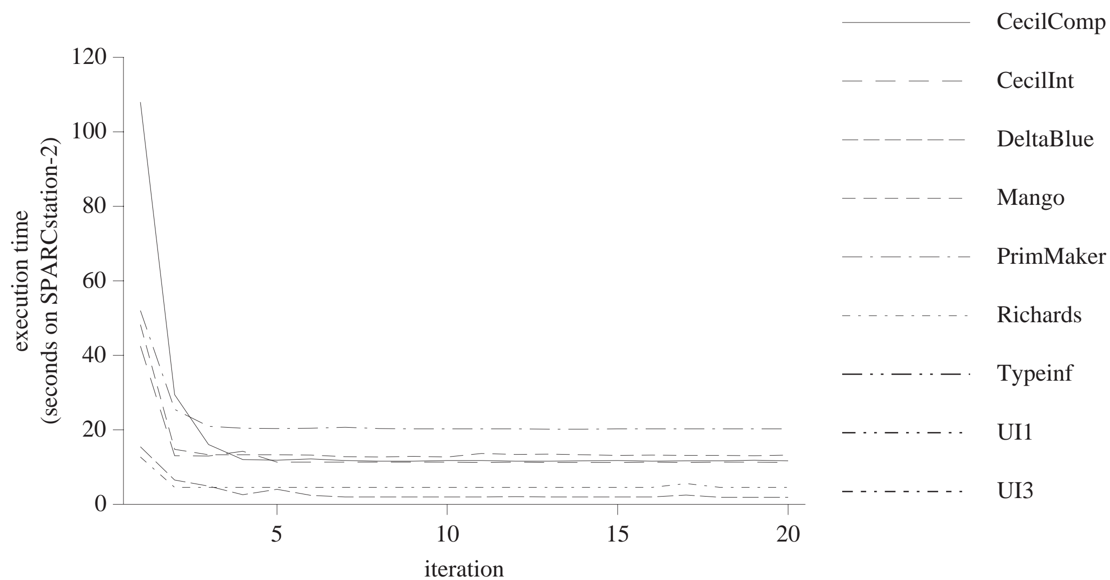


Figure 9-12. Alternate visualization of data in Figure 9-11

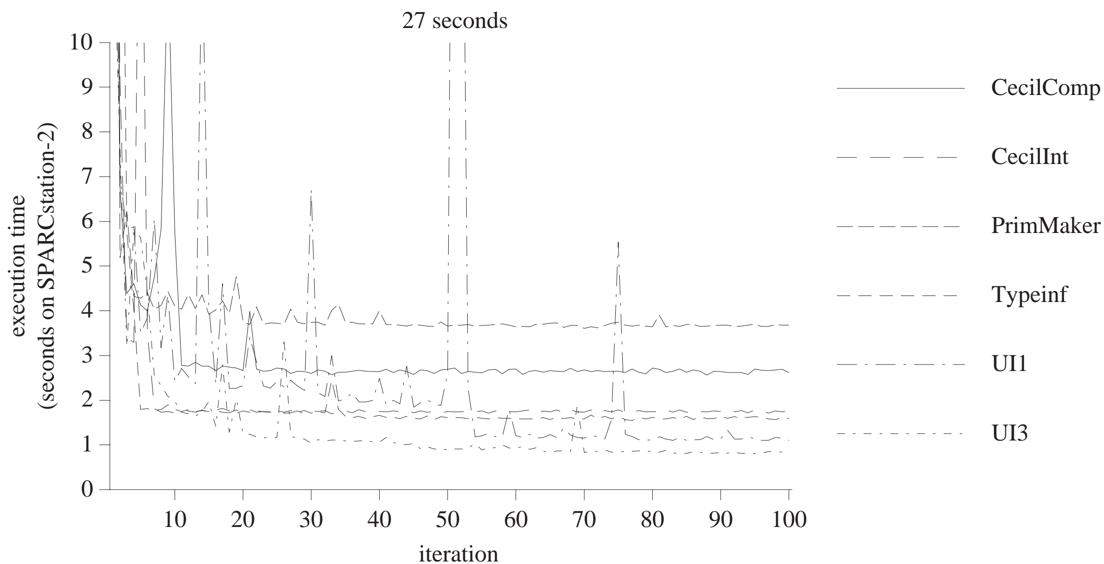


Figure 9-13. Performance variations on subset of benchmarks
(SELF-93, "worst" run)

and 75. (This difference apparently is an extreme case: when we tried to investigate the cause of the difference, we could not reproduce the behavior.)

In summary, large programs initially run more slowly, both because they aren't optimized yet and because much time is spent in compilation. The compilation time spent during this start-up period is roughly proportional to the program size. After two minutes (on a SPARCstation-2), all programs have reached stable performance.

9.4.3 Influence of configuration parameters on performance variation

While searching for a good configuration for our standard system, we experimented with a wide range of parameter values. Although we could not establish hard rules (i.e., rules without exceptions) to predict the influence of parameter changes, we found several trends:

- Invocation count decay significantly reduces variability by reducing the number of recompilations. The half-life time influences variability: the closer it is to infinity (i.e., no decay), the more variable the execution times become. Without any decay, performance never really stabilizes.
- Increasing the inlining limits (leading to more aggressive inlining) tends to produce higher peaks, i.e., longer recompilation pauses. However, the effect is not very pronounced.
- Increasing the recompilation limit (i.e., the invocation count value at which a recompilation is triggered) lengthens the start-up phase because more time is spent in unoptimized code. It does not always reduce performance variability.
- Reducing the percentage of CPU time available for compilation tends to lengthen the start-up periods while somewhat flattening the initial peaks.

Counter decay has the most influence on stability; in particular, the difference between decay and no decay is striking. Figure 9-14 shows the variability of the same system as in Figure 9-11, except that the half-life time is infinity instead

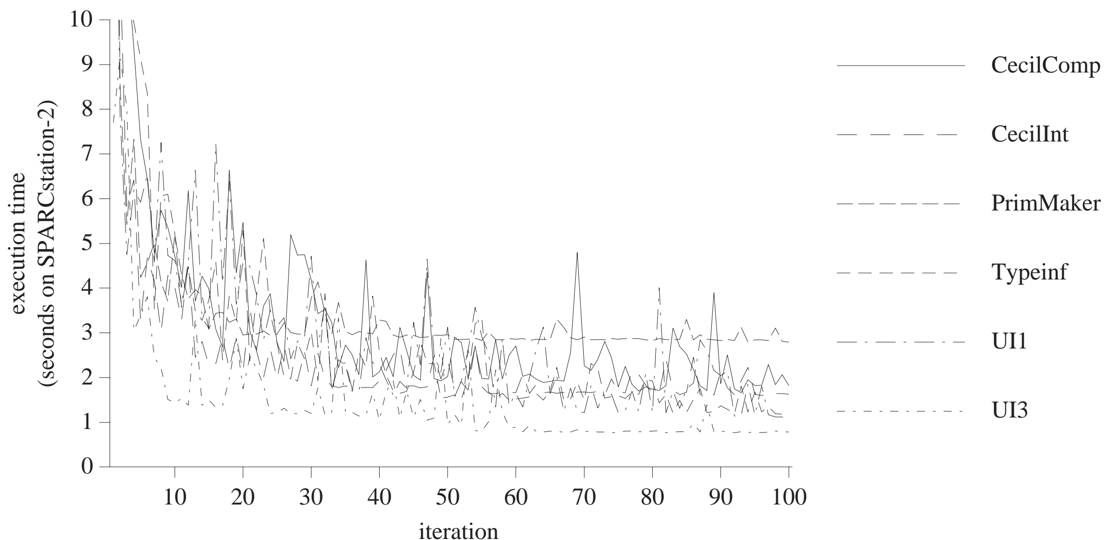


Figure 9-14. Performance variations if invocation counters don't decay

of 15 seconds. All programs show significantly higher variations. Also, several of the programs do not seem to converge towards a stable performance level even after 100 iterations. Intuitively, the reason for this behavior is clear: without decaying invocation counts, every single method will eventually exceed the recompilation threshold, and thus will be optimized. That is, performance only stabilizes when there are few methods left to recompile.

9.4.4 Influence of configuration parameters on final performance

The influence of the configuration parameters on performance is fairly clear-cut: in general, the following changes improve performance:

- increasing the inlining limits (within reason),
- reducing the invocation limit, and

• reducing the decay factor (moving it closer to 1.0).

To measure the impact of the last two factors, we varied the invocation limit and the half-life time and measured the resulting execution time. For each combination of parameters, we chose the best execution time out of 100 repetitions of a benchmark and normalized this time to the best time for that benchmark. That is, the parameter configuration resulting in the best performance for a particular benchmark receives a value of 100%. Thus, a value of 150% for another parameter combination would mean that this combination results in an execution time that is 1.5 times longer than that of the best parameter combination. Times were measured on a SPARCstation-2.

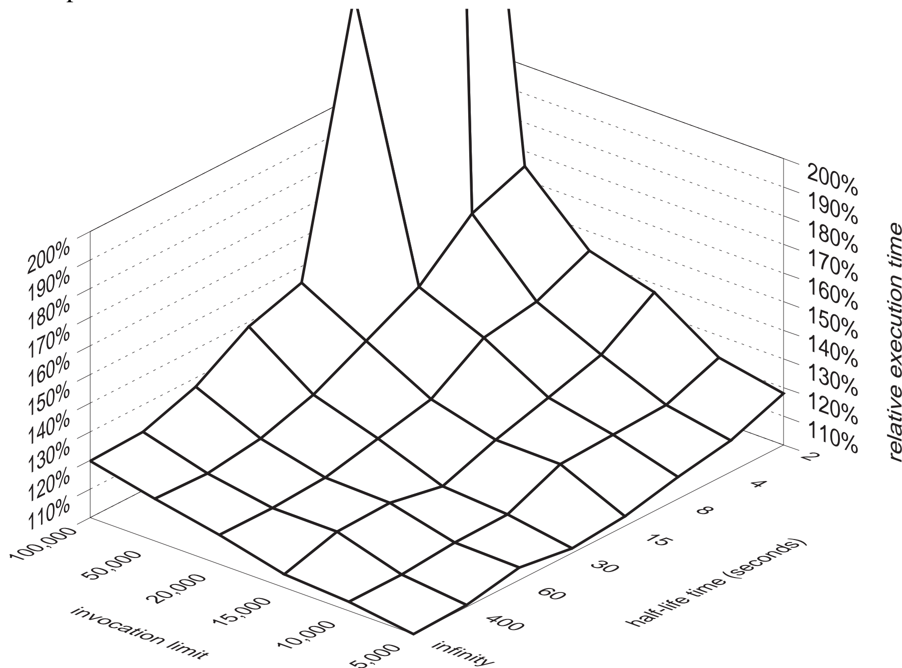


Figure 9-15. Influence of recompilation parameters on performance (overall)

Figure 9-15 shows the resulting performance profile, averaged over all benchmarks (the data were clipped at z = 200%; the true value for the worst parameter combination is over 1100%). Overall, the two parameters behave as expected: increasing the invocation limit and decreasing half-life both increase execution time because a smaller part of the application is optimized since fewer methods are recompiled. However, the performance profile is "bumpy," showing that performance does not vary monotonically as one parameter is varied. The bumps are partly a result of measurement errors (recall that variations caused by cache effects on the SPARCstation-2 can be as high as 15%†) and partly a result of an element of randomness introduced by using timer-based decaying. The latter effect is particularly strong with "bad" parameter combinations (short half-life, high invocation limit). Figure 9-16 shows the performance profile of `CecilComp`, one of the "bumpiest" profiles we measured; it shows strong variations in the area of half-life times or high invocation limits. Since the timer interrupts governing the counter-decaying routine do not always arrive at exactly the same points during the program's execution, a loop may be optimized in one run (e.g., with a half-life time of 4 seconds) but not in another run (with half-life time of 8 seconds) because the counters are always decayed in time before they can trigger a recompilation. This explanation is consistent with the fact that the bumps are not exactly reproducible (i.e., the phenomenon is reproducible, but the exact location and height of the bumps is not).

† We would have liked to measure performance using our simulation, but this was not possible because the simulations would have consumed too much CPU time (the experiment consumed several days of CPU time even without simulation). Furthermore, the simulator currently cannot simulate timer interrupts.

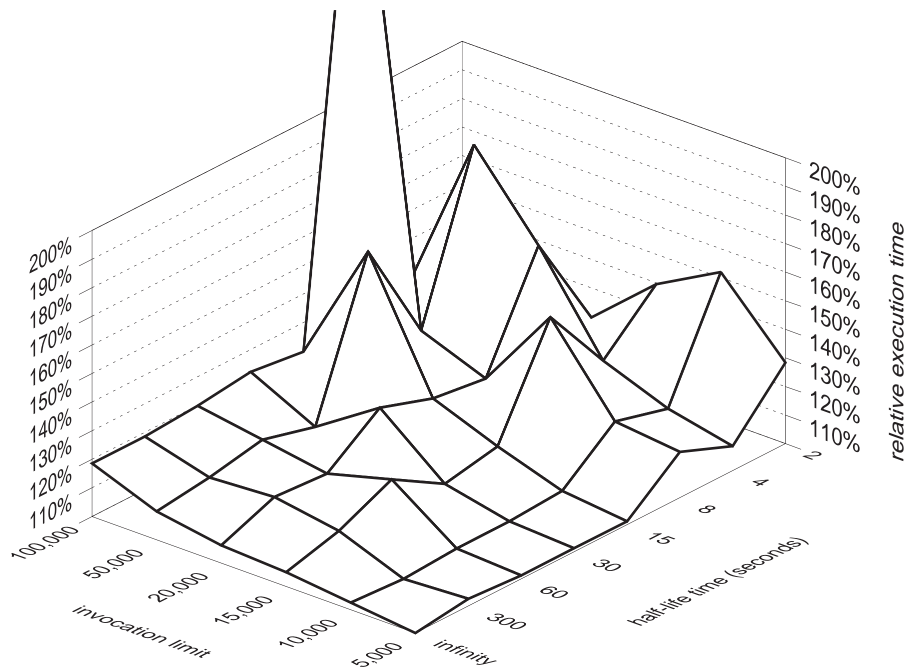


Figure 9-16. Influence of recompilation parameters on performance (`CecilComp`)

In summary, the recompilation parameters influence performance in predictable ways; generally, reducing the invocation limit and increasing counter half-life time improve performance. The performance curve is fairly flat within a range of parameter values, allowing the system to be tuned for the best compromise between interactive responsiveness and performance.

9.5 Compilation speed

This section measures the last aspect related to interactive performance, raw compilation speed. Figures 9-17 and 9-18 show the compilation speed of the SELF-93-nofeedback† and SELF-91 compilers as a function of the total number of byte codes processed per compilation (a byte code corresponds to a source-code token). The data points represent all compilations occurring during one run of the benchmark suite.‡ For example, a compilation that processed a total of 400 byte codes (including the byte codes of all inlined methods) in 300 ms would register as a dot with coordinates x = 400 and y = 300.

For both compilers, compile time is approximately a linear function of the source size as measured by the number of byte codes, with correlation coefficients of 0.98 for SELF-93-nofeedback and 0.96 for SELF-91. Since both compilers' speed is a linear function of source length (with negligible start-up overhead), the relative compilation speed can be expressed as a single number. On average, SELF-93-nofeedback takes 1.1 ms per byte code, and SELF-91 takes 2.6 ms; in other words, SELF-93-nofeedback compiles about 2.5 times faster.†† The faster compilation can be attributed to the simpler compiler front end (which does not have to perform type analysis), and to the less ambitious back end. Thus, in addition to eliminating more message sends, type feedback also speeds up compilation. Compared

† We used SELF-93-nofeedback rather than SELF-93 in order to obtain more data points for the optimizing compiler; otherwise, most methods would have been compiled with the non-optimizing compiler. (Recall that both systems use the same optimizing compiler.)

‡ CecilComp does not execute correctly in SELF-91 (see footnote on page 67) and was not measured for SELF-91.

†† In a paper to be submitted to the Journal of Irreproducible Results, we also observe that compilation speed (2.6 vs. 1.1 ms / byte code) perfectly correlates (r = 1.000000) with the compilers' source code sizes, 26,000 vs. 11,000 lines.

Figure 9-17. Compilation speed of SELF-93-nofeedback

Figure 9-18. Compilation speed of SELF-91

to the nonoptimizing compiler described in Chapter 4, SELF-93-nofeedback is about 5-6 times slower per byte code (see Figure 4-2 on page 25 for details).

As seen in Figure 9-8 on page 114, the start-up phase of some programs is dominated by optimizing compilations, and thus it would be desirable to further improve the compiler's speed. Figure 9-19 shows how compile time is spent in SELF-93-nofeedback. Almost 60% is spent in the front end, and more than half of that time is spent performing message lookups and making inlining decisions. Much of the lookup time could be eliminated with a lookup cache, i.e., a cache mapping lookup keys to slot descriptors. Such a cache has not yet been implemented yet because message lookups currently perform two functions: they find the target slot, and they record the set of slots and objects on which the lookup result depends. The latter is needed to invalidate compiled code after source code changes. Each compiled method includes a dependency set containing the union of the lookup dependency sets of all lookups performed during the compilation. Whenever an object changes, all compiled methods containing that object are discarded. A lookup cache would have to store the dependency sets as well as the lookup results and thus would require a reorganization of the dependency system.

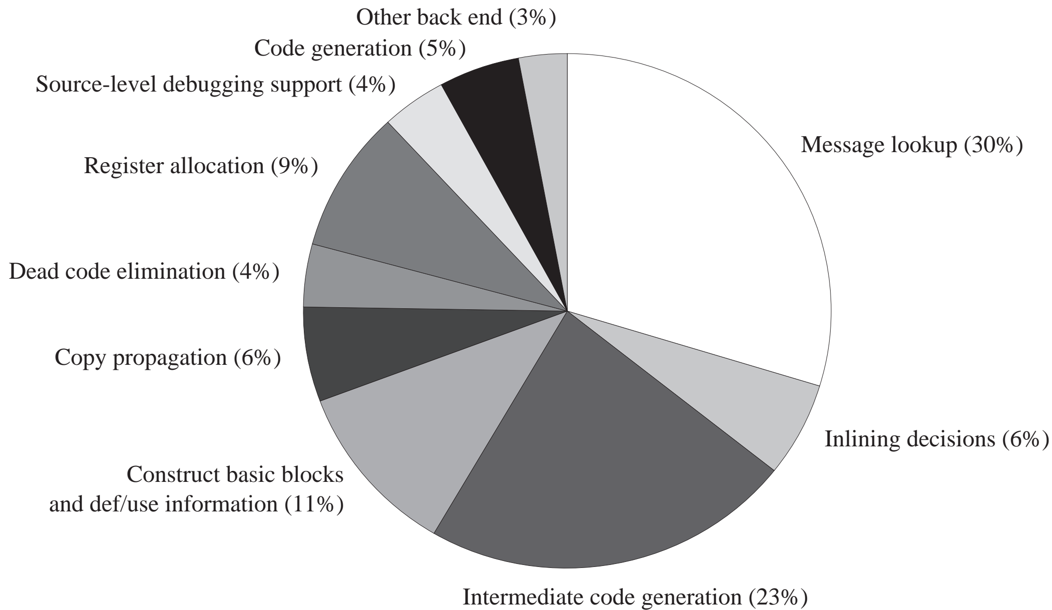


Figure 9-19. Profile of SELF-93-nofeedback compilation

With a lookup cache, the compiler would be about one third faster, and the back end would start to dominate compilation time even though it does not perform very many optimizations. While the compiler could probably be sped up somewhat using standard programming tricks, more significant speedups would require fundamental design changes. An important source of inefficiency is that the compiler has to perform a lot of work for common operations, and that this work is repeated many times. For example, the compiler has to inline several message sends for every `if` control structure in a method, creating many intermediate nodes and pseudo registers. Eventually, most of those nodes are optimized away, but of course this process consumes valuable compile time. A promising approach to reducing compile time would be to partially evaluate the compilation of common message sends like `ifTrue:` or `+`. If it were possible to generate templates of pre-optimized intermediate code for these messages, much compilation effort could be avoided. Unfortunately, implementing such templates involves several nontrivial problems, such as how to parameterize the templates and how to pre-optimize them independently of the code in which they are embedded (for example, optimization of the method implementing a `for` loop depends heavily on knowing the type (integer) and value (positive or negative) of the arguments.†

9.6 Summary

When discussing pause times, it is imperative to measure pauses as they would be experienced by a user. We have defined such a pause metric, pause clustering. Pause clustering combines consecutive short pauses into one longer pause, rather than just counting them as individual pauses. Applying pause clustering to the compilation pauses of the SELF-93 system changes the pause distribution by an order of magnitude, emphasizing the importance of pause clustering. We believe that pause clustering should be used whenever pause length is important, for example, when evaluating incremental garbage collectors.

Like other languages, object-oriented languages need both good runtime performance and good interactive performance. Pure object-oriented languages make this task harder since they need aggressive optimization to run at acceptable speed, thus compromising interactive performance. With adaptive recompilation, a system can provide both good runtime performance and good interactive performance, even for a pure object-oriented language like SELF.

† These problems are similar to the problems encountered by Dean and Chambers in their work towards better inlining decisions based on results of previous compilations [40].

Even on a previous-generation workstation like the SPARCstation-2, fewer than 200 pauses exceeded 200 ms during a 50-minute interaction, and 21 pauses exceeded one second. With faster CPUs, compilation pauses start becoming unnoticeable: on a current-generation workstation, only 13 pauses would exceed 400 ms, and on a next-generation workstation (likely to be available in 1995), no pause would exceed 400 ms, and only four pauses would exceed 200 ms.

In addition to speeding up the execution of programs, type feedback also speeds up compilation. Because type feedback allows the compiler to be simpler, individual compilations become shorter and compilation speed increases; compared to SELF-91, the optimizing SELF-93-nofeedback compiler compiles about 2.5 times faster per source-code unit. Recompilation reduces average individual compile pauses even further because most compilations are performed by the fast, non-optimizing compiler and take only a few milliseconds.

Adaptive recompilation also helps to improve the system's responsiveness to programming changes. For example, it takes less than 15 seconds on a SPARCstation-2 for the SELF user interface to start responding to user events again after the radical change of redefining the integer addition method (which invalidates all compiled code that has inlined integer addition). Compared to the SELF-91 system (which does not use adaptive recompilation), SELF-93 improves responsiveness by a factor of 6 in this case.

Large programs initially run more slowly if started without precompiled code. The time required to reach stable performance correlates well with source length: the longer the program, the longer the initial start-up phase. In general, the recompilation process settles down fairly quickly; only two of the benchmarks do not reach stable performance after running for about 45 seconds on a SPARCstation-2. In the worst case (large benchmarks that execute only for a short time), the current system cannot hide the initial overhead of dynamic compilation; however, if large programs execute longer (e.g., for a minute), the relative overhead diminishes. A system could also save precompiled optimized code on disk and load it on demand to reduce the start-up overhead. Thus, while pauses may still be significant when making far-reaching changes to large programs during program development, they need not be a problem in "production mode" where precompiled code could be used.

In the future, it should be possible to hide compilation pauses even better than the current SELF-93 system does. With dynamic recompilation, optimization is "optional" in the sense that the optimized code is not needed immediately. Thus, if the system decides that a certain method should be optimized, the actual optimizing compilation could be deferred if desired. For example, the system could enter the optimization requests into a queue and process them during the user's "think pauses" (similar to opportunistic garbage collection [138]). Alternatively, optimizing compilations could be performed in parallel with program execution on a multiprocessor machine.

With the increasing speed of hardware, interpreters or nonoptimizing dynamic compilers (as used in current Smalltalk systems) may no longer represent the optimal compromise between performance and responsiveness. Today, it is practical to push for better performance—thus widening the applicability of such systems—without forfeiting responsiveness. Adaptive recompilation exploits the speed of contemporary hardware to reconcile run-time performance with compile-time responsiveness. We hope that this work will encourage implementors of object-oriented languages to explore this new region in the design space, resulting in new high-performance exploratory programming environments for object-oriented languages.

10. Debugging optimized code

SELF's pure message-based model of computation requires extensive optimization to achieve good performance. But an interactive programming environment also demands rapid turnaround time and complete source-level debugging. Merely providing correct execution semantics despite optimization is not enough: for optimal programmer productivity, the system should provide the illusion of directly executing the programmer's source code. In other words, the system should provide interpreter semantics at compiled-code speed, combining global optimization with *expected behavior* [144].

Most existing systems do not support the debugging of optimized code. Programs can either be optimized for full speed, or they can be compiled without optimizations for full source-level debugging. Recently, techniques have been developed that strive to make it possible to debug optimized code [65, 143, 37]. However, none of these systems is able to provide full source-level debugging. For example, it generally is not possible to obtain the values of all source-level variables, to single-step through the program, or to change the value of a variable. Optimization is given priority over debugging, and consequently these systems provide only restricted forms of debugging.

In contrast, we were not willing to compromise source-level debugging: to maximize programmer productivity, the system must provide full source-level debugging at all times. Thus, we chose to restrict *some* optimizations to preserve debugging, but tried to keep the performance impact of debugging to be as small as possible. Unfortunately, if the source-level state of the program must be recoverable at every point in the program, i.e., at virtually every instruction boundary (to support single-stepping), existing techniques [2, 3, 65, 143, 37] severely restrict the optimizations that could be performed, effectively disabling many common optimizations.

We solve this dilemma by relaxing the debugging requirements placed on optimized code. Rather than requiring the source-level state to be available at every point in the program, optimized code need only be able to recover this state at relatively few points (essentially, at non-inlined calls). To the programmer, this restriction is invisible because code is deoptimized if needed to provide finer-grain debugging. Programs can always be inspected or single-stepped as if they were completely unoptimized. To the best of our knowledge, the system we describe in this chapter is the first practical system providing expected behavior in the presence of many global optimizations; compared to previous techniques, our use of *dynamic deoptimization* and *interrupt points* permits us to place fewer restrictions on the kind of optimizations that can be performed while still preserving expected behavior.

10.1 Optimization vs. debugging

The code transformations performed by global optimization make it hard to debug optimized code at the source level. Because optimizations delete, change, or rearrange parts of the original program, they become visible to the user who tries to debug the optimized program. This section presents some of the problems that must be solved to provide source-level debugging of optimized code.

10.1.1 Displaying the stack

Optimizations such as inlining, register allocation, constant propagation, and copy propagation create methods whose activation records have no direct correspondence to the source-level activations. For example, a single physical stack

frame may contain several source-level activations because message sends have been inlined. Variables may be in different locations at different times, and some variables may not have runtime locations at all.

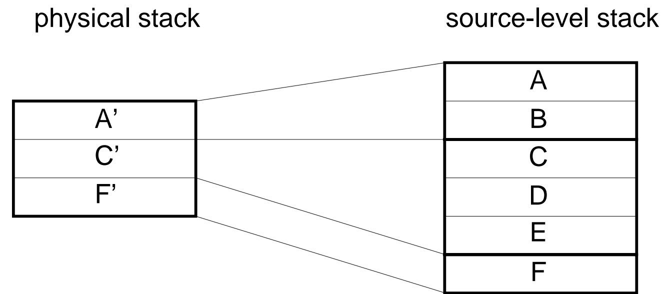


Figure 10-1. Displaying the stack

The example in Figure 10-1 shows the effects of inlining. The physical stack contains three activations A', C', and F'. In contrast, the source-level-stack contains additional activations which were inlined by the compiler. For example, the activation B was inlined into A', and so B does not appear in the physical stack trace.

10.1.2 Single-stepping

To single-step, the debugger has to find and execute the machine instructions belonging to the next source operation. Optimizations such as code motion or instruction scheduling make this a hard problem: the instructions for one statement may be interspersed with those of neighboring statements, and statements may have been reordered to execute out of source-level order. In contrast, single-stepping is simple with unoptimized code since the code for a statement is contiguous.

10.1.3 Changing the value of a variable

Consider the following code fragment:

```
i := 3;                         i := 3;
j := 4;      optimization       j := 4;
k := i + j;  ----------->       k := 7;
```

Since the expression `i + j` is a compile-time constant, the compiler has eliminated its computation from the generated code. But what if the program is suspended just before the assignment to `k` and the programmer changes `j` to be 10? Execution of the optimized code cannot be resumed since it would produce an unexpected value for `k`. With unoptimized code, of course, there would be no problem since the addition would still be performed by the compiled code.

10.1.4 Changing a procedure

A similar problem arises when an inlined procedure is changed during debugging. Suppose that the program is suspended just before executing the inlined copy of function `f` when the programmer changes `f` because she has found a bug. Obviously, execution cannot simply continue since `f`'s old definition is hardwired into its caller. On the other hand, it would be easy to provide expected behavior with unoptimized code: `f`'s definition could simply be replaced, and the subsequent call to `f` would execute the correct code.

10.2 Deoptimization

None of these problems would exist with unoptimized code. If optimized code could be converted to unoptimized code on demand, programs could be debugged easily while still running at full speed most of the time. SELF's debugging system is based on such a transformation. Compiled code exists in one of two states:

- *Optimized code* can be suspended only at relatively widely-spaced interrupt points. At every interrupt point, the source-level state can be reconstructed.
- *Unoptimized code* can be suspended at any arbitrary source-level operation and thus supports all debugging operations (such as single-stepping).

Section 10.2.1 explains the data structures used to recover the source-level state from the optimized program state. Sections 10.2.2 and 10.2.3 describe how optimized code can be transformed into unoptimized code on demand, and Section 10.2.4 discusses how interrupt points lessen the impact of debugging on optimization.

10.2.1 Recovering the unoptimized state

To display a source-level stack trace and to perform deoptimization, the system needs to reconstruct the source-level state from the optimized machine-level state. To support this reconstruction, the SELF compiler generates *scope descriptors* [23] for each scope contained in a compiled method, i.e., for the initial source method and all methods inlined within it.† A scope descriptor specifies the scope's place in the virtual call tree of the physical stack frame and records the locations or values of its arguments and locals (see Figure 10-2). The compiler also describes the location or value of each subexpression within the compiled method. This information is needed to reconstruct the stack of evaluated expressions that are waiting to be consumed by later message sends.

```
struct ScopeDesc {
    oop method;                          // pointer to the method object
    ScopeDesc* caller;                   // scope into which this scope was inlined (if any)
    int posWithinCaller;                 // source position within caller
    ScopeDesc* enclosingScope;           // lexically enclosing scope (if any)
    NameDesc args[];                     // descriptors for receiver and arguments
    NameDesc locals[];                   // descriptors for locals
    NameDesc expressionStack[];          // descriptors for all subexpressions
};

struct NameDesc {
    enum { const, loc } tag;             // compile-time constant or run-time value
    union {
        oop value;                       // constant value
        Location location;               // run-time location
    };
};
```

Figure 10-2. Pseudo-code declarations for scope data structures

To find the correct scope for a given physical program counter, the debugger needs to know the *virtual program counter* (source position), i.e., the pair of a scope description and a source position within that scope. Therefore, the debugging information generated with each compiled method also includes a mapping between physical and virtual program counters.

With the help of this information, the debugger can hide the effects of inlining, splitting, register allocation, constant propagation, and constant folding from the user. For example, if the compiler eliminates a variable because its value is a compile-time constant, the variable's descriptor would contain that constant. A straightforward extension of the descriptor structure could be used to handle variables whose values can be computed from other values (such as eliminated induction variables).

† The scope descriptor mechanism was originally developed by Chambers, Ungar, and Lee [23] and was recently reimplemented by Lars Bak to reduce memory requirements.

Figure 10-3 shows a method suspended at two different times. When the method is suspended at time t_1, the physical PC is 28 and the corresponding source position is line 5 of method B. A stack trace would therefore display B being called by A, hiding the fact that B has been inlined into A by the compiler. Similarly, at time t_2 the source-level view would show D being called by C being called by A, displaying three virtual stack frames instead of the single physical stack frame. To display a complete stack trace, this process is simply repeated for each physical stack frame.

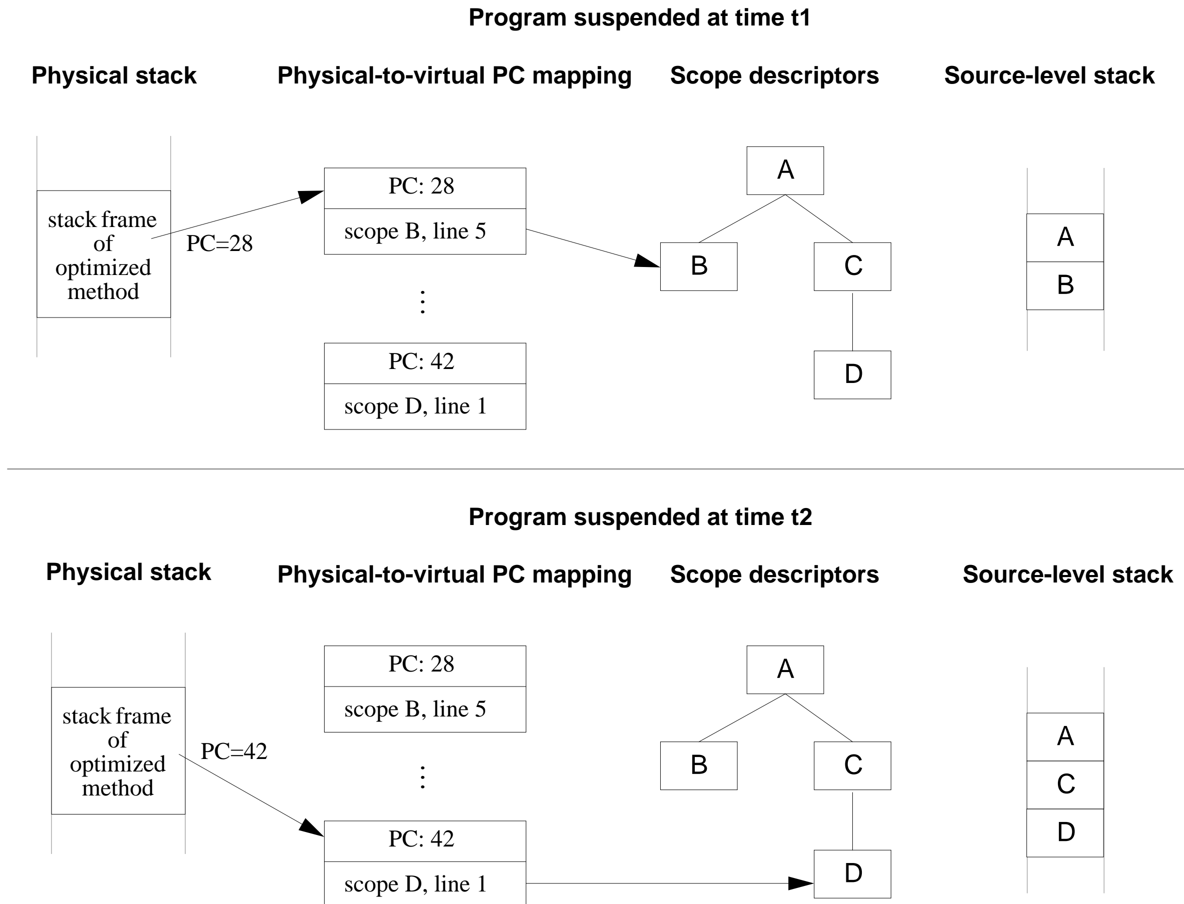


Figure 10-3. Recovering the source-level state

10.2.2 The transformation function

Whenever needed for debugging, the system transforms an optimized method into one or more equivalent unoptimized methods. For the moment, we assume that only the topmost stack activation needs to be transformed so that stack frames can easily be removed or added; Section 10.2.3 explains how to remove this restriction. The deoptimizing transformation can then be performed as follows:

1. Save the contents of the physical activation (stack frame) which is to be transformed, and remove it from the runtime stack.

2. Using the mechanisms described in the previous section, determine the source-level (*virtual*) activations contained in the physical activation, the values of their locals, and their virtual PC.

3. For each virtual activation, create a new compiled method and a corresponding physical activation. To simplify the transformation function and subsequent debugging activities, the new methods (the *target* methods) are completely unoptimized: every message send corresponds to a call, and no optimizations such as constant folding or common subexpression elimination are performed.

4. For each virtual activation, find the new physical PC in the corresponding compiled method. Since the target method is unoptimized, there will be exactly one physical PC for the given virtual PC. (This would not necessarily be the case if the target methods were optimized.) Initialize the stack frames created in the previous step by filling in the return PC and other fields needed by the runtime system, such as the frame pointer.

5. For each virtual activation, copy the values of all parameters, locals, and expression stack entries from the optimized to the unoptimized activation. Since the unoptimized method is a straightforward one-to-one translation of the source method, all variables will be mapped to locations in the target activation, and thus all copied values will have an unambiguous destination. (This would not necessarily be the case if the target methods were optimized.) Furthermore, since the target method is unoptimized, it does not contain any hidden state which would need to be initialized (such as the value of a common subexpression). Thus, together with step 4, the new stack frames are completely initialized for all virtual activations, and the transformation is complete.

Figure 10-4 illustrates the process. The transformation expands an optimized stack frame containing three virtual activations into a sequence of three unoptimized stack frames, thus creating a one-to-one correspondence between virtual and physical frames.

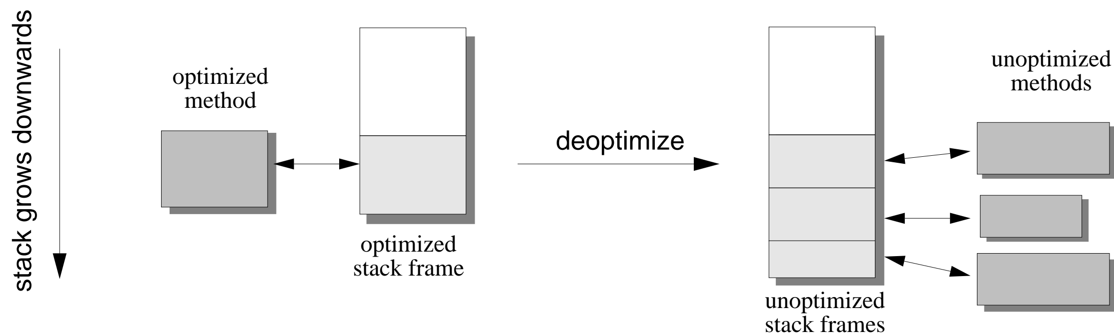


Figure 10-4. Transforming an optimized stack frame into unoptimized form

Only those parts of the program which are actively being debugged (e.g., by stepping through them) need to be transformed. These parts will be the only parts of the program running unoptimized code; all other parts can run at full speed. No transformations are necessary just to inspect the program state, as described in Section 10.2.1.

10.2.3 Lazy deoptimization

But how can a stack frame be deoptimized when it is in the *middle* of the stack, where new stack frames cannot be inserted easily? To solve this problem, our current implementation always transforms stack frames *lazily*: deoptimiza-

tion is deferred until control is about to return into the frame (see Figure 10-5). For example, if the virtual activation

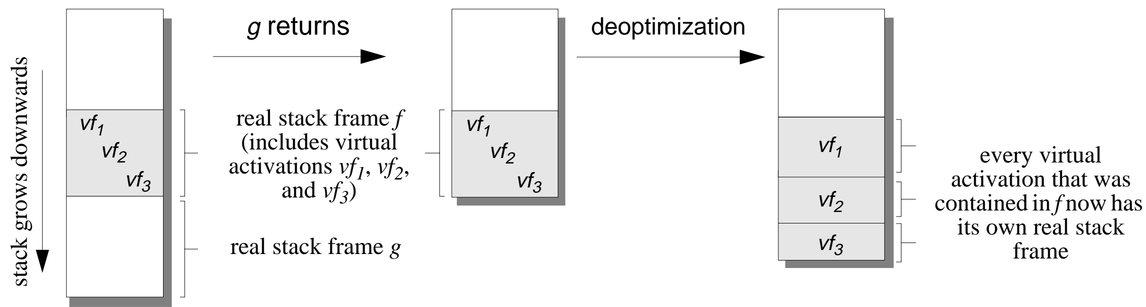


Figure 10-5. Lazy deoptimization of stack frames

vf_2 to be deoptimized is inlined in frame *f* in the middle of the stack, *f* is not immediately deoptimized. Instead, the return address of *g* (the stack frame called by *f)* is changed to point to a routine which will transform *f* when *g* returns. At that point, *f* is the topmost frame and is deoptimized. Transforming only the most recent activation simplifies the transformation process because no other stack frames need to be adjusted even if deoptimization causes the stack frames to grow in size.

Lazy deoptimization can simplify a system considerably, but it may also restrict the debugging functionality unless additional steps are taken. The SELF system currently does not allow the contents of local variables to be changed during debugging because a variable might not have a runtime location. In order to create a runtime location for the variable, it might be necessary to transform an activation in the middle of the stack, which the system currently cannot do. However, this is not a fundamental problem; for example, the transformed stack frames could be heap-allocated as described in [44]. An even simpler solution would be to always allocate stack locations for eliminated variables. These locations would be unused during normal program execution but would spring into life when the programmer manually changed the value of an eliminated variable. Since the compiled code depended on the old (supposedly constant) value, it would be invalidated as if the programmer had changed the method's source code (see Section 10.3).

10.2.4 Interrupt points

If optimized programs could be interrupted at any instruction boundary, debugging optimized code would be hard, since the source-level state would have to be recoverable at every single point in the program. To ease the restrictions this would impose on optimization, an optimized SELF program can be interrupted only at certain *interrupt points*†
where its state is guaranteed to be consistent. Notification of any asynchronous event occurring between two interrupt points is delayed until the next interrupt point is reached. Currently, the SELF system defines two kinds of interrupt points: method prologues (including some process control primitives) and the end of loop bodies ("backward branches"). This definition implies that the maximum interrupt latency is bounded by the length of the longest code sequence containing neither a call nor a loop end, typically only a few dozen instructions. Because the latency is so short, the use of interrupt points is not noticed by the programmer. (If only sends were interrupt points, loops without calls could not be interrupted.) Backward branches only need to check for interrupts if the loop contains an execution path without calls; otherwise, each loop iteration performs at least one call and thus already checks for interrupts.

Interrupt points need to cover all possible points where a program could be suspended; that is, they also need to handle synchronous events such as arithmetic overflow. In SELF, all possible runtime errors are interrupt points because all primitives are safe: if the requested operation cannot be performed, the primitive calls a user-defined error handler which usually invokes the debugger.

† Interrupt points have been used in other systems before; see Section 10.7 for a discussion of the Deutsch-Schiffman Smalltalk-80 system. Also, Hennessy [65] used a similar mechanism to define *stopping points* between each statement, as did Zellweger [144].

Once an optimized program is suspended, the current activation can be deoptimized if necessary to carry out debugging requests. In an unoptimized method, every source point is an interrupt point, and the program can therefore stop at any point.

Since the debugger can be invoked only at interrupt points, debugging information need be generated only for those points. This optimization reduces the space used by the debugging information, but more importantly it allows extensive optimizations between interrupt points. Essentially, the compiler may perform any optimization whose effects either do not reach an interrupt point or can be undone at that point. For example, the compiler can reuse a dead variable's register as long as there are no subsequent interrupt points within the variable's scope. The more widely-spaced interrupt points are, the fewer restrictions source-level debugging imposes on optimization.

Interrupt points also lessen the impact of garbage collection on compiler optimization. Garbage collections can only occur at interrupt points, and so the compiler can generate code between interrupt points that temporarily violates the invariants needed by the garbage collector [31].

10.3 Updating active methods

During debugging, a programmer might not only change the value of a variable but also the definition of a method. To invalidate the compiled code affected by such a change, the SELF system maintains *dependency links* [23] between compiled code and the objects representing source code methods. For example, if a compiled method contains inlined copies of a method that was changed, the compiled method is discarded.

However, if a compiled method containing an inlined copy of a changed method is active (i.e., has at least one activation), it cannot simply be discarded. Instead, the compiled method must be replaced by a new compiled method before execution can continue. Fortunately, deoptimization can be used for this purpose. After the active compiled method has been deoptimized, it will no longer contain any inlined methods. When its execution continues, all subsequent calls to the changed method will correctly invoke the new definition.

If the changed method itself is currently active, updating its activation is hard. Fortunately, in SELF we don't have to solve this problem because in SELF's language model, activations are created by cloning the method object. Once created, the clone is independent from its original, so changes to the original do not affect the clone.

Lazy transformation elegantly solves the problem of invalidated compiled methods in the middle of the stack: we simply wait until the invalid method is on top of the stack, then transform it. Lazy transformation is desirable in an interactive system since it spreads out the repair effort over time, avoiding distracting pauses. Furthermore, it handles sequences of changes well, for example when reading in a file containing new definitions for a group of related objects. With eager transformation, every new definition would cause all affected compiled methods to be recompiled, and many methods would be recompiled several times since they are likely to be affected by several of the changes. With lazy transformation, these compiled methods will be invalidated repeatedly (which is no problem since invalidation is very cheap) but will be transformed only once.

In conclusion, with our debugging mechanisms it is almost trivial to support changing running programs. Our current implementation consists of a few hundred lines of C++ code on top of the previously described debugging functionality and the code maintaining the dependencies.

10.4 Common debugging operations

This section describes the debugging operations implemented in the SELF system and outlines possible implementations of additional operations. With deoptimization, it is relatively easy to implement common debugging operations such as *single-step* and *finish* because these operations are simple to perform in unoptimized code, and deoptimization can supply unoptimized code for every program piece on demand. In contrast, neither *single-step* nor *finish* could generally be provided by previous systems for debugging optimized code [144, 37].

10.4.1 Single-step

Because every source point has an interrupt point associated with it in a deoptimized method, the implementation of single-stepping becomes trivial. The system deoptimizes the current activation and restarts the process with the interrupt flag already set. The process will relinquish control upon reaching the next interrupt point, i.e., after executing a single step.

10.4.2 Finish

The *finish* operation continues program execution until the selected activation returns. It is implemented by changing the return address of the selected activation's stack frame to a special routine that will suspend execution when the activation returns. Thus, the program is not slowed down during the *finish* operation because it can run optimized code.

If the selected activation does not have its own physical stack frame (because it was inlined into another method), its stack frame is deoptimized using lazy deoptimization. In this case, the program can still run optimized code most of the time; only at the very end (when lazy deoptimization is performed) does it run unoptimized code.

10.4.3 Next

The *next* operation (also called "step over") executes the next source operation without stepping into calls. That is, the program will stop after the next source operation has completed. *Next* can be synthesized by performing a *single-step*, possibly followed by a *finish* (if the operation was a call). Consequently, *next* is implemented by a few lines of SELF code in our system.

10.4.4 Breakpoints and watchpoints

SELF currently supports breakpoints through source transformation: the programmer inserts a breakpoint by simply inserting a send of `halt` into the source method (`halt` explicitly invokes the debugger). To implement breakpoints without explicit changes by the programmer, the debugger could perform this source transformation transparently.

Watchpoints ("stop when the value of this variable changes") are also easy to provide because SELF is a pure object-oriented language, and all accesses are performed through message sends (at least conceptually; the compiler will usually optimize away such sends). To monitor all accesses to an object's instance variable `x`, we can rename the variable to `private_x` and install two new methods `x` and `x:` which monitor accesses and assignments, respectively, and return or change `private_x`. (Remember that SELF's prototype-based model allows us to change a single object without affecting others, so we can monitor specific objects easily.) The dependency system will invalidate all code that inlined the old definition of `x` or `x:` (i.e., that directly accessed or changed `x`).

Instead of requiring the programmer to explicitly make these changes, the system could perform them transparently to support breakpoints and watchpoints. In the general case, some code would need to be recompiled to accommodate the new interrupt point(s), but the system does not need any additional basic mechanisms to support this functionality.

10.5 Discussion

In this section, we discuss some of the strengths and weaknesses of our approach and assess its generality.

10.5.1 Benefits

Our debugging technique has several important advantages. First, it is simple: the current implementation of the transformation process consists of less than 400 lines of C++ code on top of the code implementing the debugging information described in Section 10.2.1. Second, it allows a loose coupling between debugger and compiler—neither has to know very much about the other. Third, it places no additional restrictions beyond those described in section 2

on the kind of optimizations which can be performed by the compiler. Thus, many common optimizations such as inlining, loop unrolling, common subexpression elimination, and instruction scheduling can be used between interrupt points without affecting debuggability. Finally, our method is well suited for an interactive system since it is incremental: usually, at most one stack frame needs to be converted as a result of a user command.

10.5.2 Current limitations

The use of unoptimized code during debugging introduces a performance problem when the user decides to continue execution. Execution should proceed at full speed, but some of the stack frames may be unoptimized. However, this problem usually is not severe: only a few frames are running unoptimized code, and the unoptimized code will be discarded as soon as these frames return. All other parts of the system can run at full speed. Methods containing loops could still pose a problem since they could remain on the stack in unoptimized form indefinitely. However, such frames are automatically recompiled with optimization by the recompilation system described in Chapter 5.

Certain optimizations cause problems that cannot be handled by our system. Since it must always provide full source-level debugging, the SELF compiler does not perform such optimizations. In general, dead stores cannot be eliminated, and so the registers of dead variables cannot be reused without spilling the variable to memory first. Similarly, some other optimizations (e.g., code motion or induction variable elimination) may sometimes not be performed if the debugger's recovery techniques are not powerful enough to hide the optimizations' effects at one or more interrupt points. The extent of this problems depends on the particular recovery techniques used (e.g., [2, 3, 65, 143, 37]) and on the average distance between interrupt points. All these optimizations can be performed, however, if there is no interrupt point within the affected variable's scope (see Section 10.2.4). Finally, the SELF compiler does not perform tail recursion elimination or tail call elimination because they cannot be supported transparently: in general, it is not possible to reconstruct the stack frames eliminated by these optimizations.

10.5.3 Generality

The debugging approach presented here is not specific to SELF and could be exploited in other languages as well. Our system appears to require runtime compilation for deoptimization, but systems without runtime compilation could include an unoptimized copy of every procedure in an executable or dynamically link these in as needed.

For pointer-safe languages like Lisp, Smalltalk, or a pointer-safe subset of C++, our approach seems directly applicable. We cannot estimate the performance impact of source-level debugging in other systems since it depends on the characteristics of the particular language, compiler, and on the techniques used to recover values at interrupt points. In general, however, a system using interrupt points and deoptimization should allow for faster code than any other system providing full source-level debugging since deoptimization allows the optimized code to support only a subset of all logically required interrupt points. In pointer-unsafe languages[†] such as C which allow pointer errors, interrupt points might be more closely spaced since the debugger could potentially be invoked at every load or store where the compiler could not prove that no address fault would occur. But even if interrupt points caused by unsafe loads or stores were indeed very frequent, our approach would still allow at least as many optimizations as other approaches for source-level debugging.

Pointers into the stack require special care during deoptimization if the locations of such pointers are unknown. In this case, the address of a stack variable potentially referenced by a pointer may not be changed. However, this problem could probably be solved at the expense of some stack space by requiring the layout of optimized and unoptimized stack frames to be identical.

[†] True source-level debugging of unsafe languages is something of an oxymoron: since programs can overwrite arbitrary memory regions, they can always produce behavior which cannot be explained at the language (source) level. For example, if an integer is erroneously stored into the location of a floating-point variable, the resulting behavior cannot be explained without referring to the particular integer and floating-point representations used by the system.

10.6 Implementation cost

Providing full source-level debugging in the presence of an optimizing compiler does not come for free. In this section, we examine the impact of our techniques on responsiveness, runtime performance, and memory usage.

10.6.1 Impact on responsiveness

Neither the deoptimization process nor the use of interrupt points are perceptible to users. The compiler typically creates the unoptimized methods in less than a millisecond on a SPARCstation-2, and thus the pauses introduced by dynamic deoptimization are negligible. Interrupt points increase the latency for user and system interrupts by only a few microseconds because an interrupt point is usually reached within a few dozen instructions after the runtime system has set the interrupt flag. In summary, providing full source-level debugging in the SELF system has not reduced its responsiveness.

10.6.2 Impact on runtime performance

Ideally, the performance impact of full source-level debugging could be measured by completely disabling it and re-measuring the system. However, this is not possible because source-level debugging was a fundamental design goal of the SELF system. Disabling debugging support would require a major redesign of the compiler and runtime system if any better performance is to be achieved. Furthermore, the garbage collector already imposes some constraints on the optimizer, such as the requirement that live registers may not contain derived pointers (pointers into the middle of objects). In many cases, the optimizations inhibited by garbage collection are very similar to those inhibited by debugging requirements, such as dead store elimination and some forms of common subexpression elimination [31]. Thus, it would be difficult to separate the impact of garbage collection on optimization from the impact of full source-level debugging. While we could not measure the full performance impact of our debugging scheme, inspection of the generated code indicated no obvious debugging-related inefficiencies.

However, we have measured some effects of source-level debugging in the SELF system. To determine the impact of debugger-visible names, the SELF-91 compiler† was changed to release registers allocated to dead variables even if they were visible at an interrupt point. The performance improvement with the changed compiler was insignificant (less than 2%) for a wide range of programs [21]. That is, the extension of variable lifetimes needed to support debugging seems to incur virtually no cost in our system. (One reason for this might be that SELF methods are typically very short, so that few variables are unused in significant portions of their scope.)

The system currently detects interrupts by testing a special register; each test takes two cycles on a SPARC. This polling slows down typical programs by about 4%; some numerical programs with very tight loops are slowed down by up to 13% [21]. With a more complicated runtime system using conditional traps, the overhead could be reduced to one cycle per check, and loop unrolling could further reduce the problem for tight loops. Alternatively, the system could employ a non-polling scheme where the interrupt handler would patch the code of the currently executing procedure to cause a process switch at the next interrupt point.

To summarize, the exact performance impact of source-level debugging in the SELF-93 system is hard to determine. However, based on the data points above, we believe that source-level debugging slows down typical programs by less than 10%.

10.6.3 Memory usage

Table 10-1 shows the memory usage of the various parts of compiled methods relative to the space used by the machine instructions. For example, the relocation information needed for garbage collection is about 17% the size of

† This study was performed before SELF-93 was fully functional. Thus, we used SELF-91 [21] for the measurements. Since this compiler has a better back end, the results are probably more indicative of the true performance impact of source-level debugging than if we had used SELF-93 which performs fewer optimizations.

the actual machine code. The data were obtained from running all benchmark programs in sequence and represent 3.3 Mbytes of compiler-generated data. Since unoptimized and optimized methods have different characteristics, data

Category		SELF-93	SELF-93 (optimized only)
Machine instructions	actual machine instructions and control information	1.00	1.00
Method headers		0.33	0.11
Dependency links	to invalidate code after programming changes	0.45	0.55
Scope descriptors and physical-to-virtual PC mapping	to recreate the source-level state of optimized code and to recompile methods	0.45	0.76
Relocation information for GC		0.17	0.17

Table 10-1. Space cost of debugging information (relative to instructions)

for the optimizing compilations only is shown in the second data column. Thus, the first data column represents the space distribution of the configuration actually used in practice, whereas the second data column shows the characteristics of pure optimized code.

The space consumption can be split into three main groups. The first group contains the method headers and the machine instructions; together, these represent all the information needed to actually execute programs. The second group contains the dependency links needed to invalidate compiled code after programming changes (see Section 10.3). The third group contains all debugging-related information: the scope descriptors and the PC mapping (see Section 10.2.1) and relocation information for the garbage collector (to update object pointers contained in the debugging information). This includes all information needed to recompile methods but not the source code itself.[†]

The space consumed by debugging information varies with the degree of optimization. Optimized methods show a higher relative space overhead than unoptimized methods because the debugging information for an inlined method is typically larger than the inline-expanded code. Therefore, the debugging information grows faster with more aggressive inlining than the compiled code.

The total space overhead for debugging is reasonable. In the standard system, debugging information is less than half the size of the instructions themselves; in the system that optimizes everything, the overhead is 76%. In other words, adding the debugging information increases space usage (excluding the dependencies) by a factor of between 1.5 and 1.8.

In order to be conservative, we have left out the space used by the method headers even though they would also be needed in a system without debugging. (The headers contain the lookup key and various control information.) If we include the headers, debugging increases space usage by a factor of between 1.3 and 1.7.

The cost of supporting changes to running programs is slightly smaller; the dependency information occupies between 0.45 and 0.55 times the size of the instructions. (Since the current representation of the dependencies contains significant redundancies, an alternate implementation could probably reduce the space usage significantly.)

As a rough comparison, when compiling the SELF virtual machine, GNU C++ (using GNU-specific pragmas) generates an executable of 11.3 Mbytes for a text size of 2.0 MBytes.[‡] Thus, the C++ debugging information is 5.7 times

[†] This grouping is a slight simplification. For example, the compiler occasionally generates instructions just to support debugging. Also, a small portion of the relocation information can be attributed to the code rather than the debugging information. However, these simplifications do not significantly distort the numbers presented here.

larger than the actual code, whereas the total overhead of the current SELF system is less than a factor of 2.5 even including the support for programming changes and for garbage collection.† While this comparison should be taken with a grain of salt, it indicates that despite the increased functionality, the space overhead for the data structures supporting source-level debugging is probably not higher than in other systems.

10.7 Related work

The Smalltalk-80 system described by Deutsch and Schiffman [44] pioneered the use of dynamic compilation and interrupt points. To hide the effects of compilation to native code, compiled methods included a mapping from compiled code to source position. Activations normally were created on the stack for runtime efficiency but were converted on demand to the full-fledged activation objects required by the language definition, and converted back when needed for execution. As in our system, interrupts were delayed until the next call or backward branch. Since the compiler performed no global optimizations, the system could provide expected behavior without deoptimization.

Zurawski and Johnson [146] describe a model for a debugger (developed concurrently with this work) which closely resembles ours, using "inspection points" and dynamic deoptimization to provide expected behavior for optimized code. However, the system does not use lazy conversion. Furthermore, their definition of inspection points allows asynchronous events such as user interrupts to be delayed arbitrarily. Some of their ideas were implemented for the Typed Smalltalk compiler [79], but the system apparently could only run very small programs and was not used in practice, unlike our system which is in daily use.

Most of the other work on debugging optimized code is orthogonal to our work since it addresses the problem of recovering a source-level value at a given point in the program. In contrast, our work addresses *where* source-level values need to be recovered, not *how* they are recovered. Thus, using interrupt points and deoptimization amplifies the benefits of techniques aimed at recovering source-level values. Most of the techniques discussed below could be combined with our work to support even more optimizations.

Hennessy [65] addresses the problem of recovering the values of variables in the presence of selected local and global code reordering optimizations. His algorithms can detect when a variable has an incorrect value (in terms of the source program) and can sometimes reconstruct the source-level value. Hennessy used stopping points that are similar to interrupt points: the debugger is invoked only at stopping points, i.e., at the beginning of a statement. Code sequences are restricted so that no stores are completed unless the entire statement runs to completion. Thus, if an error occurs during execution of the statement, the debugger can display the program state as if the program were stopped at the beginning of the statement.

Adl-Tabatabai et al. [2, 3] investigate the problem of detecting and recovering the values of source-level variables in the presence of instruction scheduling. Other recovery mechanisms are described by Coutant et al. [37] and by Schlaeppi and Warren [114].

Zellweger [143, 144] describes an interactive source-level debugger for Cedar which handles two optimizations, procedure inlining and cross-jumping, to provide expected behavior in most cases. While her techniques can always recover the source-level values of variables, they cannot hide certain code location problems; for example, single-stepping through optimized code would be difficult if the compiler performed optimizations not considered by Zellweger, such as instruction scheduling or dead code elimination. Since the SELF system can switch to unoptimized code, it is able to avoid these problems.

LOIPE [52] uses transparent incremental recompilation for debugging purposes. For example, when the user sets a breakpoint in some procedure, this procedure is converted to unoptimized form to make debugging easier. However,

‡ Unlike in the SELF system, where the debugging information is always present in virtual memory, the Unix debugging information only consumes memory when the program is actually being debugged.

† GNU C++ allows the source-level debugging of optimized code but offers only restricted functionality. Many optimizations are not transparent to the programmer. The compiler version used for the measurements was g++ 2.4.5.

LOIPE cannot perform such transformations on active procedures. Thus, if the program is suspended in an optimized procedure, it is generally not possible to set a breakpoint in this procedure or to continue execution by single-stepping. To mitigate this problem, users were able to specify the amount of optimization to be performed (possibly impacting debuggability) and the amount of debugging transparency needed (possibly affecting code quality). As far as we know, most of the support for optimized code in LOIPE was not actually implemented.

Tolmach and Appel [127] describe a debugger for ML where the compiler always performs optimizations, but where the program is automatically annotated with debugging statements before compilation. To debug an optimized program, the programmer has to manually recompile and re-execute the program. Like unoptimized programs, annotated programs run significantly slower than fully optimized programs.

10.8 Conclusions

The SELF system increases programmer productivity by providing full source-level debugging in the presence of many global optimizations such as constant folding, common subexpression elimination, dead code elimination, procedure inlining, code motion, and instruction scheduling. Other systems would have to severely restrict optimization in order to achieve full source-level debugging, because debugging operations such as single-stepping or breakpoints require the source-level state to be recoverable at virtually every instruction boundary. In contrast, our debugging system gives the optimizing compiler more freedom since it does not require full debugging at every instruction.

Full source-level debugging need not preclude global optimization. Two techniques make is possible to combine the two: *lazy deoptimization* and *interrupt points*. The optimizations performed by the compiler are hidden from the programmer by deoptimizing code whenever necessary. Deoptimization supports single-stepping, running a method to completion, replacing an inlined method, and other operations, but only affects the procedure activations which are actively being debugged; all other code runs at full speed. Debugging information is only needed at relatively widely-spaced interrupt points, so that the compiler can perform extensive optimizations between interrupt points without affecting debuggability. The debugging system also allows optimized programs to be changed while they are running (so execution can be resumed after a change). In particular, the system guarantees that no invocation will use the old definition of a method after a change, even if this method was inlined into other compiled methods.

To the best of our knowledge, the SELF system is the first system to provide full source-level debugging (in particular, single-stepping) of optimized programs. As far as we know, it is also the first system to provide source-level semantics when changing running programs. Together with adaptive optimization, the debugging system makes it possible to integrate an optimizing compiler into an exploratory programming environment.

11. Conclusions

“Later is better”: late binding can be optimized with late compilation. Late binding is a problem for traditional compilers because they do not have enough information to generate efficient code from the source code alone. By delaying optimization until the necessary information is available, a compiler can generate better code. For an interactive system like SELF, delaying optimization has the additional benefit of reducing compilation pauses by confining costly optimization to the time-critical parts of the program.

Polymorphic inline caches have proven to be an attractive mechanism for collecting the type information needed for type feedback. In addition to recording receiver types for every send, they actually speed up dynamic dispatch. In other words, programs “instrumented” for type feedback run faster than programs without instrumentation. Thus, polymorphic inline caches are an attractive implementation of dynamic dispatch even for systems that do not use type feedback.

When combined, type feedback and adaptive recompilation improve both better runtime performance and interactive behavior. On average, programs run 1.7 times faster when compiled with type feedback and execute 3.6 times fewer calls. Overall, the execution characteristics of optimized SELF programs are surprisingly close to those of the SPECint89 benchmark suite. Our measurements indicate that object-oriented programs can execute efficiently without special hardware support, given the right compiler technology.

Compared to the previous SELF compiler, the new compiler is significantly simpler (11,000 vs. 26,000 lines of code) and compiles about 2.5 times faster per source-code unit. Adaptive recompilation improves the apparent speed of compilation even more since most compilations are performed by the fast non-optimizing compiler, and optimizing recompilations are spread out over time. In the SELF-93 system, adaptive recompilation manages to hide optimization to a large extent on a current-generation workstation; with even faster machines, compilation pauses should become virtually unnoticeable.

The most visible effect of dynamic (re)compilation is that programs initially run more slowly when starting from scratch (i.e., without any compiled code). Our experiments indicate that a 10,000-line program can be expected to get up to speed in about half a minute on a current-generation workstation. Similarly, a program change temporarily slows down execution proportionally to the impact it has on existing code. Usually, changes are small and local, so that execution can proceed with virtually no delay.

Finally, dynamic deoptimization allows us to reconcile global optimization with source-level debugging. The debugging information provided by the compiler only needs to support reading the source-level program state at points where the program may be interrupted, not at every instruction boundary. Between two interrupt points, any optimization is allowed. Debugging requests that go beyond reading the current state are handled by transparently deoptimizing compiled code and performing the request on the unoptimized code. Thus, the programmer is not forced to chose between speed and source-level debugging: programs can be debugged at any time, even if they are optimized.

Dynamic compilation is often regarded as complicated and hard to implement. We hope that our work shows that dynamic compilation can actually make the implementor’s life easier. Once the underlying mechanisms are in place, new functionality based on dynamic compilation can be added relatively easily. For example, both the source-level debugging system (Chapter 10) and the handling of exceptional situations (Section 6.1.4) could be implemented with relatively little effort (in about 3,000 and 100 lines of code, respectively) because the system already supported dynamic compilation. We believe that dynamic compilation can be an attractive choice for interactive development environments even for languages where the compiler has more static information than in SELF (i.e., where the compiler could generate reasonable code for debugging purposes without using runtime information).

11.1 Applicability

Although our implementation relies on dynamic compilation, most of the techniques described in this thesis do not require it. Type feedback would be straightforward to integrate into a conventional compiling system, similar to other profile-based optimizations. Source-level debugging using deoptimized code could be implemented by keeping precompiled unoptimized code in a separate file. Polymorphic inline caches only require a simple stub generator, not full-fledged dynamic compilation. Only dynamic reoptimization is—by its very nature—specific to dynamic compilation.

Neither are the techniques described in this thesis specific to the SELF language. The debugging system is largely language-independent, although it works best in pointer-safe languages since interrupt points can be spaced further apart in these languages. Type feedback could optimize late binding in any language; besides object-oriented languages, non-object-oriented languages that make heavy use of late binding (e.g., APL with its generic operators, or Lisp with its generic arithmetic) could profit from this optimization. Finally, any system using dynamic compilation might profit from adaptive recompilation to improve performance or to reduce compile pauses.

11.2 Future work

One of the possible areas for improvement is compilation overhead. Speeding up compilation will reduce compilation pauses and start-up times. Increasing processor speeds will help to reduce pauses—today's workstations already outperform the system used for our measurements by a factor of three. However, with increasing processor speed comes increasing problem size, and thus the start-up times of large applications may not become smaller since applications may become larger. Thus, software techniques further reducing compile pauses remain an interesting area of future research. As discussed in Section 4.5, the non-optimizing compiler could possibly be replaced by an interpreter with little loss in execution speed. Speeding up the optimizing compiler is harder, although introducing a compile-time lookup cache would improve compilation speed by about 25% with relatively little effort. Possible approaches resulting in more significant speedups include improving the inlining system and precomputing the intermediate code of frequently-inlined methods (Section 9.5). On multiprocessor workstations, recompilations could be executed on a separate processor, thus reducing recompilation pauses to virtually zero. Alternatively, optimization could be performed opportunistically during the user's "think pauses."

In the current system, the parameters of the recompilation system are defined statically. As a result, their values sometimes represent a compromise between interactiveness and ultimate performance. For example, if the half-life time for the invocation counters is low, ultimate performance is usually better but compile pauses are worse. A better system would extend the adaptiveness of compilation by varying the half-life parameter according to the situation. During start-up situations (characterized by many compilations), the half-life time would be short to prevent premature optimization. Later, when the initial flurry of compilations has died down, the half-life time could be increased dynamically to allow more of the application to be optimized. Similarly, a longer-term recompilation mechanism could optimize program parts that execute for a long time; such a system would gather its information over periods of minutes rather than seconds, for example, by using some inexpensive form of profiling.

Our results could be extended by combining type feedback with a more aggressive global optimizer. Since inlining enlarges the scope of global optimizations, a good optimizer may significantly increase the benefits of inlining. In the context of exploratory programming, more aggressive optimization may not be warranted since it is likely to increase compile pauses. However, such optimizations could be useful in an application extractor generating a stand-alone executable for an application, or in a system using batch-style compilation. Such an optimizer could also use additional information from profilers [30] or from type inferencers [5] to generate even better code.

Finally, SELF-93 inherits the overcustomization problem from previous SELF systems. Because the system always customizes compiled code to the exact receiver type, it sometimes duplicates code even when it would be better to share compiled code between different receiver types, either because the code is identical or because the optimization benefits of customization are negligible. Originally (when customization was first introduced in SELF-89), there was

no choice but to always customize, since the system was non-adaptive. Now, with adaptive recompilation, it should be possible to customize lazily, i.e., only in performance-critical parts of the program, and only if customization would benefit the code (i.e., if many messages are sent to `self`). Similarly, the system could choose to perform method cloning [62] rather than method inlining. Such a system might simultaneously reduce code size and improve performance (because of reduced instruction cache overhead) and thus represents an interesting area for future work.

11.3 Summary

When SELF was first proposed in 1987, most people (including the author) would have placed it in the "hopeless" category when asked to judge the feasibility of an efficient implementation. Only a few years later, Chambers and Ungar showed that SELF could achieve excellent performance for a restricted set of programs [26]. Unfortunately, large object-oriented programs still didn't perform as well. Furthermore, runtime performance was achieved at the expense of considerable compiler complexity and compilation speeds that were too slow for interactive use. This thesis has presented a third-generation SELF system that simultaneously improves execution and compilation speed while decreasing implementation complexity. One lesson we certainly learned during the past five years is that "hopeless" isn't. We hope that the progress made in implementing SELF will encourage others to find even better solutions to the implementation challenges posed by object-oriented languages, by other systems heavily using late binding and abstraction, or by other minimalistic languages.

We also hope our work contributes to make exploratory programming environments more popular. In the past, such systems have often suffered from performance problems that have limited their acceptance. For example, programs executed by interpreted Smalltalk systems could run many times slower than their C equivalents. Adaptive recompilation and type feedback, combined with the speed of modern hardware, could significantly extend the applicability of exploratory programming environments. With better performance yet good interactive behavior, these techniques make exploratory programming possible both for pure languages and for application domains requiring higher ultimate performance, finally reconciling exploratory programming with high performance.

Glossary

Adaptive recompilation. The technique of adaptively (at runtime) selecting heavily-used methods and recompiling (and optimizing) them for better performance (see chapters 5 and 6).

Block. A block is an object representing a lexically-scoped closure (similar to a Smalltalk block). SELF blocks cannot be executed after their enclosing scope has returned.

Compiled method. A compiled method contains native code for a SELF source method, plus various additional information such as dependency links (to invalidate compiled code after source-code changes), relocation information (for the garbage collector), and debugging information (for source-level debugging). Because of customization, a single source method may have several compiled methods (see Chapter 2).

Customization. The idea of generating several versions of compiled code for a single source method, each version customized to a particular receiver type (see Section 2.2.2 on page 5).

Dynamic deoptimization. The process of transforming optimized code into unoptimized code on demand, e.g., for source-level debugging (see Chapter 10).

Inline cache. A dispatch mechanism often used in dynamically-typed languages (see Chapter 3).

Map. An internal data structure describing the implementation type of an object, i.e., its exact format and the contents of its constant slots [23]. Among other things, maps are used for message dispatch and for customization.

Megamorphic send. A highly polymorphic send (very many receiver types occur).

Method. An object representing a source method (function). The method object contains a byte-coded version of the source code and is created by the parser. The SELF compilers use the byte codes as their input (rather than parsing the source).

Message. A message is a request to an object to perform some operation. The object to which the request is sent is called the *receiver*. A message send is the action of sending a message to a receiver.

Monomorphic send. A non-polymorphic send (i.e., all calls have the same receiver type).

NIC. "Non-Inlining Compiler." The non-optimizing compiler used to generate initial compiled code in SELF-93. See Chapter 4.

Non-local return. A non-local return is a return from a method activation resulting from performing a return (i.e., evaluating an expression preceded by the '^' operator) from within a lexically enclosed block. A non-local return forces returns from all activations between the method activation and the activation of the block performing the return.

Pause clustering. The technique of combining clusters of pauses occurring in short succession into one longer pause in order to better characterize pauses as they are experienced by a user (see Section 9.1).

PIC. "Polymorphic Inline Cache"—an extension of inline caching that handles polymorphic call sites more efficiently. See Chapter 3.

Polymorphic send. A send that encounters more than one receiver type during execution.

SELF-91. The previous SELF implementation described by Chambers [21].

SELF-93. The SELF implementation described in this thesis.

Splitting. The idea of delaying a control flow merge (and the loss of type information that comes with the merge) by copying code (see Section 2.2.2 on page 5).

Type. Types come in two flavors. Abstract types (or interface types) specify only the operations that an object supports. Concrete types (also called implementation types) specify implementation details such as the number and order of instance variables and the methods specific to the object. SELF has no explicit types; when we say “type” in this thesis, we usually mean “implementation type.” In other words, we use the terms “type” and “map” interchangeably.

Type feedback. An optimization technique based on feeding back concrete receiver type information to the compiler. The type information is collected during program execution and can be fed back to the compiler either during execution (with dynamic compilation) or afterwards (static compilation). See Chapter 5.

Type prediction. The technique of optimizing a send by predicting likely receiver types and specializing the send for the predicted type(s). See Section 2.2.2 on page 5, and Chapter 5.

Appendix A. Detailed Data

This appendix contains detailed data for graphs or abbreviated tables in the main text.

Number of types	% of all inline caches
0	12.4%
1	81.9%
2	4.4%
3	0.8%
4	0.2%
5-10	0.2%
11-99	0.05%
> 99	0.02%

Table A-1. Distribution of degree of polymorphism in call sites

Benchmark	time saved
Richards	26.8%
Parser	22.3%
PrimMaker	10.9%
UI	0.0%
PathCache	2.5%
PolyTest	81.6%

Table A-2. Execution time saved by PICs

Benchmark	execution time (ms)		
	SELF-93 nofeedback	SELF-93	SELF-91
CecilComp	1,348	953	1,144
CecilInt	2,035	1,085	2,026
DeltaBlue	7,44	210	687
Mango	2,423	1,526	2,292
PrimMaker	2,520	1,227	2,279
Richards	922	591	693
Typeinf	1,448	769	1,388
UI1	716	686	645
UI3	656	528	571

Table A-3. Execution time comparison

Benchmark	unoptimized	SELF-91	SELF-93	SELF-93 nofeedback
CecilComp	3,542,858	N/A	120,418	472,422
CecilInt	1,254,244	262,424	48,383	274,166
DeltaBlue	2,030,319	407,283	202,241	413,024
Mango	3,290,836	642,545	204,048	681,070
PrimMaker	3,934,308	819,277	76,273	602,217
Richards	6,962,721	839,478	151,819	888,817
Typeinf	2,363,131	288,982	101,858	293,815
UI1	1,727,021	256,573	213,145	288,176
UI3	1,274,863	274,262	101,884	301,344

Table A-4. Number of dynamically-dispatched calls

		C++			
Benchmark	SELF-93	GNU g++	Sun CC	GNU g++ (virtuals)	Sun CC (virtuals)
DeltaBlue	210	87	98	149	148
Richards	591	249	352	546	634

Table A-5. Performance relative to C++ (all times in ms simulated time)

	CPU time (s on SPARCstation-2)		
Benchmark	SELF-93	Smalltalk	Lisp
DeltaBlue	0.27	0.60	0.36
Richards	0.71	2.58	1.93

Table A-6. Performance relative to Smalltalk and Lisp

Table A-4 shows the reduction in execution time per eliminated non-inlined call, i.e., (time(SELF–93–nofeedback) – time(SELF-93)) / (sends(SELF–93–nofeedback) – sends(SELF-93)).

Benchmark	time saved per eliminated call (microseconds)
CecilComp	1.24
CecilInt	4.83
DeltaBlue	2.97
Mango	1.76
PrimMaker	2.94
Richards	0.53
Typeinf	3.59
UI1	0.93
UI3	0.59

Table A-7. Time saved per inlined call

	execution time spent in type tests	
	SELF-93	SELF-93-nofeedback
CecilComp	12.3%	13.8%
CecilInt	11.9%	9.0%
DeltaBlue	25.0%	18.8%
Mango	12.8%	11.6%
PrimMaker	10.3%	11.2%
Richards	25.1%	21.0%
Typeinf	15.0%	12.8%
UI1	13.7%	14.7%
UI3	12.4%	13.5%
median	12.8%	13.5%

Table A-8. Time spent performing type tests

	message dispatch			inline type tests		
	tests	compares	cycles	tests	compares	cycles
CecilComp	546,015	793,855	6,790,883	277,228	355,319	1,137,057
CecilInt	295,288	400,168	7,069,389	100,649	135,700	685,761
DeltaBlue	447,876	662,369	5,791,762	217,346	258,579	607,880
Mango	727,969	932,536	11,669,182	184,671	217,240	810,692
PrimMaker	1,029,215	1,348,048	10,661,923	522,203	727,772	1,632,402
Richards	925,171	1,048,157	6,160,975	851,675	1,252,514	2,074,679
Typeinf	352,815	621,545	7,027,102	184,894	215,052	726,844
UI1	313,154	415,423	3,916,524	144,274	161,884	466,381
UI3	327,869	396,965	3,227,868	161,789	170,350	526,749

Table A-9. Type tests in SELF-93-nofeedback

	message dispatch			inline type tests		
	tests	compares	cycles	tests	compares	cycles
CecilComp	41,767	74,711	1,119,979	575,388	641,296	3,810,074
CecilInt	31,318	57,852	1,229,190	245,624	265,815	3,792,112
DeltaBlue	17,420	21,860	149,580	479,015	500,914	2,098,188
Mango	94,536	183,049	2,450,616	665,499	727,873	7,025,884
PrimMaker	64,695	91,547	1,084,755	619,888	667,575	3,894,629
Richards	230,622	390,800	1,986,384	1,506,243	1,645,618	3,946,876
Typeinf	74,103	145,230	1,666,096	339,608	379,485	3,237,115
UI1	62,071	105,381	1,127,272	315,671	328,247	2,583,563
UI3	31,075	37,013	507,468	345,181	374,186	2,336,869

Table A-10. Type tests in SELF-93

	average number of comparisons per type test sequence	
	dispatch	feedback tests
CecilComp	1.79	1.11
CecilInt	1.85	1.08
DeltaBlue	1.25	1.05
Mango	1.94	1.09
PrimMaker	1.42	1.08
Richards	1.69	1.09
Typeinf	1.96	1.12
UI1	1.70	1.04
UI3	1.19	1.08
median	1.70	1.08

Table A-11. Number of comparisons per type test sequence

Benchmark	execution time (ms)			
	SELF-91		SELF-93	
	original	modified	original	modified
bubble	89	540	299	372
intmm	240	465	275	293
perm	57	191	136	171
puzzle	842	1452	1238	1394
queens	42	165	111	130
quick	84	219	232	263
sieve	108	499	326	345
towers	128	309	179	226
tree	620	758	601	690
bubble_oo	83	83	283	283
intmm_oo	247	248	262	262
perm_oo	59	59	175	176
queens_oo	36	112	96	115
quick_oo	86	86	232	232
towers_oo	72	131	86	101
tree_oo	544	681	572	664

Table A-12. Performance on the Stanford integer benchmarks

SPEC'89	gcc	espresso	li	eqntott	geomean
load	18.5%	23.2%	22.9%	15.4%	19.7%
store	8.1%	4.9%	10.3%	1.1%	4.7%
cond. branch	16.3%	17.8%	17.0%	25.7%	18.9%
uncond. branch	1.4%	1.4%	2.4%	0.1%	0.9%
call/return/jump	2.5%	1.2%	4.4%	0.7%	1.7%
nop	1.0%	0.3%	1.4%	0.1%	0.4%
sethi	11.7%	12.4%	6.4%	19.7%	11.6%
arithmetic	8.2%	9.6%	6.6%	10.2%	8.5%
logical	4.9%	6.7%	0.8%	0.2%	1.5%
shift	8.3%	3.8%	7.7%	1.0%	3.9%
compare	17.2%	17.7%	17.0%	25.8%	19.1%
other	1.8%	1.0%	3.2%	0.1%	0.9%

Table A-13. SPECInt89 instruction usage (from [33])

	CecilComp	CecilInt	DeltaBlue	Mango	PrimMaker	Richards	Typeinf	UI1	UI3	geomean
load	16.8%	17.4%	18.2%	16.9%	19.4%	14.7%	15.4%	15.9%	11.7%	16.1%
store	7.2%	7.1%	6.2%	5.1%	10.0%	4.9%	5.6%	6.2%	6.9%	6.4%
cond. branch	13.9%	12.0%	16.7%	13.4%	11.3%	17.5%	13.9%	16.2%	13.2%	14.1%
uncond. branch	2.9%	2.3%	3.6%	2.7%	2.0%	4.0%	2.3%	3.5%	1.6%	2.7%
call/return/ jump	4.0%	5.8%	1.7%	5.7%	3.9%	2.2%	4.7%	4.5%	5.8%	4.0%
nop	6.2%	4.4%	12.5%	4.0%	4.9%	11.5%	5.3%	5.8%	5.8%	6.2%
sethi	5.1%	5.7%	5.9%	5.9%	4.0%	6.2%	8.0%	6.5%	5.9%	5.8%
arithmetic	11.1%	11.4%	7.7%	10.4%	18.1%	6.8%	10.4%	9.0%	6.9%	9.8%
logical	15.0%	15.1%	12.3%	15.9%	11.9%	11.1%	16.1%	13.2%	20.8%	14.4%
shift	2.9%	4.0%	1.3%	4.0%	2.4%	3.3%	1.8%	2.6%	4.1%	2.7%
compare	12.6%	11.5%	12.3%	11.8%	10.0%	16.2%	13.1%	13.6%	9.0%	12.1%
other	2.4%	3.2%	1.6%	4.2%	2.0%	1.6%	3.3%	3.0%	8.3%	2.9%

Table A-14. SELF-93 instruction usage

	Richards gcc	Richards CC	Richards (virtual) gcc	Richards (virtual) CC	DeltaBlue gcc	DeltaBlue CC	DeltaBlue (virtual) gcc	DeltaBlue (virtual) CC
load	25.1%	19.8%	37.2%	46.2%	20.4%	16.9%	30.5%	36.1%
store	8.1%	5.6%	4.1%	3.7%	7.4%	6.2%	4.2%	4.7%
cond. branch	15.5%	10.8%	7.8%	7.1%	13.1%	9.1%	7.1%	6.8%
uncond. branch	2.6%	0.8%	0.9%	0.5%	2.3%	2.3%	0.7%	1.7%
call/return/jump	8.0%	21.4%	17.0%	14.1%	8.6%	18.2%	15.7%	13.5%
nop	2.2%	6.6%	1.5%	2.6%	3.0%	4.7%	1.4%	3.1%
sethi	2.1%	1.6%	0.8%	0.8%	5.0%	4.3%	2.9%	3.3%
arithmetic	3.3%	2.4%	9.7%	8.6%	8.3%	5.7%	11.1%	10.4%
logical	9.4%	14.9%	8.9%	5.8%	10.0%	16.3%	8.8%	8.2%
shift	0.8%	0.5%	0.4%	0.4%	3.1%	2.3%	6.5%	1.7%
compare	16.6%	10.0%	7.5%	6.6%	12.5%	7.7%	6.0%	5.7%
other	6.5%	5.7%	4.2%	3.8%	6.3%	6.3%	5.1%	4.7%

Table A-15. Instruction usage on Richards and Deltablue (C++)

	10^3 bytes allocated
CecilComp	3,165
CecilInt	3,070
DeltaBlue	809
Mango	2,688
PrimMaker	5,951
Richards	3
Typeinf	1,573
UI1	1,362
UI3	1,002

Table A-16. Allocation behavior of benchmarks (SELF-93)

The following graphs show the cache miss ratios for the caches discussed in Section 8.4. All ratios are given relative to the number of references in their category. For example, the data read miss ratio is relative to the total number of data reads performed by the program.

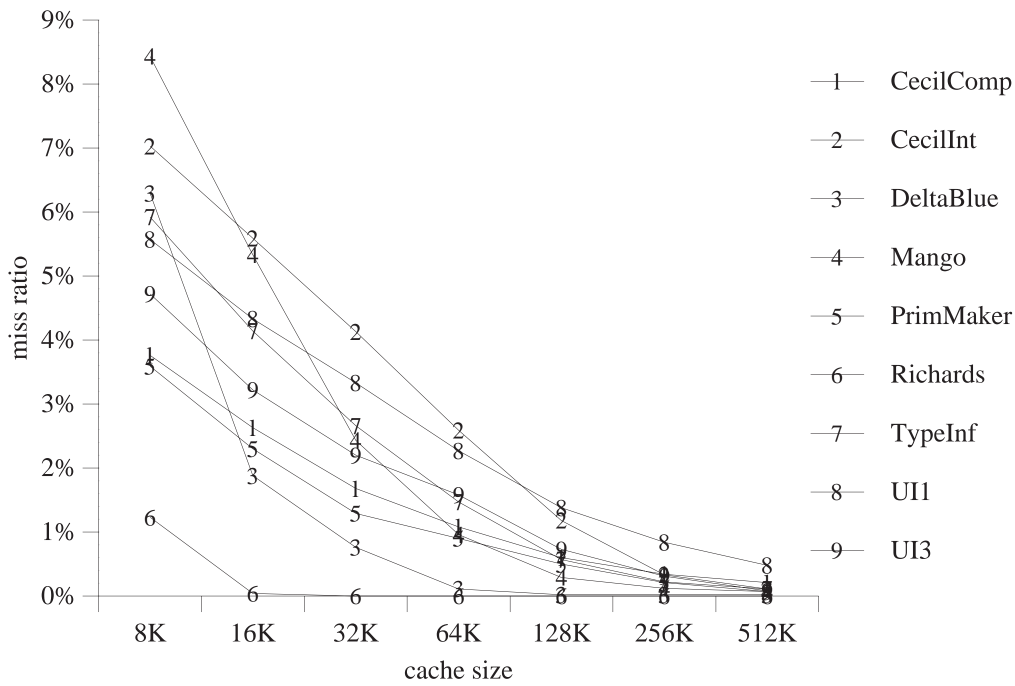


Figure A-1. I-cache miss ratios in SELF-93

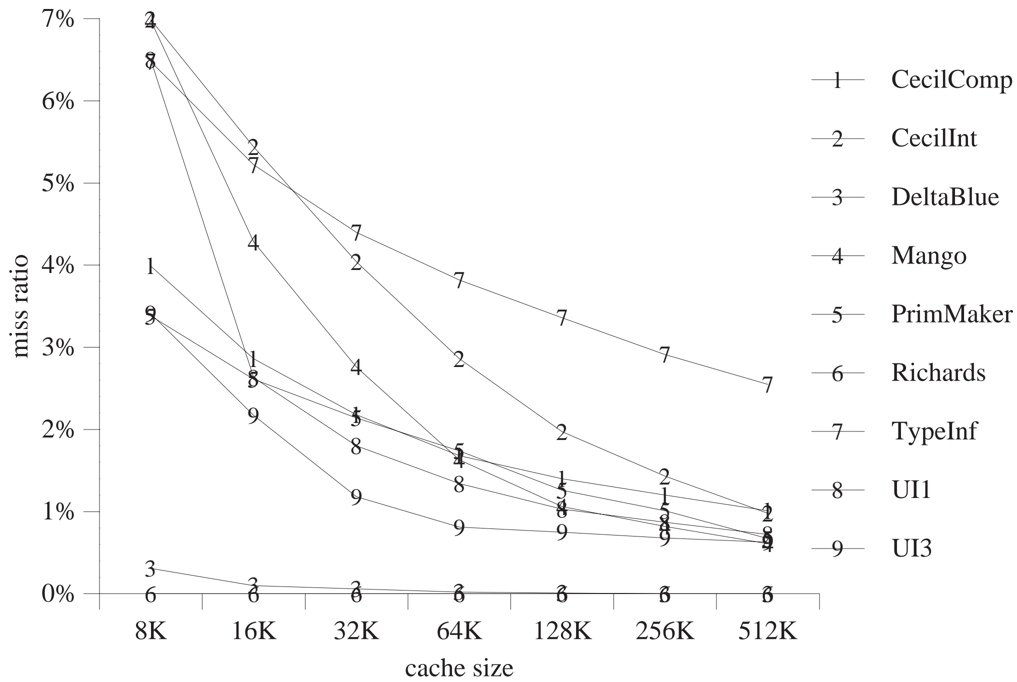


Figure A-2. Data read miss ratios in SELF-93

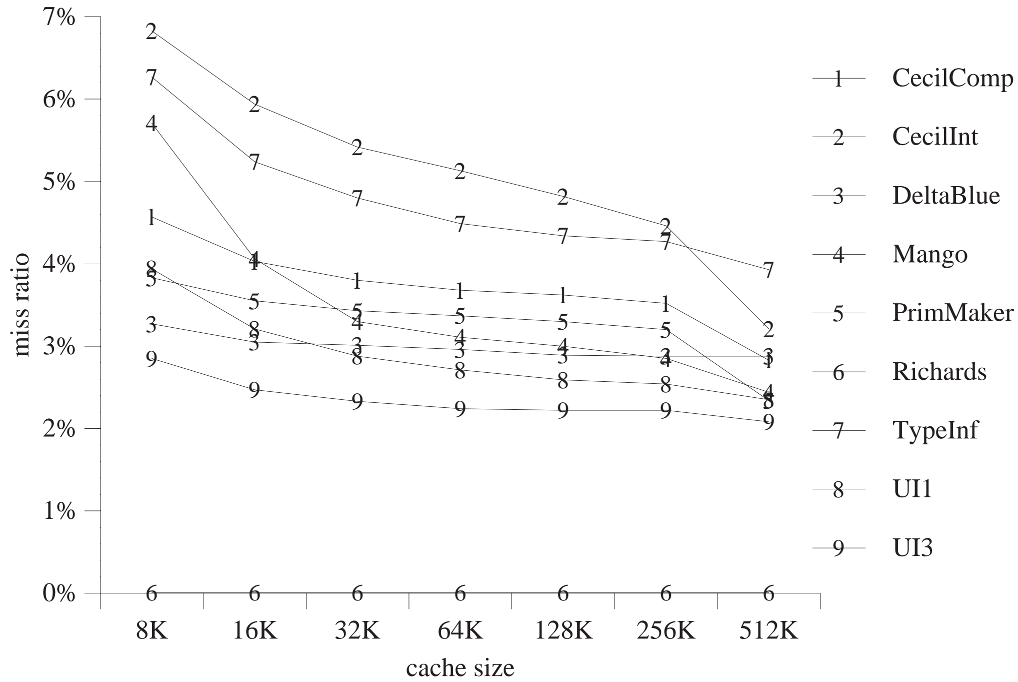


Figure A-3. Data write miss ratios in SELF-93

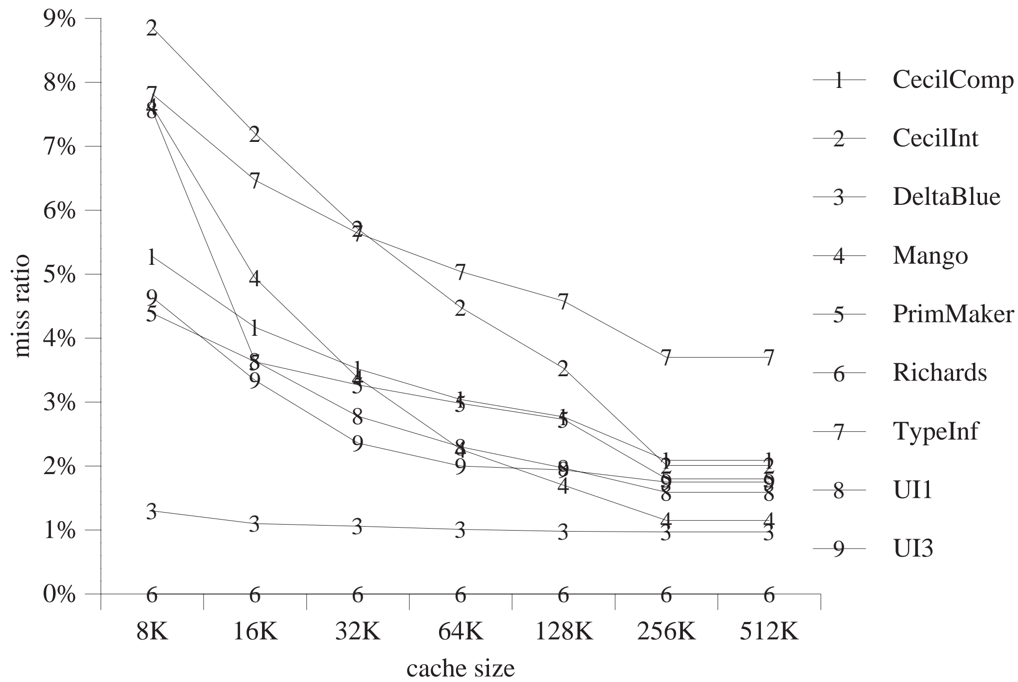


Figure A-4. Data read miss ratios in SELF-93 (write-noallocate cache)

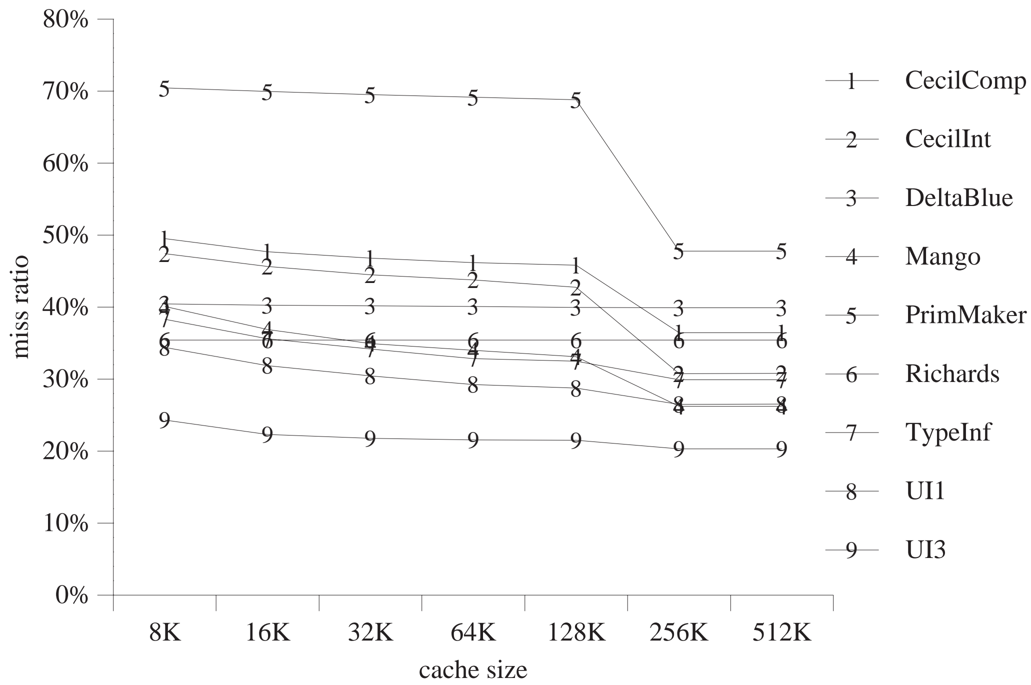


Figure A-5. Data write miss ratios in SELF-93 (write-noallocate cache)

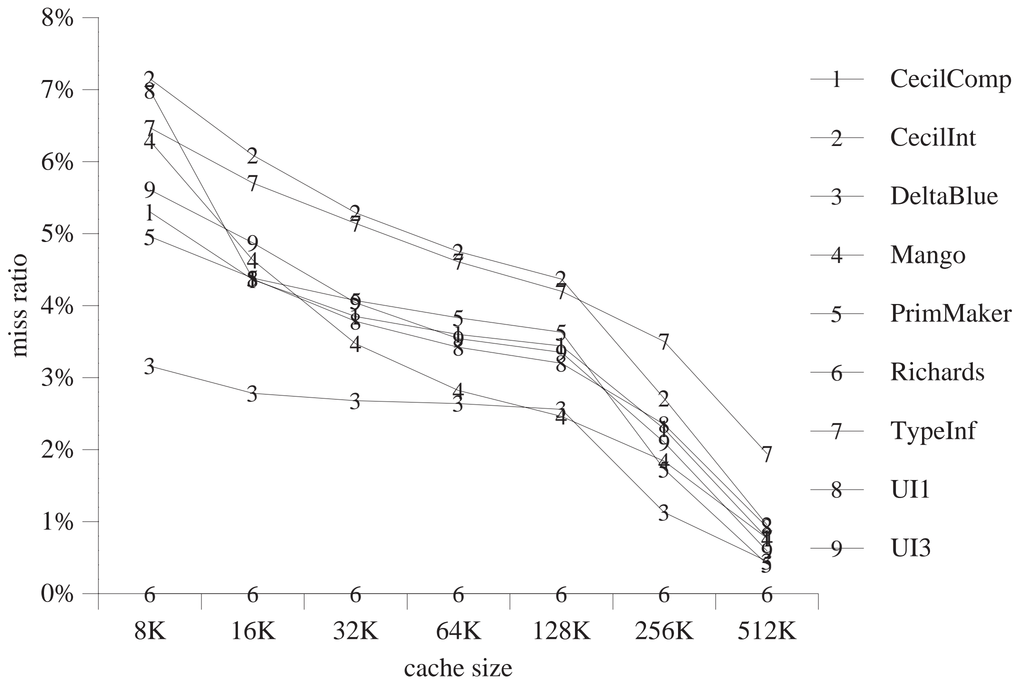


Figure A-6. Data read miss ratios in SELF-93 (50K eden)

parameter	value
max. size of compiled method	2,500 instructions
max. size of inlined method w/o block arguments	100 instructions
max. size of inlined method with block arguments	150 instructions
invocation limit (for recompilation)	10,000
invocation counter half-life time	15 seconds[a]

Table A-17. SELF-93 compiler configuration parameters

a. The performance measurements obtained witth the instruction-level simulator effectively use an infinite half-life time since the simulator cannot simulate timer interrupts.

Pause length (seconds)	SPARCstation-2	“Current” (3x faster)	“Future” (10x faster)
0.0	407	496	625
0.1	77	81	22
0.2	37	37	3
0.3	28	23	1
0.4	23	2	0
0.5	32	3	0
0.6	19	4	0
0.7	14	3	0
0.8	9	0	0
0.9	12	0	0
1.0	7	1	0
1.1	4	0	0
1.2	1	0	0
1.3	1	0	0
1.4	1	0	0
1.5	1	0	0
1.6	0	0	0
1.7	0	0	0
1.8	0	0	0
1.9	4	0	0
2.0	1	0	0
3.1	1	0	0

Table A-18. Pause length histogram data

References

[1] Mark B. Abbot and Larry L. Peterson. A Language-Based Approach to Protocol Implementation. *Computer Communications Review 22 (4)*: 27-38, 1992.

[2] Ali-Reza Adl-Tabatabai and Thomas Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *PLDI '93 Conference Proceedings*, pp. 13-25. Published in *SIGPLAN Notices 28(6)*, June 1993.

[3] Ali-Reza Adl-Tabatabai and Thomas Gross. Evicted variables and the interaction of global register allocation and symbolic debugging. In *POPL '93 Conference Proceedings*, 1993.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, Tools*. Addison-Wesley, 1988.

[5] 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.

[6] Frances E. Allen and John Cocke. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, ed. R. Rustin. Prentice-Hall, 1972.

[7] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. *OOPSLA '92 Conference Proceedings*, p. 110-127, Vancouver, BC, October 1992.

[8] Apple Computer, Inc. Object Pascal User's Manual. Cupertino, 1988.

[9] A. H. Borning and D. H. H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *Conference Record of the Ninth Annual Symposium on Foundations of Computer Science*, p. 133-139, 1982.

[10] Frank Mueller and David B. Whalley. Avoiding Unconditional Jumps by Code Replication. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, p. 322-330. Published as *SIGPLAN Notices 27(7)*, July 1992.

[11] Robert G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. In *OOPSLA '86 Conference Proceedings*, p. 151-158, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November 1986.

[12] J. R. Bell. Threaded Code. *Communications of the ACM 16*:370-372, 1973.

[13] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the 1986 ACM/IEEE Fall Joint Computer Conference*, p. 36-40, Dallas, TX, November 1986.

[14] Brian K. Bray and M. J. Flynn. *Write caches as an alternative to write buffers*. Technical Report CSL-TR 91-470, Computer Systems Laboratory, Stanford University, April 1991.

[15] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica 21*, p. 473-484, 1984.

[16] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[17] 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.

[18] G. Chaitin et al. Register Allocation via Coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 98-105, June 1982.

[19] D. D. Chamberlin et al. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Transactions on Database Systems 6(1)*, p. 70-94, March 1981.

[20] Craig Chambers. Cost of garbage collection in the Self system. *OOPSLA'91 GC Workshop*, October 1991.

[21] Craig Chambers, *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. Thesis, Stanford University, April 1992.

[22] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989. Published as *SIGPLAN Notices 24(7)*, July 1989.

[23] 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. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[24] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 150-164, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[25] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. Published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[26] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, October 1991.

[27] Craig Chambers. *The Cecil Language - Specification and Rationale*. University of Washington, Technical Report CS TR 93-03-05, 1993.

[28] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. *User Interface Software and Technology Conference Proceedings*, Atlanta, GA, November 1993.

[29] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience 22(5)*, p. 349-369, May 1992.

[30] Pohua P. Chang, Scott A. Mahlke, and W. W. Hwu. *Using profile information to assist classic code optimizations.* Technical Report UILU-ENG-91-2219, University of Illinois at Urbana-Champaign, April 1991.

[31] David Chase. *Garbage Collection and Other Optimizations.* Ph.D. dissertation, Computer Science Department, Rice University, 1987.

[32] Andrew A. Chien, Vijay Karamcheti, and John Plevyak. *The Concert System: Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs*. University of Illinois at Urbana-Champaign, Technical Report UIUC DCS-R-93-1815, 1993.

[33] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks. *ASPLOS IV*, Santa Clara, CA, April 1991.

[34] 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.

[35] Thomas J. Conroy and Eduardo Pelegri-Llopart. *An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations*. In [86].

[36] K.D. Cooper, M.W. Hall, and Ken Kennedy. Procedure Cloning. In *IEEE Intl. Conference on Computer Languages*, p. 96-105, Oakland, CA, April 1992.

[37] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, p. 125-134.

[38] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP performance using TCP and SPEC workloads. *ISCA'21 Conference Proceedings*, p. 60-70, May 1994.

[39] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience* 18(8): 775-90.

[40] Jeffrey Dean and Craig Chambers. *Training compilers for better inlining decisions.* University of Washington Technical Report 93-05-05, 1993.

[41] Jeffrey Dean and Craig Chambers. *Toward better inlining decisions using inlining trials.* 1994 ACM Conference on Lisp and Functional Programming, p. 273-282, Orlando, FL, June 1994.

[42] Marcia A. Derr and Shinichi Morishita. Design and Implementation of the Glue-Nail Database System. In *SIGMOD '93 Conference Proceedings*, pp. 147-156, 1993. Published as *SIGMOD Record, 22(2)*, June 1993.

[43] L. Peter Deutsch. *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*. In [86].

[44] 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.

[45] L. Peter Deutsch. Private Communication.

[46] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *OOPSLA '89 Conference Proceedings*, p. 211-214, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[47] David Ditzel. Private Communication, 1993.

[48] Amer Diwan, David Tarditi, and Eliot Moss. Memory Subsystem Performance of Programs with Intensive Heap Allocation. In *21st* Annual ACM Symposium on Principles of Programming Languages, p. 1-14, January 1994.

[49] Amer Diwan, David Tarditi, and Eliot Moss. Private communication, August 1993.

[50] Karel Driesen. Selector Table Indexing and Sparse Arrays. In *OOPSLA '93 Conference Proceedings*, pp. 259-270, Washington, D.C., 1993. Published as *SIGPLAN Notices 28(10)*, October 1993.

[51] Eric J. Van Dyke. A dynamic incremental compiler for an interpretative language. *HP Journal*, p. 17-24, July 1977.

[52] Peter H. Feiler. *A Language-Oriented Interactive Programming Environment Based on Compilation Technology.* Ph.D. dissertation, Carnegie-Mellon University, 1983.

[53] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[54] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[55] Michael Franz. Technological steps toward a software component industry. *International Conference on Programming Languages and System Architecture*, Springer Verlag Lecture Notes in Computer Science 782, March 1994.

[56] Free Software Foundation. GNU C++ compiler. Boston, MA, 1993.

[57] 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.

[58] Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[59] Susan L. Graham, Peter Kessler, and Marshall McKusick. An execution profiler for modular programs. *Software—Practice and Experience* 13:617-685.

[60] Justin Graver and Ralph Johnson. A Type System for Smalltalk. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January, 1990.

[61] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, p. 1-8, 1978.

[62] Mary Wolcott Hall. *Managing Interprocedural Optimization*. Technical Report COMP TR91-157 (Ph.D. thesis), Computer Science Department, Rice University, April 1991.

[63] Gilbert J. Hansen, *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. Ph.D. Thesis, Carnegie-Mellon University, 1974.

[64] Richard L. Heintz. *Low-Level Optimizations for an Object-Oriented Programming Language*. M. Sc. Thesis, University of Illinois, Urbana-Champaign, 1990.

[65] John L. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions of Programming Languages and Systems 4(3)*, July 1982.

[66] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Technical Report UCB/CSD 87/381, Computer Science Division, University of California, Berkeley, November 1987.

[67] W. Wilson Ho and Ronald A. Olsson. An Approach to Genuine Dynamic Linking. *Software—Practice and Experience 21(4)*, p. 375-390, April 1991.

[68] C. A. R. Hoare. Hints on programming language design. In *First Annual ACM Symposium on Principles of Programming Languages*, p. 31-40, 1973.

[69] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The* SELF *Manual, Version 1.1*. Unpublished manual, February 1991.

[70] 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.

[71] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, p. 32-43. Published as *SIGPLAN Notices 27(7)*, July 1992.

[72] Urs Hölzle. A Fast Write Barrier for Generational Garbage Collectors. *Proceedings of the OOPSLA'93 Workshop on Garbage Collection*, Washington, D.C., September 1993.

[73] Antony Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. In *OOPSLA'92 Proceedings*, pp. 92-109.

[74] W.W. Hwu and P.P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, p. 246-57, Portland, OR, June 1989. Published as *SIGPLAN Notices 24(7)*, July 1989.

[75] Daniel H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA '86 Conference Proceedings*, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[76] Gordon Irlam. *SPA—SPARC analyzer toolset*. Available via ftp from cs.adelaide.edu.au, 1991.

[77] Geeske Joel, Gregory Aplet, and Peter M. Vitousek. Leaf morphology along environmental gradients in Hawaiian Metrosideros polymorpha. *Biotropica 26(1)*, p. 17-22, 1994.

[78] Ralph Johnson(ed.). Workshop on Compiling and Optimizing Object-Oriented Programming Languages. In *Addendum to the OOPSLA '87 Conference Proceedings*, p. 59-65, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 23(5)*, May 1988.

[79] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*, p. 18-26, San Diego, CA, October 1988. Published as *SIGPLAN Notices 23(11)*, November 1988.

[80] Ronald L. Johnston. The Dynamic Incremental Compiler of APL\3000. In *Proceedings of the APL '79 Conference*. Published as *APL Quote Quad 9(4)*, p. 82-87.

[81] Norm Jouppi. Cache Write Policies and Performance. In *ISCA'20 Conference Proceedings*, pp. 191-201, San Diego, CA, 1993. Published as *Computer Architecture News 21(2)*, May 1993.

[82] David Keppel, Susan J. Eggers, and Robert R. Henry. *A Case for Runtime Code Generation*. Technical Report UW TR 91-11-04, Department of Computer Science and Engineering, University of Washington, Seattle, 1991.

[83] David Keppel, Susan J. Eggers, and Robert R. Henry. *Evaluating Runtime-Compiled Value-Specific Optimizations*. Technical Report UW TR 93-11-02, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[84] Peter Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 78-84, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990.

[85] Gregor Kiczales and Luis Rodriguez. *Efficient Method Dispatch in PCL*. Technical Report SSL-89-95, Xerox PARC, 1989.

[86] Glenn Krasner, ed., *Smalltalk-80: Bits of History and Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[87] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, p. 301-314, San Francisco, CA, April, 1990.

[88] Elgin Lee. *Object Storage and Inheritance for* SELF*, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University, 1988.

[89] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Communications of the ACM 26 (6)*: 419-429.

[90] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*, p. 214-223, Portland, OR, September 1986. Published as *SIGPLAN Notices 21(11)*, November 1986.

[91] Henry Lieberman, Lynn Andrea Stein, and David Ungar. The Treaty of Orlando. In *Addendum to the OOPSLA '87 Conference Proceedings*, p. 43-44, Orlando, FL, October 1987. Published as *SIGPLAN Notices 23(5)*, May 1988.

[92] Mark Linton, John Vlissides, and Paul Calder. Composing User Interfaces with Interviews. *IEEE Computer 22(2)*:8-22, February 1989.

[93] Cathy May. MIMIC: A fast system /370 simulator. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, June 1987.

[94] Scott McFarling. Procedure Merging with Instruction Caches. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, p. 71-79. Published as *SIGPLAN Notices 26(6)*, June 1991.

[95] Scott McFarling. *Program analysis and optimization for machines with instruction cache.* Technical Report CSL-TR-91-493, Stanford University, 1991.

[96] MIPS Computer Systems, *MIPS Language Programmer's Guide*. MIPS Computer Systems, Sunnyvale, CA, 1986.

[97] J. G. Mitchell, *Design and Construction of Flexible and Efficient Interactive Programming Systems*. Ph.D. Thesis, Carnegie-Mellon University, 1970.

[98] W. G. Morris. A Prototype Coagulating Code Generator. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, p. 45-58. Published as *SIGPLAN Notices 26(6)*, June 1991.

[99] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers 38(5)*:663-678, May 1989.

[100] John Ousterhout. *Why aren't operating systems getting faster as fast as hardware?* DEC Western Research Laboratory, Technical Note TN-11, 1989.

[101] David A. Patterson and David Ditzel. The Case for the Reduced Instruction Set Computer. *Computer Architecture News 8(6)*:25-33, October 1980.

[102] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM 28 (1)*: 8-21, January 1985.

[103] Rob Pike, Bart N. Locanthi, and John F. Reiser. Hardware/Software Trade-Offs for Bitmap Graphics on the Blit. *Software—Practice and Experience 15(2)*, p. 131-151, February 1985.

[104] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance Trade-offs in Cache Design. *Proceedings of the 15th International Symposium on Computer Architecture*, p. 290-298, May 1988.

[105] Calton Pu and Henry Massalin. *An Overview of the Synthesis Operating System.* Technical Report CUCS-470-89, Department of Computer Science, Columbia University, New York, 1989.

[106] 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.

[107] B. R. Rau. Levels of Representation of Programs and the Architecture of Universal Host Machines. *Proceedings of Micro 11*, Asilomar, CA, November 1978.

[108] Mark B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. Technical Report MIT/LCS/TR-581, MIT, September 1993.

[109] Stephen E. Richardson. *Evaluating interprocedural code optimization techniques*. Ph.D. thesis, Computer Systems Laboratory Technical Report CSL TR 91-460, Stanford University, Feb 1991.

[110] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *OOPSLA '88 Conference Proceedings*, p. 27-35, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[111] H. J. Saal and Z. Weiss. A Software High Performance APL Interpreter. *APL Quote Quad 9 (4)*:74-81, 1979.

[112] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and Runtime Compilation. *Proceedings of the International Workshop on Compilers for Parallel Computers*, Paris, December 1990.

[113] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience* 23 (5): 529-566.

[114] H. Schlaeppi and H. Warren. Design of the FDS Interactive Debugging System. *IBM Research Report RC7214*, IBM Yorktown Heights, July 1978.

[115] The SELF Group. *The SELF Manual, Version 2.0*. Unpublished manual, August 1992.

[116] Robert A. Shaw. *Empirical Analysis of a LISP System*. Stanford University, Computer Systems Laboratory, Technical Report CSL-TR-88-351, 1988.

[117] Patrick G. Sobalvarro. *A lifetime-based collector for LISP systems on general-purpose computers.* B.S. Thesis, EECS Dept., Massachusetts Institute of Technology, 1988.

[118] SPARC International. *The SPARC Architecture Manual (Version 8)*. Prentice Hall, NJ, 1992.

[119] SPARC International. *The SPARC Architecture Manual (Version 9)*. Prentice Hall, NJ, 1993.

[120] Richard Stallman. *The GNU C compiler.* Free Software Foundation, 1991.

[121] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and Software Approaches. In *ASPLOS II Conference Proceedings*, October 1987.

[122] Markus Strässler. *Implementierung von Oberon-*SELF. Diploma Thesis, Institute for Computer Systems, ETH Zürich, March 1992.

[123] Sun Microsystems. *The Viking Microprocessor (T.I. TMS S390Z50) User Documentation.* Part No. 800-4510-02, November 1990.

[124] Norihisa Suzuki. Inferring Types in Smalltalk. In *Proceedings of the 8th Symposium on the Principles of Programming Languages*, 1981.

[125] Norihisa Suzuki and Minoru Terada. Creating Efficient Systems for Object-Oriented Languages. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, January, 1984.

[126] G. Taylor, P. Hilfinger, J. Larus, D. Patterson, and B. Zorn. Evaluation of the SPUR Lisp Architecture. *Proceedings of the 13th Symposium on Computer Architecture*, p. 444-452, Tokyo, 1986.

[127] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML Without Reverse Engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990, p. 1-12.

[128] David Ungar and David Patterson. *Berkeley Smalltalk: Who Knows Where the Time Goes?* In [86].

[129] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Eleventh Annual International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

[130] David Ungar and David Patterson. What Price Smalltalk? In *IEEE Computer 20(1)*, January 1987.

[131] David Ungar. Generation Scavenging: A Non-Disruptive High-Performance Storage Reclamation Algorithm. *SIGPLAN Symposium on Practical Software Development Environments*, p. 157-167, Pittsburgh, PA, April 1984.

[132] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, Cambridge, MA, 1987.

[133] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, p. 227-241, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[134] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. Published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[135] Jan Vitek and Nigel Horspool. Taming message passing: Efficient method lookup for dynamically-typed languages. In *ECOOP '94 Proceedings*, p. 432-449, Springer Verlag Lecture Notes on Computer Science 821, July 1994.

[136] David W. Wall. Global Register Allocation at Link Time. In *SIGPLAN '86 Symposium on Compiler Construction*, p. 264-275.Published as *SIGPLAN Notices 21 (7)*, July 1986.

[137] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, p. 59-70. Published as *SIGPLAN Notices 26(6)*, June 1991.

[138] Paul R. Wilson and Thomas G. Mohler. Design of the Opportunistic Garbage Collector. In *OOPSLA '89 Conference Proceedings*, p. 23-35, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[139] Paul R Wilson and Thomas G Moher. A card-marking scheme for controlling intergenerational references in generation-based GC on stock hardware. *SIGPLAN Notices 24 (5)*:87-92.

[140] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. *Caching Considerations for Generational GC: a Case for Large and Set-Associative Caches*. University of Illinois at Chicago Technical Report UI-EECS-90-5, December 1990.

[141] Mario Wolczko. Private communication, April 1991.

[142] Ifor Williams and Mario Wolczko. An Object-Based Memory Architecture. In *Proc. 4th Intl. Workshop on persistent object systems*, Martha's Vineyard, MA, September 1990.

[143] Polle T. Zellweger. *An Interactive High-Level Debugger for Control-Flow Optimized Programs*. Xerox PARC Technical Report CSL-83-1, January 1983.

[144] Polle T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Ph.D. dissertation, Computer Science Department, University of California, Berkeley, 1984. Also published as *Xerox PARC Technical Report CSL-84-5*, May 1984.

[145] Benjamin Zorn. *Barrier Methods for Garbage Collection.* Technical Report CU-CS-494-90, University of Colorado at Boulder, November 1990.

[146] Lawrence W. Zurawski and Ralph E. Johnson. *Debugging Optimized Code With Expected Behavior.* Unpublished manuscript, 1992.