Language and Compiler Support for Dynamic Code Generation

by

Massimiliano A. Poletto

S.B., Massachusetts Institute of Technology (1995)
M.Eng., Massachusetts Institute of Technology (1995)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author ....
Department of Electrical Engineering and Computer Science
June 23, 1999

Certified by ....
M. Frans Kaashoek
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Language and Compiler Support for Dynamic Code Generation

by

Massimiliano A. Poletto

Submitted to the Department of Electrical Engineering and Computer Science
on June 23, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Dynamic code generation, also called run-time code generation or dynamic compilation, is the creation of executable code for an application while that application is running. Dynamic compilation can significantly improve the performance of software by giving the compiler access to run-time information that is not available to a traditional static compiler. A well-designed programming interface to dynamic compilation can also simplify the creation of important classes of computer programs. Until recently, however, no system combined efficient dynamic generation of high-performance code with a powerful and portable language interface. This thesis describes a system that meets these requirements, and discusses several applications of dynamic compilation.

First, the thesis presents ‘C (pronounced “Tick-C”), an extension to ANSI C that allows the programmer to express and manipulate C code fragments that should be compiled at run time. ‘C is currently the only language to provide an imperative programming interface suitable for high-performance dynamic compilation. It can be used both as an optimization, to improve performance relative to static C code, and as a software engineering tool, to simplify software that involves compilation and interpretation.

Second, the thesis proposes several algorithms for fast but effective dynamic compilation, including fast register allocation, data flow analysis, and peephole optimization. It introduces a set of dynamic compilation benchmarks, and uses them to evaluate these algorithms both in isolation and as part of an entire ‘C implementation, tcc. Dynamic ‘C code can be significantly faster than equivalent static code compiled by an optimizing C compiler: two- to four-fold speedups are common on the benchmarks in this thesis. Furthermore, tcc generates code efficiently—at an overhead of about 100 to 600 cycles per generated instruction—which makes dynamic compilation practical in many situations.

Finally, the thesis establishes a taxonomy of dynamic optimizations. It highlights strategies for improving code with dynamic compilation, discusses their limits, and illustrates their use in several application areas.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis would not exist without Dawson Engler and Frans Kaashoek. Dawson was the main designer of the ‘C language, and his enthusiasm for dynamic code generation sparked my interest in this area. Throughout grad school he was a source of energy, ideas, and intellectual curiosity. I am also happy and grateful to have worked with Frans. He has been an excellent advisor—friendly, encouraging, and enthusiastic. I envy his energy and unshakeable optimism. He deserves all the credit for convincing me to finish this thesis when I was nauseated and ready to abandon it all in favor of new projects.

I also thank Wilson Hsieh, Vivek Sarkar, and Chris Fraser. Wilson was a great help at many points in my ‘C work, and has given me friendship and good advice even though I once ate his ivy plant. I much enjoyed working with Vivek, and thank him for the opportunity to be supported by an IBM Cooperative Fellowship. Chris shared my enthusiasm for the Olympic Mountains, and his minimalist aesthetic continues to influence my thinking about engineering.

My thanks to all of the PDOS group for providing an excellent (if sometimes abrasive) work environment over the last five years. It was fun and a little humbling to interact with everyone, from Anthony and Debby when I first started, to JJ, Chuck, and Benjie now that I'm almost done. Neena made bureaucracy easy, and valiantly tried to keep my office plants alive: my evil dark thumb eventually vanquished her good green thumb, and by this I am deeply saddened. A special thank-you to my two long-time officemates, David and Eddie, who put up with my moods and taught me more than most about computers and about myself, and also to Emmett, whose humor helped to keep me out of the bummer tent.

I would have quit grad school many crises ago without the support and love of Rosalba. I hope to have given back a fraction of what I have received.

Infine, di nuovo, un grazie di cuore ai miei genitori per tutto.

Sometimes a scream is better than a thesis.

Ralph Waldo Emerson

The ‘C language was originally designed by Engler et al. [23]. I was involved in later modifications and extensions to the language. Several parts of this thesis, including much of Chapters 2, 3, and 4, are adapted from Poletto et al. [57]. Section 5.2 is a revision of text originally published in Poletto and Sarkar [58].

I was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985, N66001-96-C-8522, and F30602-97-2-0288, by a NSF National Young Investigator Award to Frans Kaashoek, and by an IBM Cooperative Fellowship.

Contents

1 Introduction **15**
1.1 An Example . . . 15
1.2 Contributions . . . 17
1.3 Issues in Dynamic Compilation . . . 18
1.3.1 Implementation Efficiency . . . 18
1.3.2 Interface Expressiveness . . . 20
1.4 Related Work . . . 21
1.4.1 Ease of Use and Programmer Control . . . 21
1.4.2 Compilation Overhead and Programmer Control . . . 23
1.5 Outline . . . 25

2 Language Design **27**
2.1 The ‘C Language . . . 27
2.1.1 The Backquote Operator . . . 28
2.1.2 Type Constructors . . . 29
2.1.3 Unquoting Operators . . . 30
2.1.4 Special Forms . . . 31
2.2 Subtleties and Pitfalls . . . 33
2.2.1 The Dynamic Compilation Model . . . 33
2.2.2 Unquoting . . . 34
2.2.3 Interactions with C . . . 35
2.3 Potential Extensions to ‘C . . . 37
2.3.1 Syntactic Sugar for Dynamic Code . . . 37
2.3.2 Declarative Extensions . . . 38
2.3.3 To Add or Not to Add? . . . 40

3 Applications **43**
3.1 A Taxonomy of Dynamic Optimizations . . . 43
3.1.1 Specialization to Data Values . . . 44
3.1.2 Specialization to Data Types . . . 50
3.1.3 Elimination of Call Overhead . . . 53
3.1.4 Elimination of Interpretation Overhead . . . 54
3.2 Limits of Dynamic Compilation . . . 58

4 Implementation and Performance **59**
4.1 Architecture . . . 59
4.2 The Dynamic Compilation Process . . . 60
4.2.1 Static Compile Time . . . 60
4.2.2 Run Time . . . 62
4.3 Run-Time Systems . . . 64
4.3.1 VCODE . . . 64
4.3.2 ICODE . . . 64
4.4 Evaluation Methodology . . . 66

4.4.1 Benchmark-Specific Details 66
4.4.2 Measurement Infrastructure 67
4.4.3 Comparison of Static and Dynamic Code 67
4.5 Performance Evaluation 67
4.5.1 Performance 68
4.5.2 Analysis 71
4.6 Summary 72

5 Register Allocation **75**
5.1 One-Pass Register Allocation 75
5.2 Linear Scan Register Allocation 76
5.2.1 Background 77
5.2.2 The Algorithm 78
5.2.3 Evaluation 81
5.2.4 Discussion 86
5.3 Two-Pass Register Allocation 90

6 One-Pass Data Flow Analyses **95**
6.1 The Algorithms 95
6.2 Evaluation 98
6.3 Summary 100

7 Fast Peephole Optimizations **101**
7.1 Related Work 101
7.2 Design 102
7.2.1 Rewrite Rules 102
7.2.2 Optimizer Architecture 103
7.2.3 The bpo Interface 103
7.3 Efficient Pattern-Matching Code 105
7.4 Applications 107
7.5 Evaluation 109
7.6 Summary 111

8 Conclusion **113**
8.1 Summary and Contributions 113
8.1.1 Language 113
8.1.2 Implementation 113
8.1.3 Applications 114
8.2 Future Work 114
8.2.1 Code Generation Issues 114
8.2.2 Architectural Interactions 115
8.2.3 Automatic Dynamic Compilation 115
8.3 Final Considerations 115

A A Dynamic Compilation Cookbook **117**
A.1 Numerical Methods 117
A.1.1 Matrix Operations 117
A.1.2 Numerical Integration 119
A.1.3 Monte Carlo Integration 120
A.1.4 Evaluation of Functions 121
A.1.5 Fast Fourier Transform 121
A.1.6 Statistics 122
A.1.7 Arbitrary Precision Arithmetic 122
A.2 Non-Numerical Algorithms 123
A.2.1 Sorting Algorithms 123

A.2.2 Graph Algorithms . . . 126
A.2.3 String and Pattern Matching . . . 126
A.2.4 Circuit Simulation . . . 127
A.2.5 Compression . . . 128
A.3 Computer Graphics . . . 129
A.3.1 Clipping . . . 129
A.3.2 Shading . . . 129
A.3.3 Scan Conversion of Conics . . . 129
A.3.4 Geometric Modeling . . . 129
A.3.5 Image Transformation . . . 130

List of Figures

1-1 General outline of the dynamic compilation process. 16
1-2 An example of dynamic compilation. 16
1-3 Tradeoff of dynamic compilation overhead versus code quality. 19
1-4 Dynamic compilation overhead versus code quality for Java. 19
1-5 Partial design space for dynamic code generation systems. 22
1-6 Another view of the dynamic code generation design space. 24

3-1 A hash function written in C. 45
3-2 Specialized hash function written in ‘C. 45
3-3 Code to create a specialized exponentiation function. 46
3-4 ‘C code to specialize multiplication of a matrix by an integer. 46
3-5 A dot-product routine written in C. 47
3-6 ‘C code to build a specialized dot-product routine. 47
3-7 A tail-recursive implementation of binary search. 47
3-8 ‘C code to create a “self-searching” executable array. 47
3-9 An executable data structure for binary search. 48
3-10 Code to create a hard-coded finite state machine. 49
3-11 ‘C code to generate a specialized swap routine. 50
3-12 Generating specialized copy code in ‘C. 51
3-13 Sample marshaling code in ‘C. 52
3-14 Unmarshaling code in ‘C. 52
3-15 ‘C code that implements Newton’s method for solving polynomials. 53
3-16 A small pipe written in ‘C. 54
3-17 ‘C code for composing data pipes. 55
3-18 Compilation of a small query language with ‘C. 56
3-19 A sample statement parser from a compiling interpreter written in ‘C. 57

4-1 Overview of the tcc compilation process. 60
4-2 Sample closure assignments. 61
4-3 Sample code-generating functions. 61
4-4 Speedup of dynamic code over static code. 68
4-5 Cross-over points, in number of runs. 69
4-6 Dynamic code generation overhead using VCODE. 71
4-7 Dynamic code generation overhead using ICODE. 72

5-1 Linear scan register allocation algorithm. 79
5-2 An example set of live intervals. 80
5-3 Register allocation overhead for ‘C benchmarks. 82
5-4 Two types of pathological programs. 83
5-5 Register allocation overhead as a function of the number of simultaneously live variables for programs of type (a). 84
5-6 Register allocation overhead as a function of program size for programs of type (b). . 85
5-7 Run time of ‘C benchmarks compiled with different register allocation algorithms. . 85

5-8 Run time of static C benchmarks compiled with different register allocation algorithms. 86
5-9 An acyclic flow graph. 87
5-10 Overhead of full live variable analysis and SCC-based liveness analysis. 88
5-11 The two-pass register allocation process. 92

6-1 A small control flow graph. 97
6-2 Work done (basic blocks touched) during null pointer check removal. 99
6-3 Results from null pointer check removal. 99
6-4 Work done (basic blocks touched) during availability analysis. 99
6-5 Results from availability analysis. 100

7-1 Grammar for binary peephole optimization rewrite rules. 102
7-2 Generating a binary peephole optimizer in two different ways. 103
7-3 An outline of the bpo rewriting process. 104
7-4 Example of bpo operation. 104
7-5 Generating efficient binary pattern matching code with multiway radix search trees. 106
7-6 An example of code generated by the MRST-based algorithm. 108
7-7 Comparative performance of lookup-table algorithm and MRST-based algorithm. . . 109
7-8 Run-time performance of code generated using VCODE and then optimized by bpo. 110
7-9 Compile-time performance of binary optimization. 111

List of Tables

2.1 The ‘C special forms. 31

4.1 Descriptions of benchmarks. 66
4.2 Performance of convolution on an 1152 x 900 image in xv. 70

5.1 Register allocation overhead for static C benchmarks using linear scan and binpacking. 83
5.2 Run time of static C benchmarks, as a function of register allocation algorithm. . . . 86
5.3 Run time of static C benchmarks compiled with linear scan allocation, as a function of liveness analysis technique. 88
5.4 Run time of static C benchmarks compiled with linear scan allocation, as a function of flow graph numbering. 89
5.5 Run time of static C benchmarks compiled with linear scan allocation, as a function of spilling heuristic. 89

Chapter 1

Introduction

Traditional compilers operate in an off-line manner. Compilation—running the compiler to produce an executable from source code—is seen as completely distinct from the execution of the compiler-generated code. All instructions that are executed are located in the executable's text segment, and the text segment is never changed. On some architectures, the text segment is even mapped into read-only memory. Split instruction and data caches underscore, at an architectural level, this assumption of the fundamental difference between code and data.

Dynamic code generation (DCG), also called run-time code generation (RTCG) or dynamic compilation, blurs this traditional separation. A compiler that enables dynamic code produces both "normal" static code and special code (often called a "generating extension" [43] in the partial evaluation community) that generates additional code at run time. This process is outlined in Figure 1-1. Delaying some compilation until run time gives the compiler information that it can use to improve code quality and that is unavailable during static compilation. Information about run-time invariants provides new opportunities for classical optimizations such as strength reduction, dead-code elimination, and inlining. Furthermore, explicit access to dynamic compilation can simplify some programming tasks, such as the creation of compiling interpreters and lightweight languages. This thesis discusses ways to make dynamic code generation available to programmers and to improve the quality of dynamic code beyond that of static code.

As a result of traditional compilers' off-line nature, few care about their performance. It is assumed that the optimizing compiler will be run only once, when development is complete, whereas the resulting code may be run many times. As a result, static compiler optimizations are almost always designed with the sole goal of improving the compiled code, without care for compilation speed. In contrast, compilation overhead has a direct effect on the performance of programs that involve dynamic code generation. More dynamic compilation effort usually leads to better dynamic code, but excessive dynamic optimizations can be counterproductive if their overhead outweighs their benefits. Much of this thesis addresses performance and implementation issues in fast dynamic code generation.

1.1 An Example

One of the main applications of dynamic compilation is the creation of code that is specialized to some run-time data—essentially, dynamic partial evaluation. Consider the example in Figure 1-2. The static code on the left hand side implements a two-dimensional median filter. For each point in the data set, it multiplies neighboring points by values in the mask, computes the mean, and writes out the new value. For each data point, the code performs approximately $2ij$ memory reads and ij multiplications. Furthermore, to handle the borders of the data, the code must either perform special checks or use several convolution loops with masks of different shapes.

If the mask were to always remain the same, we could easily partially evaluate—either by hand or automatically—this code with respect to the mask. If the mask contained all 1s, as in the example, then the filter would simply take the mean of adjacent data points, and each data point would

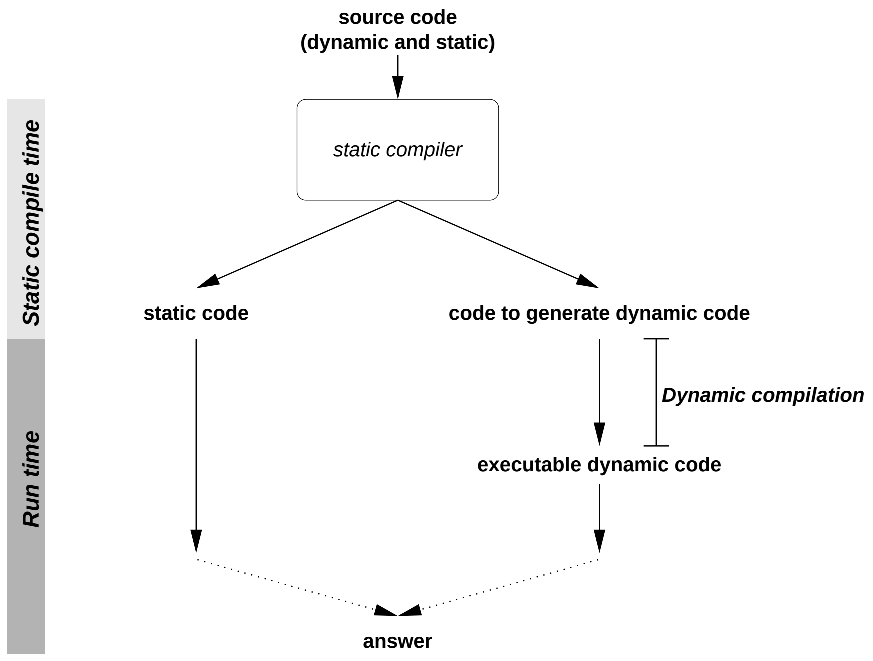

Figure 1-1: General outline of the dynamic compilation process.

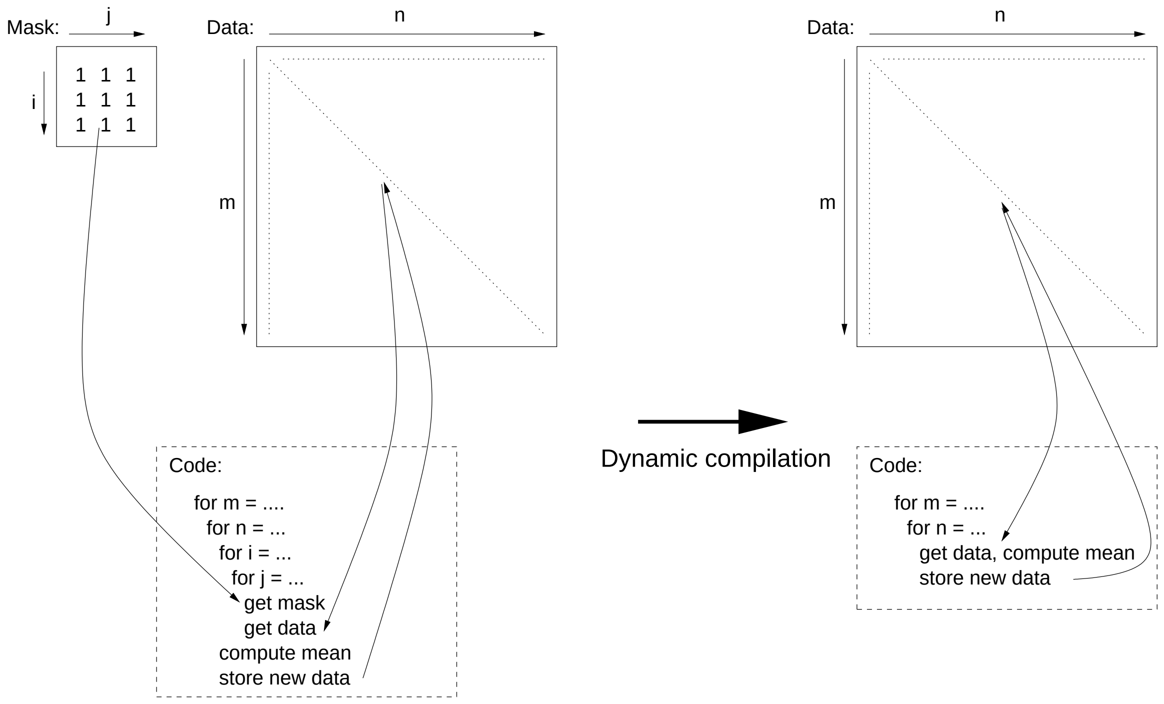

Figure 1-2: An example of dynamic compilation. On the left, static code convolves a two-dimensional data set with a convolution mask. On the right, dynamic code has been specialized with respect to the convolution mask.

require only ij memory reads and no multiplications. Border cases would be handled efficiently, and maybe the main loop could be partially unrolled.

Dynamic compilation allows us to do exactly such optimizations, but even when the values in the mask change at run time. The right half of Figure 1-2 illustrates the dynamically generated code. Section 4.5 shows that this sort of optimization can considerably improve the performance of a popular graphics package like xv. As we shall see in Chapter 3, dynamic optimizations can take on many forms: executable data structures, dynamically inlined functions, simple and efficient compiling interpreters, and many others.

1.2 Contributions

Dynamic code generation has been employed successfully in several systems [46] to provide performance that could not have been attained with static code, but implementing it was generally complicated and tedious. Interpreted programming environments such as Lisp [70] and Perl [75] furnish primitives for dynamic construction and interpretation of code that are expressive and easy to use, but they have high run-time overhead. Prior to the work in this thesis and to similar systems developed at about the same time at the University of Washington [4, 38, 39] and at INRIA [14], no language or tool offered practical and high-level access to efficient dynamic generation of high-performance code.

This thesis describes the design and implementation of a language that provides these features, and presents some of its applications. It addresses three central aspects of dynamic code generation:

Language support. It describes 'C (pronounced "Tick-C"), an extension to ANSI C that supports dynamic code generation, and discusses alternative methods of exposing dynamic compilation to the programmer.

Implementation. It presents tcc, a 'C compiler that we have implemented. It also introduces several algorithms that can be used to improve the effectiveness of dynamic compilation. These algorithms include three different schemes for fast register allocation, one-pass data flow analyses, and fast binary peephole optimizations.

Applications. It establishes a taxonomy of applications for dynamic code generation, and illustrates several uses of dynamic compilation in areas such as numerical computing and computer graphics.

'C is a superset of ANSI C that supports high-level and efficient dynamic code generation. It allows the programmer to express dynamic code at the level of C expressions and statements and to compose arbitrary dynamic code at run time. It enables a programming style slightly similar to that of Lisp, and simplifies the creation of powerful and portable dynamic code. Since 'C is a superset of ANSI C, it can be used incrementally to improve the performance of existing C programs. This thesis describes 'C in detail, and discusses possible extensions and other ways of expressing dynamic code.

tcc is an efficient and freely available experimental implementation of 'C that has been used by us and by others to explore dynamic compilation. It has two run-time systems that allow the user to trade dynamic code quality for dynamic code generation speed. If compilation speed must be maximized, dynamic code generation and register allocation can be performed in one pass; if code quality is most important, the system can construct and optimize an intermediate representation prior to code generation. The thesis describes the tcc implementation, and also evaluates fast compilation algorithms, such as linear scan register allocation and fast binary peephole optimizations, that are generally applicable to dynamic compilation.

The thesis explores applications of dynamic code in two ways. First, it defines a space of optimizations enabled by dynamic compilation. It describes the space in terms of the underlying code generation mechanisms and the higher-level objectives of the optimizations, and then populates it with representative examples. Some of these examples form a suite of dynamic code benchmarks: they served to obtain the measurements obtained in this thesis and have also been used by others [39].

Second, the thesis takes a more focused approach and presents uses of dynamic code generation in the areas of numerical methods, algorithms, and computer graphics. Each of these examples makes use of some of the methods that define the design space. The goal is both to span the breadth of dynamic compilation techniques and to provide some realistic detail within some specific application areas.

1.3 Issues in Dynamic Compilation

Dynamic code generation can serve two purposes. First, most commonly, it is used to improve performance by exposing run-time information to the compiler. Second, less obviously, it can serve as a programming tool: as we shall see, given an adequate interface, it can help to simplify important classes of programming problems.

1.3.1 Implementation Efficiency

In most cases, dynamic code generation is employed because it can improve performance. This fact places two requirements on any dynamic code generation system. First, dynamic code must be efficient. If the dynamic compiler produces poor code, then even aggressive use of run-time information on the part of the programmer may not result in performance improvements relative to static code.

Second, the dynamic compilation process must be fast. The overhead of static compilation—*static compile time*—does not matter much, but that of dynamic compilation—*dynamic compile time*—does, because it is a part of the run time of the compiled program. We measure the overhead of compilation in cycles required to produces one instruction of code. A static compiler such as gcc, the GNU C compiler, requires tens of thousands of cycles to generate one binary instruction. A dynamic compilation system should produce code hundreds of times more quickly—on the order of tens or hundreds of cycles per instruction—to be practical. Even at such speeds, however, dynamic compilation often provides a performance advantage relative to static code only if the overhead of code generation can be amortized over several runs of the dynamic code. In other words, to obtain good performance results when writing dynamic code, one must usually make sure that the dynamic code is executed more than once after it is created.

Put differently, consider some static code s and equivalent dynamic code d. Let the run time of one execution of s be simply T_s. We can express the total time necessary to create and run the dynamic code as $T_d = T_{d_c} + T_{d_r}$, where T_{d_c} is the dynamic compile time and T_{d_r} is the run time of dynamic code. Dynamic code improves performance when $T_s > T_{d_c} + T_{d_r}$. The first point discussed above translates to minimizing T_{d_r}; the second point translates to minimizing T_{d_c}. Since for many problems T_{d_c} can be large relative to T_s and T_{d_r}, one may wish to create dynamic code that is used several (k) times, such that $kT_s > T_{d_c} + kT_{d_r}$. The value of k at which this inequality becomes true is called the *cross-over point.*

Figure 1-3 illustrates these concepts. The horizontal axis is a high-level measure of work: for instance, a count of the image frames processed with some filter. It is usually a linear function of k in the discussion above. The vertical axis measures the amount of time required to perform this work. For the purposes of this discussion, we ignore situations in which dynamic optimization provides no benefit over static code. Static code is denoted in the figure by a line with a slope of T_s: it takes T_s units of time to perform one unit of work. The line marked Fast DCG represents mediocre dynamic code generated by a dynamic compiler that emphasizes compilation speed over code quality. The dynamic code is faster than static code—the corresponding line has a smaller slope, T_{d_r}. However, dynamic compilation requires some overhead, T_{d_c} before any work can done, so the line does not start at the origin, but at a point along the vertical axis. The cross-over point for this dynamic compiler is A: dynamic compilation improves performance whenever the amount of work to be done is greater than that denoted by A. The third line, marked Good DCG, represents a dynamic compiler that generates better good code more slowly. Its startup cost—the starting position on the vertical axis—is even higher, but the resulting code is still faster—the line has smaller slope. The cross-over point relative to static code occurs at B, but the cross-over point relative to "quick

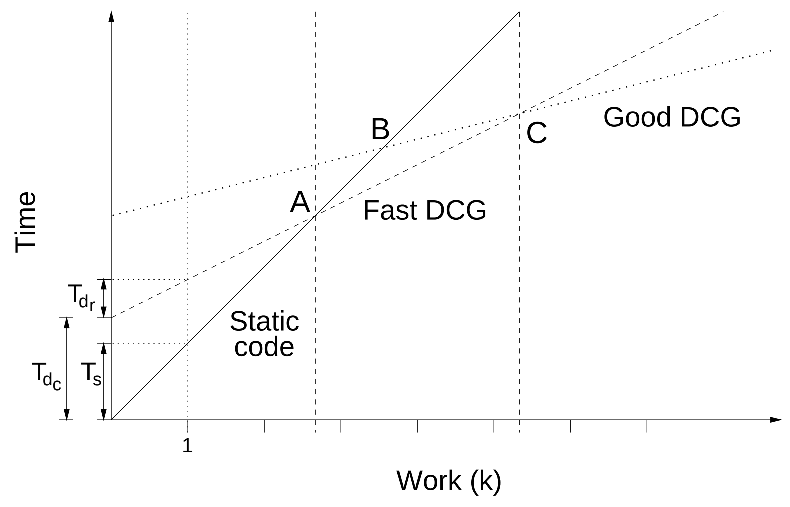

Figure 1-3: Tradeoff of dynamic compilation overhead versus code quality.

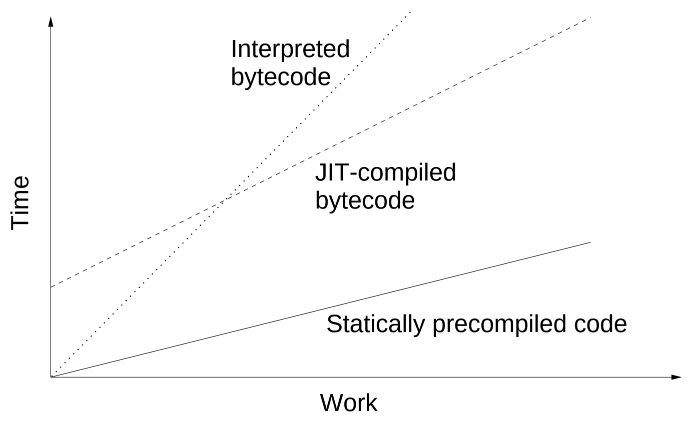

Figure 1-4: Dynamic compilation overhead versus code quality for Java.

and dirty" dynamic code only happens at C. In other words, we can divide the space into three regions separated by the dashed vertical lines in the figure. The leftmost region, to the left of A, corresponds to a small workload, in which static code performs best. The central region, between A and C, corresponds to an intermediate workload, and is best-suited to the mediocre dynamic code generated quickly. Finally, the rightmost region corresponds to large workloads, in which the greater speed of well-optimized dynamic code justifies the overhead of more dynamic optimization. The holy grail of dynamic compilation is to minimize both the slope and the vertical offset of the lines in the figure.

It is instructive to compare Figure 1-3 with Figure 1-4, which illustrates a similar tradeoff of compilation overhead versus code quality for the case of mobile code such as Java [37]. Mobile code is generally transmitted as bytecodes, which are then either interpreted or dynamically ("just-in-time") compiled to native code. Bytecode interpretation is slow but has low latency; just-in-time compilation requires a little overhead but produces better code. However, it does not aim to exceed the performance of static code. The goal of most JIT compilers is to approach the performance of statically compiled code without incurring the overhead of full static compilation. This goal is in contrast to dynamic compilation as discussed in this thesis, in which static code is taken as the baseline to be improved by dynamic optimization.

One way to reduce T_{d_r} without increasing T_{d_c} is to shift as much work as possible to static compile time. 'C does this to some extent: for instance, it is statically typed, so type checking of dynamic code can occur statically. However, as we shall see in Chapter 2, creation of dynamic code in 'C usually involves composing many small pieces of dynamic code. In general, the order of composition can be determined only at run time, which reduces the effectiveness of static analysis and makes dynamic optimization necessary. As a result, much of this thesis discusses fast code improvement algorithms that can be used to reduce T_{d_r} without excessively impacting T_{d_c}. tcc, the 'C compiler we have implemented, also addresses this problem by having two run-time code generation systems, which coincide with the two dynamic compilers illustrated in Figure 1-3. One run-time system, based on VCODE [22], generates code of medium quality very quickly; it is useful when the dynamic code will be used few times and when it will run faster than static code even if it is not optimized. The second run-time system, ICODE, incurs greater dynamic compilation overhead but produces better code, so it is suitable when one expects that the dynamic code will run many times.

1.3.2 Interface Expressiveness

Dynamic code generation is naturally suited for simplifying programming tasks that involve compilation and interpretation. Consider an interpreter for a little language. Traditionally, one would write a parser that produces bytecodes and then a bytecode interpreter. With dynamic code generation, on the other hand, one can write a front end that creates dynamic code during parsing, which leaves the back end's dirty work to the dynamic compilation system. This process removes the need for bytecodes and a bytecode interpreter, and probably improves the performance of the resulting code. The key to such flexibility is a powerful programming interface.

Current programming interfaces for dynamic compilation, fall into two categories: *imperative* and *declarative*. The imperative style of dynamic compilation, adopted by 'C, requires that the programmer explicitly create and manipulate pieces of dynamic code: until it is dynamically compiled, dynamic code is data. Similarly, lists in Lisp and strings in Perl are data, but they can be evaluated as code. This model requires some experience—the programmer must think about writing code that creates code—but it provides a high degree of control. By contrast, in the declarative model, the programmer simply annotates traditional static source code with various directives. These directives usually identify run-time constants and specify various code generation policies, such as the aggressiveness of specialization and the extent to which dynamic code is cached and reused. Dynamic code generation then happens automatically.

Declarative systems have the advantage that most annotations preserve the semantics of the original code, so it is easy to remove them and compile and debug the program without them. Furthermore, in the common case, inserting annotations into a program is probably easier than writing in an imperative style. Declarative systems also have a slight performance advantage. Imperative dynamic code programming usually relies on composition and manipulation of many small fragments of dynamic code. As a result, dynamic code is difficult to analyze statically, which limits the amount of dynamic optimization that can be moved to static compile time. By contrast, in declarative systems users cannot cannot arbitrarily combine and manipulate code. Annotations have well-defined effects, so the compiler can usually predict at least the structure—if not the exact size and contents—of the dynamic code. The difference in interface, therefore, has a fundamental effect on the implementation of the two kinds of systems, and makes declarative systems somewhat better suited to static set-up of dynamic optimizations.

Nonetheless, imperative systems have a significant advantage: expressiveness. We must first distinguish between two kinds of expressiveness—that related to optimization, and that related more closely to the semantics of the program. The first kind, optimization expressiveness, is the degree to which the programming system allows the programmer to express dynamic optimization and specialization. An imperative system like 'C has a slight advantage in this area, but the most sophisticated declarative system, DyC [38, 39], can achieve similar degrees of control.

The main appeal of imperative systems lies in their programming flexibility, the "semantic expressiveness" mentioned above. This flexibility goes beyond simple optimization, and is not one of the design goals of declarative systems. Consider once again the little language interpreter example. Using a sophisticated declarative system, we could dynamically partially evaluate the bytecode interpreter with respect to some bytecodes, producing a compiler and therefore improving the code's performance. With an imperative system like 'C, however, we would not even have to write the back end or bytecode interpreter in the first place—we could write a parser that built up dynamic code during the parse, and rely on the dynamic code generator to do everything else. The performance of compiled code would come for free, as an incidental side effect of the simpler programming style!

Realistically, of course, not all applications contain compiler- or interpreter-like components. In those cases, the main purpose of both imperative and declarative systems is to allow dynamic optimizations, so the potentially greater ease of use of an annotation-based system may be preferable. In the end, the choice of system is probably a matter both of individual taste and of the specific programming situation at hand.

1.4 Related Work

Dynamic code generation has been used to improve the performance of systems for a long time [46]. Keppel et al. [47] demonstrated that it can be effective for several different applications. It has served to increase the performance of operating systems [7, 24, 61, 62], windowing operations [55], dynamically typed languages such as Smalltalk [20] and Self [12, 41], and simulators [74, 77]. Recent just-in-time (JIT) compilers for mobile code such as Java [37] use dynamic compilation techniques to improve on the performance of interpreted code without incurring the overhead of a static compiler.

Berlin and Surati [6] reported 40x performance improvements thanks to partial evaluation of data-independent scientific Scheme code. The conversion from Scheme to a low-level, special purpose program exposes vast amounts of fine-grained parallelism, which creates the potential for additional orders of magnitude of performance improvement. They argued that such transformations should become increasingly important as hardware technology evolves toward greater instruction-level parallelism, deeper pipelines, and tightly coupled multiprocessors-on-a-chip. Although Berlin and Surati used off-line partial evaluators, dynamic specialization of high-level languages like Scheme can be expected to have similar benefits.

Unfortunately, past work on high-performance dynamic code generation was not concerned with making it portable or easy to use in other systems. Keppel [45] addressed some of the issues in retargeting dynamic code generation. He developed a portable system for modifying instruction spaces on a variety of machines. His system dealt with the difficulties presented by caches and operating system restrictions, but it did not address how to select and emit actual binary instructions.

Some languages already provide the ability to create code at run time: most Lisp dialects [44, 70], Tcl [54], and Perl [75], provide an "eval" operation that allows code to be generated dynamically. The main goal of this language feature, however, is programming flexibility rather than performance. For instance, most of these languages are dynamically typed, so not even type checking can be performed statically. Furthermore, the languages are usually interpreted, so the dynamic code consists of bytecodes rather than actual machine code.

1.4.1 Ease of Use and Programmer Control

Figure 1-5 presents one view of a dynamic code generation design space. We distinguish several dynamic code generation systems according to two criteria: on the horizontal axis, the degree of control that the programmer has on the dynamic compilation process, and on the vertical axis, the system's ease of use. Both of these categories are somewhat subjective, and the figure is not completely to scale, but the classification exercise is still useful.

At the top left of the scale are systems that do not give the programmer any control over the dynamic compilation process, but that use dynamic compilation as an implicit optimization. compilation. For instance, Self [12] uses dynamic profiling of type information and dynamic code generation to optimize dynamic dispatch. Just-in-time Java [37] compilers use dynamic compilation to quickly produce native code from bytecodes. In both of these cases, the programmer is unaware that dynamic compilation is taking place and has no control over the optimization process.

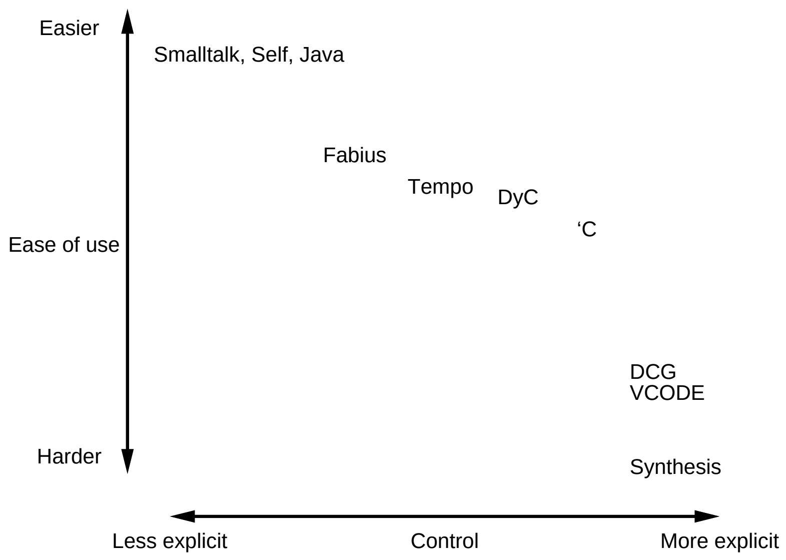

Figure 1-5: Partial design space for dynamic code generation systems.

At the other end of the spectrum are systems that provide instruction-level control of dynamic code. Given the low level of abstraction, these are all inherently difficult to use. The Synthesis kernel [62], for example, used manually generated dynamic code to improve scheduling and I/O performance, but required a heroic programming effort. More recently, Engler developed two libraries, DCG [25] and VCODE [22], that make manual dynamic code generation simpler and more portable. DCG allows the user to specify dynamic code as expression trees. VCODE provides a clean RISC-like interface for specifying instructions, but leaves register allocation and most instruction scheduling to the user—it is the closest existing approximation to a dynamic assembly language.

In between these two extremes, the center of the design space is occupied by high-level programming systems that expose dynamic code generation to the programmer. They all enable a high degree of control while remaining relatively easy to use. In general, however, as the amount of control increases, the ease of use decreases. At one end of this range is the 'C language, which is discussed in detail in Chapter 2 and has also been described elsewhere [23, 56, 57]. 'C is an extension of ANSI C that enables imperative dynamic code generation: the programmer can write fragments of C that denote dynamic code and then explicitly manipulate them to create large and complex dynamic code objects. 'C gives a vast amount of control over the compilation process and the structure of dynamic code, but it requires a learning curve.

Other current high-level dynamic compilation systems adopt a declarative approach. Two such systems have been developed at the University of Washington [4, 38, 39]. The first UW compiler [4] provided a limited set of annotations and exhibited relatively poor performance. That system performed data flow analysis to discover all derived run-time constants given the run-time constants specified by the programmer. The second system, DyC [38, 39], provides a more expressive annotation language and support for several features, including polyvariant division (which allows the same program point to be analyzed for different combinations of run-time invariants), polyvariant specialization (which allows the same program point to be dynamically compiled multiple times, each specialized to different values of a set of run-time invariants), lazy specialization, and interprocedural specialization. Unlike 'C, DyC does not provide mechanisms for creating functions and function calls with dynamically determined numbers of arguments. Nonetheless, DyC's features allow it to achieve levels of functionality similar to 'C, but in a completely different programming style. Depending on the structure of the dynamic code, DyC generates code as quickly as or a little more quickly than tcc. As illustrated in Figure 1-5, DyC gives less control over dynamic compilation than ‘C, but it allows easier specification of dynamic optimizations.

Tempo [14], a template-based dynamic compiler derived from the GNU C compiler, is another C-based system driven by user annotations. It is similar to DyC, but provides support for only function-level polyvariant division and specialization, and does not provide means of setting policies for division, specialization, caching, and speculative specialization. In addition, it does not support specialization across separate source files. Unlike DyC, however, it performs conservative alias and side-effect analysis to identify partially static data structures. The performance data indicate that Tempo's cross-over points tend to be slightly worse than DyC, but the speedups are comparable: Tempo generates code of comparable quality, but more slowly. Tempo does not support complex annotations and specialization mechanisms, so it is both easier to use and less expressive than DyC. The Tempo project has targeted ‘C as a back end for its run-time specializer.

Fabius [51] is another declarative dynamic compilation system. It is based on a functional subset of ML rather than on C. It uses a syntactic form of currying to allow the programmer to express run-time invariants. Given these hints about run-time invariants, Fabius performs dynamic compilation and optimization—including copy propagation and dead code elimination—automatically. Fabius achieves fast code generation speeds, but it does not allow direct manipulation of dynamic code and provides no annotations for controlling the code generation process. Thus, it is both the easiest to use and the least flexible of the high-level systems we have considered so far.

The Dynamo project [50]—not pictured in Figure 1-5—is a successor to Fabius. Leone and Dybvig are designing a staged compiler architecture that supports different levels of dynamic optimization: emitting a high-level intermediate representation enables “heavyweight” optimizations to be performed at run time, whereas emitting a low-level intermediate representation enables only “lightweight” optimizations. The eventual goal of the Dynamo project is to build a system that will automatically perform dynamic optimization.

In summary, employing ease of use and degree of programmer control as metrics, we can divide dynamic compilation systems into roughly three classes: those, like Self and Java implementations, that use dynamic compilation as an implicit optimization, without programmer intervention; those, like VCODE, that give the programmer instruction-level—and therefore often excessively complicated—control of code generation; and a “middle ground” of programming systems that expose dynamic compilation to the programmer at a high level. ‘C, the language described in this thesis, falls into this intermediate category.

1.4.2 Compilation Overhead and Programmer Control

Another way to dissect the dynamic compilation design space is to look at code generation overhead. Figure 1-6 uses the same horizontal axis as Figure 1-5, but the vertical axis measures the overhead of dynamic code generation for each system.

As before, the Self implementation is in the top left corner. Unlike most other systems, Self uses run-time type profiling to identify optimization candidates. It then performs aggressive dynamic optimizations such as inlining and replacing any old activation records on the stack with equivalent ones corresponding to the newly optimized code. However, Self is a relatively slow purely object-oriented dynamically typed language, and dynamically optimized code can be almost twice as fast as unoptimized code. As a result, the overhead of dynamic compilation is easily amortized.

Java just-in-time compilers cover a wide area of the spectrum; their overhead depends very much on the amount of optimization that they perform. Naive translation of bytecodes to machine code requires a few tens or hundreds of cycles per instruction; more ambitious optimizations can require much more.

At the other end of the design space, low-level systems like VCODE or the dynamic code generation parts of the Synthesis kernel generate code in under fifty cycles per instruction.

Not unlike Java JITs, the ‘C implementation spans a wide range of code generation overhead. The one-pass dynamic run-time system based on VCODE generates code of fair quality in under 100 cycles per instruction. The more elaborate run-time system, ICODE, generates better code but incurs about six times more overhead.

The declarative systems have varying amounts of overhead, but they tend to be at the low end

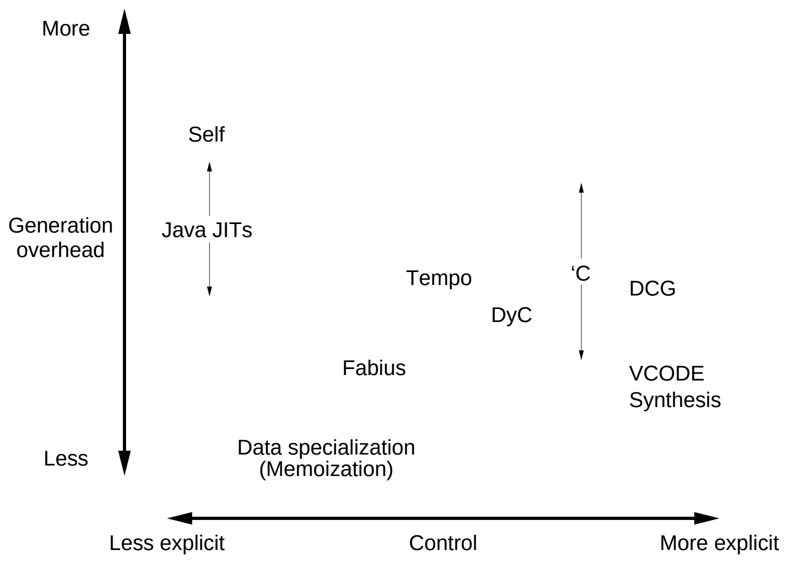

Figure 1-6: Another view of the dynamic code generation design space.

of the ‘C overhead spectrum. Fabius, which only provides limited annotations, is the fastest system, but DyC, which supports a large set of annotations, also generates code in well under 100 cycles per instruction. Such performance is not simply an artifact of specific implementations, but a side-effect of these systems' declarative programming interface. Declarative annotations allow the static compiler to predict the structure of dynamic code more easily than imperative constructs. They therefore increase the amount of compilation work that can be done at static compile time.

A final interesting point in this design space is data specialization [48]—essentially, compiler-driven memoization. Unlike any of the previous systems, data specialization does not involve dynamic code generation. Most of the dynamic compilation systems described above have the following type signature:

$$\mathit{Code} \times \mathit{Annotations} \rightarrow (\mathit{Input}_{\mathit{fix}} \rightarrow (\mathit{Input}_{\mathit{var}} \rightarrow \mathit{Result}))$$

In other words, the static code and dynamic code annotations—or imperative dynamic code constructs—are compiled statically into a program that takes a fixed (run-time constant) set of inputs and dynamically generates code that takes some varying inputs and produces a result. At a high level, dynamic compilation is really dynamic partial evaluation.

By contrast, data specialization relies on memoization of frequently used values. As shown below, the source code and annotations are compiled to code that stores the results of run-time constant computations in a cache; the values in the cache are combined with non-run-time constant values to obtain a result.

$$\mathit{Code} \times \mathit{Annotations} \rightarrow (\mathit{Input}_{\mathit{fix}} \rightarrow \mathit{Cache}) \times (\mathit{Input}_{\mathit{var}} \times \mathit{Cache} \rightarrow \mathit{Result})$$

This approach is simple and fast. Reads and writes to the cache are planned and generated statically, and the dynamic overhead of cache accesses is limited to a pointer operation. Of course, it is not as flexible as true dynamic compilation, but it can be very effective. Knoblock and Ruf report speedups as high as 100x [48]. Sometimes it is just not necessary to generate code dynamically!

1.5 Outline

This thesis is divided into three parts. First, Chapter 2 discusses the design of the ‘C language, including its most serious problems and potential extensions and improvements. Second, Chapter 3 presents applications of dynamic code generation. It describes the optimizations that are enabled by dynamic compilation and the factors that limit its applicability, and provides many examples. It is complemented by Appendix A, which studies uses of dynamic compilation in a handful of specific application areas. Third, the rest of the thesis is concerned with implementation. Chapter 4 gives an overview of the tcc implementation of ‘C and evaluates its overall performance on several benchmarks. Chapters 5, 6, and 7 discuss and evaluate in more detail specific aspects of the implementation, including register allocation and code improvement algorithms.

Chapter 2

Language Design

The usability and practicality of dynamic code generation depend to a large degree on the interface through which it is provided. Early implementations of dynamic code generation were complex and not portable. The natural solution to these problems is a portable library or language extension that provides dynamic code generation functionality. The library approach is appealing because it is simple to implement and can be adapted to provide excellent performance. For example, the VCODE [22] library is implemented as a set of C macros that generate instructions with an overhead of just a handful of instructions per generated instruction.

Libraries, however, have serious drawbacks. A library executes completely at run time: systems more complicated than VCODE, which just lays out bits on the heap, can easily incur high overhead. A language, on the other hand, allows the compiler to perform some part of dynamic compilation at static compile time. Declarative systems such as DyC [39] are particularly effective at staging compilation in this manner. In addition to performance, there is also a usability argument. A library cannot interact with the type system, which limits its ability to check for errors statically. Furthermore, the interface of any library is likely to be less flexible, succinct, and intuitive than native language constructs. It is difficult to imagine a library that provides complete code generation flexibility but does not expose a low-level ISA-like interface similar to VCODE's. In contrast, the possibility to express dynamic code in a language almost identical to that for static code should result in large productivity and ease-of-use gains relative to library-based systems.

As a result, a language-based approach is the one most likely to provide an efficient and portable interface to dynamic code generation. Language extensions themselves can be either imperative or declarative. The main advantages of the declarative style are that annotations preserve the semantics of the original code, and that it enables more aggressive optimizations at static compile time. 'C's choice of an imperative approach, on the other hand, can provide greater control over the dynamic compilation process, and makes dynamic compilation a programming tool in addition to simply an optimization tool. The 'C approach is familiar to people who have programmed in languages that allow code manipulation, such as Lisp and Perl. 'C grew out of experience with such languages; its very name stems from the backquote operator used in the specification of Lisp list templates [23].

This chapter describes 'C, discusses several subtleties and difficulties of its design, and presents some potentially useful extensions.

2.1 The 'C Language

'C [23] grew from the desire to support easy-to-use dynamic code generation in systems and applications programming environments. C was therefore the natural starting point on which to build the dynamic compilation interface. This choice motivated some of the key features of the language:

- 'C is a small extension of ANSI C: it adds few constructs and leaves the rest of the language intact. As a result, it is possible to convert existing C code to 'C incrementally.

- ‘C shares C’s design philosophy: the dynamic code extensions are relatively simple, and not much happens behind the programmer’s back. To a large degree, the programmer is in control of the dynamic compilation process. This desire for control is a significant reason for ‘C’s imperative approach to dynamic compilation.

- Dynamic code in ‘C is statically typed. Static typing is consistent with C and improves the performance of dynamic compilation by eliminating the need for dynamic type checking. The same constructs used to extend C with dynamic code generation should be applicable to other statically typed languages.

In ‘C, a programmer can write *code specifications*, which are static descriptions of dynamic code. These code specifications differ from the specifications used in software engineering and verification; they are simply fragments of C that describe (“specify”) code that will be generated at run time. Code specifications can capture the values of run-time constants, and they may be *composed* at run time to build larger specifications.

The compilation process in ‘C has three phases. At *static compile time* the ‘C compiler processes the written program and generates code-generating code for each code specification. Dynamic compilation, which occurs when the program runs, can be broken down into two phases. First, during *environment binding time*, code specifications are evaluated and capture their run-time environment. Then, during *dynamic code generation time*, a code specification is passed to ‘C’s compile special form, which generates executable code for the specification and returns a function pointer.

2.1.1 The Backquote Operator

The backquote, or “tick”, operator (‘) is used to create dynamic code specifications. It can be applied to an expression or compound statement and indicates that the corresponding code should be generated at run time. A backquote expression can be dynamically compiled using the compile special form, described in Section 2.1.4. For example, the following code fragment implements “Hello World” in ‘C:

```
void make_hello(void) {
      void (*f)() = compile(‘{ printf("hello, world"); }, void);
      (*f)();
}
```

The code within the backquote and braces is a specification for a call to printf that should be generated at run time. The code specification is evaluated (environment binding time), and the resulting object is then passed to compile, which generates executable code for the call to printf and returns a function pointer (dynamic code generation time). The function pointer can then be invoked directly.

‘C disallows the dynamic generation of code-generating code: the syntax and type system of C are already sufficiently complicated. As a result, backquote does not nest—a code specification cannot contain a nested code specification. This limitation appears to be of only theoretical interest: three years of experience with ‘C have uncovered no need for dynamically generated code-generating code.

Dynamic code is lexically scoped: variables in static code can be referenced in dynamic code. Lexical scoping and static typing allow type checking and some instruction selection to occur at static compile time, decreasing dynamic code generation overhead.

The use of several C constructs is restricted within backquote expressions. In particular, a break, continue, case, or goto statement cannot be used to transfer control outside the enclosing backquote expression. For instance, the destination label of a goto statement must be contained in the same backquote expression that contains the goto. ‘C provides other means for transferring control between backquote expressions; they are described in Section 2.1.4. The limitation on goto and other C control-transfer statements enables a ‘C compiler to statically determine whether such a statement is legal.

2.1.2 Type Constructors

'C introduces two new type constructors, cspec and vspec. cspecs are static types for dynamic code (the contents of backquote expressions), whereas vspecs are statics types for dynamic lvalues (expressions that may be used on the left-hand side of an assignment). Their presence allows most dynamic code to be type-checked statically.

A cspec or vspec has an associated *evaluation type*, which is the type of the dynamic value of the specification. The evaluation type is analogous to the type to which a pointer points.

cspec **Types**

cspec (short for *code specification*) is the type of a dynamic code specification; the evaluation type of the cspec is the type of the dynamic value of the code. For example, the type of the expression '4 is int cspec. The type of any dynamic statement or compound statement is void cspec. This type is also the generic cspec type (analogous to the generic pointer void *).

Here are some simple examples of cspec:

```
int cspec expr1 = '4;            /* Code specification for expression "4" */
float x;
float cspec expr2 = '(x + 4.);  /* Capture free variable x: its value will be
                                    bound when the dynamic code is executed */

/* All dynamic compound statements have type void cspec, regardless of
   whether the resulting code will return a value */
void cspec stmt = '{ printf("hello world"); return 0; };
```

vspec **Types**

vspec (short for *variable specification*) is the type of a dynamically generated lvalue, a variable whose storage class (whether it should reside in a register or on the stack, and in what location exactly) is determined dynamically. The evaluation type of the vspec is the type of the lvalue. void vspec is the generic vspec type. Objects of type vspec may be created by invoking the special forms local and param. local is used to create a new local variable for the function currently under construction. param is used to create a new parameter; param and vspecs allows us to construct functions with dynamically determined numbers of arguments. See Section 2.1.4 for more details on these special forms.

With the exception of the unquoting cases described in Section 2.1.3 below, an object of type vspec is automatically treated as a variable of the vspec's evaluation type when it appears inside a backquote expression. A vspec inside a backquote expression can thus be used like a traditional C variable, both as an lvalue and an rvalue. For example, the following function creates a cspec for a function that takes a single integer argument, adds one to it, and returns the result:

```
void cspec plus1(void) {
    /* Param takes the type and position of the argument to be generated */
    int vspec i = param(int, 0);
    return '{ int k = 1; i = i + k; return i; };
}
```

Explicitly declared vspecs like the one above allow dynamic local variables to span the scope of multiple backquote expressions. However, variables local to just one backquote expression, such as k in the previous code fragment, are dynamic variables too. Any such variable is declared using only its evaluation type, but it takes on its full vspec type if it is unquoted. Sections 2.1.3 and 2.2.2 below discuss vspecs and unquoting in more detail.

2.1.3 Unquoting Operators

The backquote operator is used to specify dynamic code. ‘C extends C with two other unary prefix operators, @ and $, which control the binding time of free variables within backquote expressions. These two operators are said to “unquote” because their operands are evaluated early, at environment binding time, rather than during the run time of the dynamic code. The unquoting operators may only be used within backquote expressions, and they may not themselves be unquoted: in other words, like the backquote operator, they do not nest.

The @ Operator

The @ operator denotes composition of cspecs and vspecs. The operand of @ must be a cspec or vspec. @ “dereferences” its operand at environment binding time: it returns an object whose type is the evaluation type of @’s operand. The returned object is incorporated into the cspec in which the @ occurs. Therefore, it is possible to compose cspecs to create larger cspecs:

```
void cspec h = ‘{ printf("hello "); };
void cspec w = ‘{ printf("world"); };
h = ‘{ @h; @w; }; /* When compiled and run, prints “hello world” */
```

Similarly, one may incorporate vspecs and build up expressions by composing cspecs and vspecs. Here is another take on our old friend, plus1:

```
void cspec plus1(void) {
    int vspec i = param(int, 0); /* A single integer argument */
    int cspec one = ‘1; /* Code specification for the expression “1” */
    return ‘{ @i = @i + @one; return @i; };
}
```

Applying @ to a function causes the function to be called at environment binding time. Of course, the function must return a cspec or a vspec; its result is incorporated into the containing backquote expression like any other cspec or vspec.

Dynamic code that contains a large number of @s can be hard to read. As a result, ‘C provides implicit coercions that usually eliminate the need for the @ operator. These coercions can be performed because ‘C supports only one level of dynamic code generation (dynamic code cannot generate more dynamic code), so it is always possible to determine whether a cspec or vspec object needs to be coerced to its evaluation type. An expression of type vspec or cspec that appears inside a quoted expression is always coerced to an object of its corresponding evaluation type unless (1) it is itself inside an unquoted expression, or (2) it is being used as a statement. The first restriction also includes implicitly unquoted expressions; that is, expressions that occur within an implicitly coerced expression are not implicitly coerced. For example, the arguments in a call to a function returning type cspec or vspec are not coerced, because the function call itself already is. The second restriction forces programmers to write h = ‘{ @h; @w; } rather than h = ‘{ h; w; }. The latter would be confusing to a C programmer, because it appears to contain two expression statements that have no side-effects and may therefore be eliminated.

Thanks to implicit coercions, we can rewrite the last version of plus1 more cleanly:

```
void cspec plus1(void) {
    int vspec i = param(int, 0); /* A single integer argument */
    int cspec one = ‘1; /* Code specification for the expression “1” */
    return ‘{ i = i + one; return i; }; /* Note the implicit coercions */
}
```

The $ Operator

The $ operator allows run-time values to be incorporated as *run-time constants* in dynamic code. The operand of $ can be any object not of type vspec or cspec. The operand is evaluated at environment binding time, and its value is used as a run-time constant in the containing cspec. The following code fragment illustrates its use:

```
int x = 1;
void cspec c = `{ printf("$x = %d, x = %d", $x, x); };
x = 14;
(*compile(c, void))(); /* Compile and run: will print "$x = 1, x = 14". */
```

The use of $ to capture the values of run-time constants can enable useful code optimizations such as dynamic loop unrolling and strength reduction of multiplication. Examples of such optimizations appear in Chapter 3.

2.1.4 Special Forms

'C extends ANSI C with several special forms. Most of these special forms take types as arguments, and their result types sometimes depend on their input types. The special forms can be broken into four categories, as shown in Table 2.1.

Category	**Name**	**Synopsis**
Management of dynamic code	compile	T (*compile(void cspec *code*, T))()
	free_code	void free_code(T)
Dynamic variables	local	T vspec local(T)
Dynamic function arguments	param	T vspec param(T, int *param-num*)
	push_init	void cspec push_init(void)
	push	void push(void cspec *args*, T cspec *next-arg*)
Dynamic control flow	label	void cspec label()
	jump	void cspec jump(void cspec *target*)
	self	T self(T, *other-args...*)

Table 2.1: The 'C special forms (T denotes a type).

Management of Dynamic Code

The compile and free_code special forms are used to create executable code from code specifications and to deallocate the storage associated with a dynamic function, respectively. compile generates machine code from *code*, and returns a pointer to a function returning type T. It also automatically reclaims the storage for all existing vspecs and cspecs.

free_code takes as argument a function pointer to a dynamic function previously created by compile, and reclaims the memory for that function. In this way, it is possible to reclaim the memory consumed by dynamic code when the code is no longer necessary. The programmer can use compile and free_code to explicitly manage memory used for dynamic code, similarly to how one normally uses malloc and free.

Dynamic Variables

The local special form is a mechanism for creating local variables in dynamic code. The objects it creates are analogous to local variables declared in the body of a backquote expression, but they can be used *across* backquote expressions, rather than being restricted to the scope of one expression. In addition, local enables dynamic code to have an arbitrary, dynamically determined number of local variables. local returns an object of type T vspec that denotes a dynamic local variable of type T in the current dynamic function. In 'C, the type T may include one of two C storage class specifiers, auto and register: the former indicates that the variable should be allocated on the stack, while the latter is a hint to the compiler that the variable should be placed in a register, if possible.

Dynamic Function Arguments

The param special form is used to create parameters of dynamic functions. param returns an object of type T vspec that denotes a formal parameter of the current dynamic function. *param-num* is the parameter's position in the function's parameter list, and T is its evaluation type. param can be used to create a function that has the number of its parameters determined at run time:

```
/* Construct cspec to sum n integer arguments. */
void cspec construct_sum(int n) {
    int i, cspec c = ‘0;
    for (i = 0; i < n; i++) {
        int vspec v = param(int, i); /* Create a parameter */
        c = ‘(c + v); /* Add param ’v’ to current sum */
    }
    return ‘{ return c; };
}
```

push_init and push complement param: they are used together to dynamically build argument lists for function calls. push_init returns a cspec that corresponds to a new (initially empty) dynamic argument list. push adds the code specification for the next argument, *next-arg*, to the dynamically generated list of arguments *args*. The following function creates a code specification for a call to the function created by construct_sum above. The number of arguments that the function takes and with which it is called is determined at run time by the variable nargs.

```
int cspec construct_call(int nargs, int *arg_vec) {
    int (*sum)() = compile(construct_sum(nargs), int);
    void cspec args = push_init();    /* Initialize argument list */
    int i;
    for (i = 0; i < nargs; i++)       /* For each arg in arg_vec... */
        push(args, ‘$arg_vec[i]);     /* push it onto the args stack */
    return ‘($sum)(args);             /* Bind the value of sum, so it is
                                         available outside construct_call */
}
```

Dynamic Control Flow

To limit the amount of error checking that must be performed at dynamic compile time, ‘C forbids goto statements from transferring control outside the enclosing backquote expression. Two special forms, label and jump, are used for inter-cspec control flow: jump returns the cspec of a jump to its argument, *target*. *Target* may be any object of type void cspec. label simply returns a void cspec that may be used as the destination of a jump. Syntactic sugar allows jump(target) to be written as jump target. Section 3.1.1 presents example ‘C code that uses label and jump to implement specialized finite state machines.

Lastly, self allows recursive calls in dynamic code without incurring the overhead of dereferencing a function pointer. T denotes the return type of the function that is being dynamically generated. Invoking self results in a call to the function that contains the invocation, with *other-args* passed as the arguments. self is just like any other function call, except that the return type of the dynamic function is unknown at environment binding time, so it must be provided as the first argument. For instance, here is a simple recursive dynamic code implementation of factorial:

```
void cspec mk_factorial(void) { /* Create a cspec for factorial(n) */
    unsigned int vspec n = param(unsigned int, 0); /* Argument */
    return ‘{ if (n > 2) return n * self(unsigned int /* Return type */, n−1);
              else return n;
            };
}
```

2.2 Subtleties and Pitfalls

C is a small and powerful language whose programming abstractions are closely related to the underlying hardware of most computers. C does not have a completely defined semantics—much of its definition depends on machine-specific hardware details or is implementation-dependent. Writing C code that is portable and correct requires understanding the details of C's definition and imposing stylistic conventions that are usually unchecked by the compiler.

'C adds functionality to C, but it also adds complexity and the potential for subtle errors. Three aspects of the language are most confusing: its dynamic compilation model, its unquoting and implicit promotion rules, and the details of its interaction with core C.

2.2.1 The Dynamic Compilation Model

Only one dynamic function can be constructed in 'C at a given time. All cspecs and vspecs evaluated before an invocation to compile are considered part of the function to be returned from compile. After dynamic code generation time (i.e., after compile is called) all these cspecs and vspecs are deallocated. Referring to them in subsequent backquote expressions, or trying to recompile them again, results in undefined behavior. Some other dynamic compilation systems [14, 51] memoize dynamic code fragments. In 'C, however, the low overhead required to create code specifications, their generally small size, and their susceptibility to changes in the run-time environment make memoization unattractive. Here is a simple example of what can go wrong when a programmer attempts to reuse code specifications:

```
void cspec hello = `{ printf("hello"); };
void cspec unused = `{ printf("unused"); };
void (*f)() = compile(hello, void); /* f prints "hello" and returns nothing */
void (*g)() = compile(unused, void); /* Undefined runtime error! "unused" was deallocated
                                        by the previous compile, even though it was never used */
```

Not all errors are this obvious, though. Here is a tougher example:

```
void cspec make_return_const(int n) { return `{ return $n; }; }
void cspec make_print(int n) {
    int (*f)() = compile(make_return_const(n), int);
    return `{ printf("%d", ($f)()); };
}
int main() {
    void (*f)() = compile(make_print(77), void);
    f(); /* This prints "77", as one would expect ... */
        /* ... but the behavior of this is undefined ... */
    f = compile(`{ @make_print(77); }, void);
        /* ... because compile inside make_print destroys the backquote expression above */
}
```

The second call to compile is undefined because its first argument is a backquote expression that is destroyed (similarly to unused in the previous example) by the call to compile inside make_print. One must take care to consider this sort of effect when writing 'C code.

There are two alternatives to this behavior. The first is to keep the one-function-at-a-time model, but to track allocated and deallocated cspecs and vspecs in the run-time system so as to provide informative error messages. The second is to allow arbitrarily interleaved specification of dynamic code, by associating each cspec and vspec and each call to compile with a "compilation context." Both options require additional run-time overhead, and the second option would also make 'C programs substantially clumsier and syntactically more complicated than they are. Given the performance-centric and minimalist philosophy of C and 'C, it appears reasonable to provide no run-time check and to allow users to learn the hard way.

This style of minimal run-time checking and maximal user control extends to other aspects of the language. For instance, just as in C, the value of a variable after its scope has been exited is undefined. In contrast to ANSI C, however, not all uses of variables outside their scope can be detected statically. Consider the following:

```
int (*make_nplus2(int n))() {
    void cspec c = `{ return n + 2; }; /* Should use $n instead of n */
    return compile(c, int);
}
int main() {
    int (*f)() = make_nplus2(4);
    printf("%d", f()); /* May not print "6" */
}
```

The code created by compile references n, which no longer exists when f is invoked inside main. In this toy example, of course, the right solution would be to make n a run-time constant inside the backquote expression. In general, though, there are several ways to detect errors of this kind. First, the compiler could perform a data-flow analysis to conservatively warn the user of a potential error. Unfortunately, such checks are likely to find many false positives. Second, the run-time system could tag dynamic code with scope information and issue an error at run time if an out-of-scope use occurs. Accurate run-time detection with no false positives would require tagging free variables and inserting tag checks in dynamic code, a prohibitively expensive proposition. Third, and even less attractive, ‘C could support “upward funargs” and a Scheme-like environment model. This approach completely breaks the stack-based model of C. In balance, providing no checks and requiring care of the programmer is probably the most reasonable—and definitely the most C-like—solution.

A similar issue arises with values returned by dynamic functions. As mentioned previously, all dynamic statements and compound statements have type void cspec. If a backquote expression contains a return statement, the type of the return value does not affect the type of the backquote expression. Since all type checking is performed statically, it is possible to compose backquote expressions to create a function at run time with multiple incompatible return types. Again, ‘C could require run-time consistency checks, but such a decision would be at odds with the language’s philosophy of minimal run-time activity. A better alternative would be to extend static typing of dynamic code to differentiate dynamic expressions and compound statements. For instance, a dynamic expression with evaluation type int might have full type int espec (an “expression specification”), a backquoted compound statement containing return statements of type int might have full type int cspec (a “compound statement specification”), and a compound statement with no returns might have full type void cspec. Only especs and compound statements of type void cspec or T cspec could be composed into objects of type T cspec; especs could be composed with each other arbitrarily using implicit conversions. We have considered various schemes like this one, but have not implemented them. They would eliminate the problem of incompatible return types, but they might also make ‘C more complicated and difficult to read.

2.2.2 Unquoting

Another subtle aspect of ‘C concerns unquoting and the sequence of dynamic compilation phases—environment binding time, dynamic code generation time, and finally the actual run time of the dynamic code. Specifically, within an unquoted expression, vspecs and cspecs cannot be passed to functions that expect their evaluation types. Consider the following code:

```
void cspec f(int j);
int g(int j);
void cspec h(int vspec j);
int main() {
    int vspec v;
    void cspec c1 = `{ @f(v); }; /* Error: f expects an int, and v is unquoted */
    void cspec c2 = `{ $g(v); }; /* Error: g expects an int, and v is unquoted */
    void cspec c3 = `{ int v; @f(v); }; /* Error: f expects an int, and v is unquoted */
    void cspec c4 = `{ int v; @h(v); }; /* OK: h expects a vspec, and v is unquoted */
    void cspec c5 = `{ int v; g(v); }; /* OK: g expects an int, and v is not unquoted */
}
```

Both f and g above expect an integer. In the first two backquote expressions, they are both unquoted—f by the cspec composition operator @, g by the run-time constant operator $—so they are both called (evaluated) at environment binding time. But at environment binding time v is not yet an integer, it is only the *specification* of an integer (an int vspec) that will exist after dynamic code generation time (i.e., after compile is invoked).

Of course, the same holds for variables local to a backquote expression, as illustrated by c3 in the example. By contrast, the code assigned to c4 is correct, because h expects a vspec. The code assigned to c5 is also correct, because the call to g is not unquoted—in other words, it happens at dynamic code run time.

Fortunately, all these conditions can be detected statically. Things get even more tricky and unexpected when we try to walk the line between environment binding and dynamic compilation time by using pointers to dynamic code specifications. Consider the following code (@ operators have been added for clarity, but the coercions to evaluation type would happen automatically):

```
int main() {
    int vspec av[3], vspec *v; /* Array of int vspecs, pointer to int vspec */
    void cspec c;
    av[0] = local(int); av[1] = local(int); av[2] = local(int);
    v = av;
    c = `{ @*v = 1; @*(v+1) = 0; @*(v+2) = 0; /* Dereferences happen at environment binding time */
          v++; /* Increment happens at dynamic-code run time */
          (@*v)++; /* Dereference at environment binding, increment at dynamic run time */
          printf("%d %d", @*v, @*(v+1)); /* Will print "2 0" */
        };
      ⋮
}
```

One might expect that, when compiled, the code specified by c would print "1 0". The reasoning would be this: the local variables denoted by v[0. . .2] are initialized to $(1, 0, 0)$; v is incremented; the variable it points to is incremented (so that the three variables now have values $(1, 1, 0)$); and finally the last two variables are printed ("1 0"). The error in this logic is that the increment of v does not occur until the run time of the dynamic code, whereas the unquoted dereference operations happen at environment binding time. As a result, @*v is incremented to 2 at dynamic code run time, and the other variables are left unchanged.

2.2.3 Interactions with C

Last but not least, we must consider some properties of C that influence `C programming, and unexpected ways in which `C extends C.

Function Argument Conversions

In ISO C, when an expression appears as an argument to a call to a function that does not have a prototype, the value of the expression is converted before being passed to the function [40]. For instance, floats are converted to doubles, and chars and shorts are converted to ints. The full set of conversions appears in [40].

In ‘C, dynamic functions currently do not have ISO prototypes. The reason for this is that the return type of compile(*cspec*, T) is “pointer to T function,” which is incompatible with a prototype such as “pointer to T function($T_{arg_1}, \ldots, T_{arg_n}$).”

It is easy to see what can go wrong if one is not careful. Consider the following code:

```
main() {
     float vspec p = param(float, 0);
     void cspec c = ‘{ printf("%f", p); };
     void (*f)() = compile(c, void);
     float x = 3.;
     f(x); /* This will probably not print "3.0"! */
}
```

f does not have a prototype, so x is converted from float to double before the call. Unfortunately, the dynamic code expects a float. On most machines, float and double have different sizes and representations, so f will probably not print “3.0”. The same effect, of course, can be obtained if one does not use prototypes in static code:

```
main() { float x = 3.; foo(x); }
foo(float x) { printf("%f", x); } /* Probably not "3.0" */
```

Therefore, the current ‘C user has two options. She must either use the same care as a user of traditional (non-ISO) C, invoking param only with the destination types of the function argument conversions, or she must cast the result of compile to a function pointer type that explicitly specifies the desired argument types. (The second option, of course, does not work for functions with a dynamically determined number of arguments.)

A better, but unimplemented, alternative to explicit casting might be to extend compile to accept information about the types of arguments of the dynamic function. For example, compile(c,int,float,char) would compile the cspec c and return a “pointer to int function(float, char).” This is less verbose than an explicit cast to a function pointer type, and it provides the same functionality. With minimal run-time overhead it could be extended to check that the prototype furnished to compile is compatible with that dynamically created by calls to param; C’s static casts cannot offer such dynamic error checking.

Dynamic Arrays

Consider the problem of creating a stack-allocated two-dimensional array whose size is specified dynamically. One way to do this is to write:

```
‘{ int a[$dim1][$dim2]; ... }
```

where dim1 and dim2 are the run-time constant array dimensions. In this case, a is only available within the containing backquote expression. Declaring a outside the backquote expression as an explicit dynamic array vspec would make it available across multiple backquote expressions, but it is not clear how to best write such a declaration. One option is the following:

```
int (vspec a)[dim1][dim2] = local(int[dim1][dim2]);
```

This is odd for C programmers, accustomed to seeing arrays defined with constant dimensions. Of course, dim1 and dim2 *are* constant within the dynamic code, but things can still look pretty strange:

```
int (vspec a)[dim1][dim2];
dim1++; dim2++;
a = local(int[dim1][dim2]);
c = `{ a[2][3] = 7; };
```

The dimension of the dynamic array changes from declaration to invocation of local! Furthermore, in general the static compiler can only check syntactic equivalence, and not value equivalence, of the dimension expressions in the array declaration and the call to local. The only feasible solution is to use the dimensions when local is called; the vspec object can then keep track of its own dimensions throughout the dynamic code. In fact, it appears less confusing to just allow dynamic array declarations with unspecified dimensions, and to make only the call to local responsible for specifying the dimensions:

```
int (vspec a)[][] = local(int[dim1][dim2]);
```

Unfortunately, this is still a little disconcerting, especially since the textually similar declaration of a two-dimensional array of vspecs with unspecified dimensions, int vspec a[][], is clearly illegal (in static C, one cannot declare an array with unspecified dimensions). The current ‘C implementation does not support explicit array vspecs.

2.3 Potential Extensions to ‘C

Programming in ‘C usually involves the manipulation of many small code specifications. Code specifications are composed at run time to build larger specifications before being dynamically compiled to produce executable code. This method allows great expressiveness with a minimal set of primitives, but it makes the creation of “common case” dynamic code more tedious than necessary.

This section proposes mechanisms to simplify common case programming situations. The mechanisms have not been implemented, but they could make ‘C more powerful and easier to use.

2.3.1 Syntactic Sugar for Dynamic Code

Although ‘C allows functions with a dynamically determined number of arguments, most dynamic code has a fixed number of arguments. ‘C programs would become easier to read if syntactic sugar allowed one to easily specify and name these fixed arguments.

The most intuitive approach is to provide a facility similar to the named-lambda available in some Lisp dialects:

```
void cspec c = `lambda f(int x, float y) { .... };
```

This statement introduces the names x and y within the body of the lambda, and it introduces the names f@x and f@y in the scope containing the backquote expression. f@x and f@y are not parts of the type of f—in fact, f does not even have a type, it is simply a label for the named-lambda. The code above would be just sugar for the following:

```
int vspec f@x = param(int, 0);
float vspec f@y = param(float, 1);
void cspec c = `{ .... };
```

This mechanism is convenient because it allows us write code like

```
return `lambda f(int y) { return y+$x; };
```

and

```
int (*add_x)() = compile(`lambda f(int y) { return y+$x; }, int);
```

The drawback of this scheme is that it adds names to lambda's enclosing scope in a way that circumvents the standard C declaration syntax. It makes it possible to declare new dynamic argument names at any point in the code, in a manner similar to C++. In the following code,

```
void cspec c;
if (cond) c = `lambda f(int i, float f) { .... };
else c = `lambda f(char c, double d) { .... };
```

the scope of f@i and f@f must be restricted to the first branch of the if, and that of f@c and f@d to the second branch, similarly to the following C++ code:

```
if (cond) int x = 3;  /* Scope of "int x" is "then" branch */
else float x = 4.; /* Scope of "float x" is "else" branch */
```

Unlike C++, however, "declarations" can occur inside expressions, too:

```
c = cond ? `lambda f(int i, float f) { .... } : `lambda f(char c, double d) { .... };
```

This statement introduces all four names, f@i, f@f, f@c, and f@d. However, depending on cond, only the first two or the last two are initialized by param—the same pitfall created by the current 'C param mechanism!

The simplest fix to this problem is to actually introduce the name of the lambda (in our example, f) in the containing scope. In this way, the above conditional expression becomes illegal because the third operand (the lambda after the colon) redeclares f.

Another solution[1] is to make lambda a declarator. One would then write:

```
int lambda f(int i, float f);
```

which would be equivalent to

```
void cspec f = `{}; /* In addition, "int lambda" asserts that f should return int */
int vspec f@i = param(int, 0);
float vspec f@f = param(float, 1);
```

This method has three benefits. First, it does not radically extend the style of C declarations. Second, it avoids the redeclaration issue illustrated by the conditional expression example above. Third, it can be extended to elegantly declare dynamic locals visible across backquote expressions without resorting to the local special form. Specifically, one could write: int f@i, or even int @i to specify a dynamic local, rather than int vspec i = local(int). One could remove vspec and local altogether, and only leave param as a way of creating functions with dynamically determined arbitrary numbers of arguments. The main disadvantage of such a style of declaration would be that it changes the semantics of a C declaration: programmers are used to the fact that declarations have only static effects, and that only initializers and statements have dynamic effects. The lambda declarator, as well as the declaration alternatives to local, require significant run-time activity. Also, it is probably best not to remove vspec and local, since together they decouple the declaration of a vspec name from the initialization of a vspec object, and thereby allow one name to refer to more than one dynamic location.

2.3.2 Declarative Extensions

'C's imperative programming model is powerful but often unduly complicated. However, 'C could be extended with declarative annotations that would simplify common optimizations while preserving the flexibility and compositional power of the imperative model.

[1]This idea is due to Eddie Kohler.

Fine-Grained Annotations

Run-time constants specified with the $ operator are at the center of 'C dynamic optimizations. They drive local optimizations such as strength reduction and dead code elimination. However, some versions of the implementation also implicitly perform more global optimizations that have a declarative flavor. For instance, if the bounds of a for loop are both run-time constants, and if the number of iterations is determined to be relatively small, then the loop is fully unrolled automatically.

One can easily imagine further extensions. As described in Chapter 4, a more sophisticated implementation could identify derived run-time constants (values that are run-time constants because others are) and stage aggressive dynamic optimizations. One could also extend 'C with annotations to describe and control optimization policies. In this way, one could quickly optimize traditional static code with annotations, as one does with DyC or Tempo, but 'C's expressive imperative constructs would still be available when necessary.

Partial Evaluation

It is often useful to dynamically partially evaluate a function with respect to one or two run-time constant arguments. A classic example is printf and its format string argument. In 'C this sort of optimization can be done manually by enclosing the whole function in a code specification and using different code and variable specifications for different combinations of specializing arguments. Some other projects [3, 14, 38] have addressed the issue of partial evaluation in C in depth. This section proposes a partial evaluation interface that is integrated into the 'C code specification model.

The method relies on one additional keyword, bound, and one special form, specialize (though maybe bind would be a more symmetrical name). As usual, we proceed by example:

```
int f(int x, float y); /* Prototype of f */
int f(bound int x, float y); /* Specialization prototype of f w.r.t. arg 1 */
int f(int x, bound float y); /* Specialization prototype of f w.r.t. arg 2 */

int f(int x, float y) { .... } /* Definition of f */
```

The above code contains the definition of some function f, an ISO prototype for f, and two "specialization prototypes" that contain the keyword bound. The first prototype refers to f specialized with respect to its first argument, the second refers to it specialized with respect to its second argument. The compiler records this information about the desired specializations of f. Then, when it encounters the definition of f, it generates both code for f and code-generating code for each of the specialized versions of f.

The specialize special form takes as input a function name and a list of arguments with respect to which the function should be specialized, and exposes the appropriate code-generating code to the programmer. Consider a function g with return type T_r and n arguments with types $T_{a_1}, \ldots, T_{a_n}$. Given this function, specialize returns a function pointer with return type T_r vspec and, for each unspecialized argument j of g, arguments of type T_{a_j} vspec.

Values to be used for specialization are expressed via partially empty argument lists. For instance, consider a function int h(char, int, float, double). If h has the specialization prototype int h(bound char, int, bound float, double), then the statement specialize(h('a', , x,)) specializes h with respect to its first argument equal to 'a' and its third argument equal to the run-time value of x. The second and fourth arguments, an int and a double, are left unspecified and must be provided as arguments at dynamic code run time. specialize therefore returns a function pointer of type int cspec (*)(int vspec, double vspec).

Returning to our example function f from above, we have the following:

```
int cspec (*f1)(float vspec) = specialize(f(1,)); /* Specialize w.r.t. arg 1 */
int cspec (*f2)(int vspec) = specialize(f(,1.0)); /* Specialize w.r.t. arg 2 */
int cspec (*x)() = specialize(f(1,1.0)); /* Error! */
```

The first call to specialize returns a pointer to a function that returns an int cspec (since f has type int) and takes a float vspec. This vspec corresponds to the unspecialized second argument of f. The first argument, on the other hand, is hard-wired in the body of the specialized int cspec. Similarly, the second call returns a pointer to a function that returns an int cspec, but whose argument is an int vspec, corresponding to the unspecialized first argument. The third call is an error, because there is no prototype int f(bound int, bound float). Note that the prototypes for f need to be in the scope of calls to specialize. Also, f must be the name of a function; it cannot be a function pointer because in general the pointer's value cannot be determined statically.

The fact that specialize returns a function that returns a cspec and takes vspecs as arguments would make it easy to integrate function-level specialization into 'C's dynamic compilation process. For instance, one would write this:

```
int cspec (*f1)(int vspec) = specialize(f(1,)); /* Specialize w.r.t. arg 1 = 1 */
int cspec (*fk)(int vspec) = specialize(f(k,)); /* Specialize w.r.t. arg 1 = k */
float vspec p = param(float, 0); /* The dynamic (unspecialized) argument */
int (*sum)(float) = (int (*)(float))compile(`{ return @f1(p) + @fk(p); });
```

The two specialized bodies of f returned by f1 and fk would be automatically inlined into the dynamic code that computes the sum, just like any other cspecs. If we had a function vf that took the same arguments as f but did not return a value, we could not compose it into an expression, but cspec composition would still work as usual:

```
void cspec (*vf1)(int vspec) = specialize(vf(1,)); /* Specialize w.r.t. arg 1 = 1; vf returns void */
float vspec p = param(float, 0); /* The dynamic argument */
void cspec c = `{ @vf1(p); /* Compose a cspec as usual */ };
```

And if one simply wanted to create a specialized version of f, without composing it with any other code, one could write something like the following:

```
int cspec (*f1)(int vspec) = specialize(f(1,)); /* Specialize w.r.t. arg 1 */
float vspec p = param(float, 0); /* Remaining dynamic (unspecialized) argument */
int (*f2)(float) = (int (*)(float))compile(`{ return f1(p); }, int);
```

Or, leveraging the named-lambda syntactic sugar in Section 2.3.1, one could more succinctly write:

```
int (*f2)(float) = (int (*)(float))compile(`lambda f3(float p) { return (*specialize(f(1,)))(p); }, int);
```

If such "stand-alone" specialization turned out to be common, one could easily wrap this rather convoluted syntax into more sugar, such as this:

```
int (*f2)(float) = specialize_to_func(f(1,));
```

This interface to partial evaluation is relatively flexible and would fit well with the rest of 'C. Its implementation, however, would be non-trivial: good specialized code performance would require encoding aggressive optimizations in the specialized code-generating functions. Grant et al. [38] present some techniques that may be useful in this context.

2.3.3 To Add or Not to Add?

Language design is a difficult art: it involves not only engineering considerations, but also aesthetic and social aspects. 'C in its present form may be somewhat complex to use, but it reflects its underlying computation model in a faithful and straightforward way. The extensions presented in this section may make 'C easier to use in the common case. They also make it "philosophically" more distant from C, in that they distance the user from the underlying implementation and remove some level of control. By contrast, they make 'C more similar to some of the declarative dynamic compilation systems for C [14, 38]. Insofar as 'C is useful in situations where one desires full control of dynamic compilation, the language is probably better left simple, free of these extensions. On the other hand, the extensions might serve to bridge the space between 'C and more high-level systems and to provide some of the features of both worlds. The usefulness of such a union, however, is really a social issue—it would require a lot more users of dynamic code generation to know for sure.

Chapter 3

Applications

Dynamic code generation can be useful in a number of practical settings. 'C, in particular, can both improve performance and simplify the creation of programs that would be difficult to write in plain ANSI C. The programmer can use 'C to improve performance by specializing code to run-time invariants, performing dynamic inlining, creating "self-traversing" dynamic data structures, and more. 'C also simplifies the creation of programs—such as interpreters, just-in-time compilers, and database query engines—that involve compilation or interpretation: the programmer can focus on parsing and semantics, while 'C automatically handles code generation.

This chapter illustrates techniques and coding styles that make 'C useful, identifies some of the limits to the applicability of dynamic code generation, and presents the benchmarks that are used to evaluate the 'C implementation later in this thesis. Appendix A complements this chapter by describing in some depth several applications of dynamic code generation.

3.1 A Taxonomy of Dynamic Optimizations

At the highest level, dynamic optimizations are effective because they make use of run-time information unavailable to traditional compilers. These optimizations can be classified into a handful of categories based on the compilation mechanisms that are used to implement them and the specific objectives that they aim to fulfill. Such categorizations are imprecise, because many categories overlap and any realistic use of dynamic compilation spans several categories. Nonetheless, this taxonomy is helpful when one needs to write dynamic code or adapt static code to use dynamic compilation.

To take advantage of 'C, one must first identify the objectives that dynamic compilation can help to achieve. There appear to be four fundamental goals:

1. **Specialization to specific data values.** This technique is the most intuitive application of dynamic code generation. It simply customizes a piece of code to one or more pieces of data that remain unchanged for some period of time. An interesting example is the self-searching binary tree in Section 3.1.1.

2. **Specialization to data types or sizes.** Sometimes data changes too rapidly or is reused too few times to allow specialization based on its value. Nonetheless, it may be possible to obtain performance benefits by customizing dynamic code to the particular shape or size of data structures. The customized copy and swap routines in Section 3.1.2 are simple examples of these optimizations.

   An important class of data type specialization is the optimization of byte vector marshaling and unmarshaling code. Such code is often slow because it requires interpreting a vector that describes the types of the data to be marshaled. 'C's facilities for constructing functions and function calls with dynamically determined types and numbers of arguments help to create marshaling code customized to frequently occurring data.

3. **Elimination of call overhead.** Inlining has been used for a long time to remove call overhead and enable interprocedural optimization when the callee can be determined statically. Unfortunately, function pointers make traditional inlining impossible. 'C can be used to perform a kind of dynamic function inlining, as exemplified by the Newton's method code in Section 3.1.3.

4. **Elimination of interpretation overhead.** Most of the previous points actually describe elimination of interpretation in some form or another—for instance, we reduce unmarshaling overhead by removing interpretation of an argument description vector. In some cases, however, static code involves more explicit levels of interpretation: execution of bytecode, traversal of complicated data structures, and so forth. Dynamic compilation can be used to strip out this overhead. For example, 'C makes it easy to write fast compiling interpreters (Section 3.1.4), executable data structures (Section 3.1.1), and database queries (Section 3.1.4).

To achieve these objectives, 'C relies on four main mechanisms and programming techniques:

1. **Hardwiring run-time constants.** This mechanism is the most straightforward use of the $ operator. It allows arithmetic and other operations based on run-time constants to be simplified, reduced in strength, or completely eliminated. A simple example is the specialized hash function in Section 3.1.1. Hardwiring run-time constants can also have more large-scale effects, such as dynamic dead code elimination and dynamic loop unrolling.

2. **Dynamic code construction.** Code construction is probably the most widely used dynamic optimization mechanism. It is employed whenever dynamic code is generated based on the interpretation of some data structure or the execution of some other code-generating kernel. It can be used to specialize code both to values, as in the binary search example in Section 3.1.1, and to particular types and data structures, as in the swap routine in Section 3.1.2. It is also useful when eliminating interpretation overhead, as in the compiling interpreter in Section 3.1.4.

3. **Dynamic inlining.** If we replace function pointers with cspecs, and function definitions with backquote expressions, we transparently obtain the equivalent of dynamic inlining across function pointers. This functionality frequently eliminates the overhead of calls through a function pointer, as illustrated in Section 3.1.3.

4. **Dynamic call construction.** To our knowledge, 'C is the only dynamic compilation system that allows programmers to create functions—and matching function calls—with run-time determined types and numbers of arguments. Section 3.1.2 shows how this feature can be used.

The rest of this section looks at each of the abovementioned objectives in more detail and provides examples of how 'C's optimization mechanisms can be used to achieve them.

3.1.1 Specialization to Data Values

If a set of values will be used repeatedly, dynamic compilation can serve to produce efficient code tailored to those values. The values can be used to enable traditional optimizations like strength reduction and dead-code elimination, or to create more exotic objects such as executable data structures.

Hashing

A simple example of 'C is the optimization of a generic hash function, where the table size is determined at run time, and where the function uses a run-time value to help its hash. Consider the C code in Figure 3-1. The C function has three values that can be treated as run-time constants: ht→hte, ht→scatter, and ht→norm. As illustrated in Figure 3-2, using 'C to specialize the function for these values requires only a few changes. The resulting code can be considerably faster than

```
struct hte {            /* Hash table entry structure */
    int val;            /* Key that entry is associated with */
    struct hte *next;   /* Pointer to next entry */
    /* ... */
};

struct ht {             /* Hash table structure */
    int scatter;        /* Value used to scatter keys */
    int norm;           /* Value used to normalize */
    struct hte **hte;   /* Vector of pointers to hash table entries */
};

/* Hash returns a pointer to the hash table entry, if any, that matches val. */
struct hte *hash(struct ht *ht, int val) {
    struct hte *hte = ht->hte[(val * ht->scatter) / ht->norm];
    while (hte && hte->val != val) hte = hte->next;
    return hte;
}
```

Figure 3-1: A hash function written in C.

```
/* Type of the function generated by mk_hash: takes a value as input
   and produces a (possibly null) pointer to a hash table entry */
typedef struct hte *(*hptr)(int val);

/* Construct a hash function with the size, scatter, and hash table pointer hard-coded. */
hptr mk_hash(struct ht *ht) {
    int vspec val = param(int, 0);
    void cspec code = `{
        struct hte *hte = ($ht->hte)[(val * $ht->scatter) / $ht->norm];
        while (hte && hte->val != val) hte = hte->next;
        return hte;
    };
    return compile(code, struct hte *); /* Compile and return the result */
}
```

Figure 3-2: Specialized hash function written in ‘C.

the equivalent C version, because tcc hardcodes the run-time constants hte, scatter, and norm in the instruction stream, and reduces the multiplication and division operations in strength. The cost of using the resulting dynamic function is an indirect jump on a function pointer.

Exponentiation

The last example used dynamic compilation to perform strength reduction and avoid memory indirection. We can be more ambitious and actively create dynamic code based on run-time values. For instance, in computer graphics it is sometimes necessary to apply an exponentiation function to a large data set [21]. Traditionally, exponentiation is computed in a loop which performs repeated multiplication and squaring. Given a fixed exponent, we can unroll this loop and obtain straight-line code that contains the minimum number of multiplications. The ‘C code to perform this optimization appears in Figure 3-3.

Matrix Scaling

The problem of scaling a matrix by an integer constant gives us the opportunity to combine the techniques from the previous two examples: hardwiring run-time constants to perform strength reduction and avoid memory indirection, and code construction to unroll loops. The routine in

```
typedef double (*dptr)();
dptr mkpow(int exp) {
     double vspec base = param(double, 0); /* Argument: the base */
     double vspec result = local(register double); /* Local: running product */
     void cspec squares;
     int bit = 2;

     /* Initialize the running product */
     if (1&exp) squares = `{ result=base; };
     else squares = `{ result=1.; };

     /* Multiply some more, if necessary */
     while (bit <= exp) {
          squares = `{ @squares; base *= base; };
          if (bit & exp) squares = `{ @squares; result *= base; };
          bit = bit << 1;
     }
     /* Compile a function which returns the result */
     return compile(`{ @squares; return result; }, double);
}
```

Figure 3-3: Code to create a specialized exponentiation function.

```
void cspec mkscale(int **m, int n, int s) {
     return `{
          int i, j;
          for (i = 0; i < $n; i++) { /* Loop can be dynamically unrolled */
               int *v = ($m)[i];
               for (j = 0; j < $n; j++)
                    v[j] = v[j] * $s; /* Multiplication can be strength-reduced */
          } };
}
```

Figure 3-4: ‘C code to specialize multiplication of a matrix by an integer.

Figure 3-4 generates dynamic code specialized to the pointer to the current matrix, the size of the matrix, and the scale factor. It is a simple benchmark of the potential effects of dynamic loop unrolling and strength reduction.

Vector Dot Product

Dynamic code generation can also be used to perform dead code elimination based on run-time data values. Consider the C code to compute dot product shown in Figure 3-5. If one of the vectors will be used several times, as in matrix multiplication, it may be useful to generate dynamic code customized to computing a dot product with that vector. The dynamic code can avoid multiplication by zeros, strength-reduce multiplication, and encode values from the run-time constant vector directly into the instruction stream. Figure 3-6 shows the ‘C code that implements these optimizations.

Binary Search

This style of value-specific dynamic code construction leads us naturally to more complex executable data structures—data structures whose contents are embedded into specialized code that implements a common traversal or extraction operation. Consider the recursive binary search code in Figure 3-7. Of course, an iterative version is more efficient, but it still incurs overhead and memory access overhead, and the recursive version is much more clear.

When the same input array will be searched several times, one can use ‘C to write code like that in Figure 3-8. The structure of this code is very similar to that of the recursive binary search. However,

```
void dot(int *a, int *b, int n) {
     int sum, k;
     for (sum = k = 0; k < n; k++) sum += a[k]*b[k];
     return sum;
}
```

Figure 3-5: A dot-product routine written in C.

```
void cspec mkdot(int row[], int n) {
     int k;
     int *vspec col = param(int *, 0); /* Input vector for dynamic function */
     int cspec sum = `0; /* Spec for sum of products; initally 0 */
     for (k = 0; k < n; k++) /* Only generate code for non-zero multiplications */
          if (row[k]) /* Specialize on index of col[k] and value of row[k] */
               sum = `(sum + col[$k] * $row[k]);
     return `{ return sum; };
}
```

Figure 3-6: ‘C code to build a specialized dot-product routine.

```
int bin(int *x, int key, int l, int u, int r) {
     int p;
     if (l > u) return −1;
     p = u − r;
     if (x[p] == key) return p;
     else if (x[p] < key) return bin(x, key, p+1, u, r/2);
     else return bin(x, key, l, p−1, r/2);
}
```

Figure 3-7: A tail-recursive implementation of binary search.

```
void cspec gen(int *x, int vspec key, int l, int u, int r) {
     int p;
     if (l > u) return `{ return −1; };
     p = u − r;
     return ` {
          if ($(x[p]) == key) return $p;
          else if ($(x[p]) < key) @gen(x, key, p+1, u, r/2);
          else @gen(x, key, l, p−1, r/2);
     };
}
```

Figure 3-8: ‘C code to create a “self-searching” executable array.

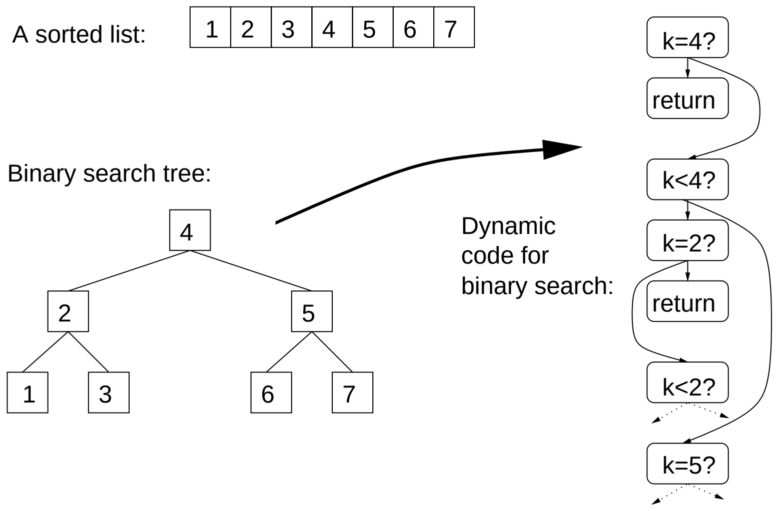

Figure 3-9: An executable data structure for binary search.

rather than searching the input array, it generates code customized to searching that specific array. The values from the input array are hardwired into the customized instruction stream, and the loop is unrolled into a binary tree of nested if statements that compare the value to be found to constants. As a result, the search involves neither loads from memory nor looping overhead, so the dynamically constructed code is considerably more efficient than its static counterpart. Figure 3-9 illustrates the relationship between data structure and executable data structure for the case of binary search on a specific seven-element array. For small input vectors (on the order of 30 or so elements), this technique results in lookup performance superior even to that of a hash table. A jump table could be faster, but it would only be practical if the values in the vector were relatively contiguous.

By adding a few dynamic code generation primitives to the original algorithm, we have created a function that returns a cspec for binary search that is tailored to a given input set. We can easily create a C function pointer from this cspec as follows:

```
typedef int (*ip)(int key);
ip mksearch(int n, int *x) {
    int vspec key = param(int, 0); /* One argument: the key to search for */
    return (ip)compile(gen(x, key, 0, n−1, n/2), int);
}
```

Finite State Machines

A sorted array is a simple data structure, but we can certainly specialize code for more complex data. For example, ‘C can be used to translate a deterministic finite automaton (DFA) description into specialized code, as shown in Figure 3-10. The function mk_dfa accepts a data structure that describes a DFA with a unique start state and some number of accept states: at each state, the DFA transitions to the next state and produces one character of output based on the next character in its input. mk_dfa uses ‘C’s inter-cspec control flow primitives, jump and label (Section 2.1.4), to create code that directly implements the given DFA: each state is implemented by a separate piece of dynamic code, and state transitions are simply conditional branches. The dynamic code contains no references into the original data structure that describes the DFA.

```
typedef struct {
    int n;                  /* State number (start state has n=0) */
    int acceptp;            /* Non-zero if this is an accept state */
    char *in;               /* I/O and next state info: on input in[k] */
    char *out;              /*   produce output out[k] and go to state */
    int *next;              /*   number next[k] */
} *state_t;
typedef struct {
    int size;               /* Number of states */
    state_t *states;        /* Description of each state */
} *dfa_t;

int (*mk_dfa(dfa_t dfa))(char *in, char *out) {
    char * vspec in = param(char *, 0); /* Input to dfa */
    char * vspec out = param(char *, 1); /* Output buffer */
    char vspec t = local(char);
    void cspec *labels = (void cspec *)malloc(dfa->size * sizeof(void cspec));
    void cspec code = `{};   /* Initially dynamic code is empty */
    int i;
    for (i = 0; i < dfa->n_states; i++)
        labels[i] = label();   /* Create labels to mark each state */
    for (i = 0; i < dfa->n_states; i++) { /* For each state ... */
        state_t cur = dfa->states[i];
        int j = 0;
        code = `{ @code;    /* ... prepend the code so far */
                @labels[i];  /* ... add the label to mark this state */
                t = *in; }; /* ... read current input */
        while (cur->in[j]) { /* ... add code to do the right thing if */
            code = `{ @code; /*      this is an input we expect */
                    if (t == $cur->in[j]) {
                        in++; *out++ = $cur->out[j];
                        jump labels[cur->next[j]];
                    } };
            j++;
        }
        code = `{ @code;    /* ... add code to return 0 if we're at end */
                if (t == 0) { /*   of input in an accept state, or */
                    if ($cur->acceptp) return 0;
                    else return -2; /* -2 if we're in another state */
                } /* or -1 if no transition and not end of input */
                else return -1; };
    }
    return compile(code, int);
}
```

Figure 3-10: Code to create a hard-coded finite state machine.

```
typedef void (*fp)(void *, void *);
fp mk_swap(int size) {
    long * vspec src = param(long *, 0); /* Arg 0: source */
    long * vspec dst = param(long *, 1); /* Arg 1: destination */
    long vspec tmp = local(long); /* Temporary for swaps */
    void cspec s = `{}; /* Code to be built up, initially empty */
    int i;

    for (i = 0; i < size/sizeof(long); i++) /* Build swap code */
        s = `{ @s; tmp = src[$i]; src[$i] = dst[$i]; dst[$i] = tmp; };
    return (fp)compile(s, void);
}
```

Figure 3-11: ‘C code to generate a specialized swap routine.

3.1.2 Specialization to Data Types

The examples so far have illustrated specialization to specific values. Value specialization can improve performance, but it may be impractical if the values change too frequently. A related approach that does not suffer from this drawback is specialization based on more general properties of data types, such as their size or structure.

Swap

It is often necessary to swap the contents of two regions of memory: in-place sorting algorithms are one such example. As long as the data being manipulated is no larger than a machine word, this process is quite efficient. However, when manipulating larger regions (e.g., C structs), the code is often suboptimal. One way to copy the regions is to invoke the C library memory copy routine, memcpy, repeatedly. Using memcpy incurs function call overhead, as well overhead within memcpy itself. Another way is to iteratively swap one word at a time, but this method incurs loop overhead.

‘C allows us to create a swap routine that is specialized to the size of the region being swapped. The code in Figure 3-11 is an example, simplified to handle only the case where the size of the region is a multiple of sizeof(long). This routine returns a pointer to a function that contains only assignments, and swaps the region of the given size without resorting to looping or multiple calls to memcpy. The size of the generated code—and therefore the cost of generating it—is usually quite small, which makes this a profitable optimization. We can obtain additional benefits by dynamically inlining the customized swap routine into the main sorting code.

Copy

Copying a memory region of arbitrary size is another common operation. It occurs frequently, for instance, in computer graphics [55]. Similarly to the previous code for swapping regions of memory, the example code in Figure 3-12 returns a function customized for copying a memory region of a given size.

The procedure mk_copy takes two arguments: the size of the regions to be copied, and the number of times that the inner copying loop should be unrolled. It creates a cspec for a function that takes two arguments, pointers to source and destination regions. It then creates prologue code to copy regions which would not be copied by the unrolled loop (if $n \bmod unrollx \neq 0$), and generates the body of the unrolled loop. Finally, it composes these two cspecs, invokes compile, and returns a pointer to the resulting customized copy routine.

```
typedef void (*fp)(void *, void *);
fp mk_copy(int n, int unrollx) {
    int i, j;
    unsigned * vspec src = param(unsigned *, 0); /* Arg 0: source */
    unsigned * vspec dst = param(unsigned *, 1); /* Arg 1: destination */
    int vspec k = local(int); /* Local: loop counter */
    cspec void copy = `{}, unrollbody = `{}; /* Code to build, initially empty */

    for (i = 0; i < n % unrollx; i++) /* Unroll the remainder copy */
        copy = `{ @copy; dst[$i] = src[$i]; };

    if (n >= unrollx) /* Unroll copy loop unrollx times */
        for (j = 0; j < unrollx; j ++)
            unrollbody = `{ @unrollbody; dst[k+$j] = src[k+$j]; };

    copy = `{ /* Compose remainder copy with the main unrolled loop */
            @copy;
            for (k = $i; k < $n; k += $unrollx) @unrollbody;
    };
    /* Compile and return a function pointer */
    return (fp)compile(copy, void);
}
```

Figure 3-12: Generating specialized copy code in ‘C.

Marshaling and Unmarshaling Code

Marhsaling and unmarshaling data into and out of byte vectors are important operations—for example, they are frequently used to support remote procedure call [8]. Unfortunately, both are expensive and clumsy to implement with static C code. Consider marshaling data into a byte vector. One option is to statically write one marshaling function for every combination of data types to be marshaled—efficient but unrealistic. Another option is to write a function with a variable number of arguments ("varargs"), whose first argument describes the number and type of the remaining ones. This approach is reasonable, but it is inefficient if the same sort of data is marshaled repeatedly.

Unmarshaling is even more awkward. Whereas C supports functions with variable numbers of arguments, it does not support function calls with variable numbers of arguments. As a result, at best, an unmarshaling function may use a switch statement to call a function with the right signature depending on the contents of the byte vector. If the byte vector contains data not foreseen by the switch statement, a consumer routine is forced to explicitly unmarshal its contents. For instance, the Tcl [54] run-time system can make upcalls into an application, but Tcl cannot dynamically create code to call an arbitrary function. As a result, it marshals all of the upcall arguments into a single byte vector, and forces applications to explicitly unmarshal them. If Tcl had access to a mechanism for constructing arbitrary upcalls, clients would be able to write their code as normal C routines, simplifying development and decreasing the chance of errors.

‘C helps to implement practical and efficient marshaling and unmarshaling code by providing support for functions and function calls with dynamically determined types and numbers of arguments. ‘C allows programmers to generate marshaling and unmarshaling routines customized to the structure of the most frequently occurring data. Specialized code for the most active marshaling and unmarshaling operations can significantly improve performance [72]. This functionality distinguishes ‘C: neither ANSI C nor any of the dynamic compilation systems discussed in Section 1.4 provides mechanisms for constructing function calls dynamically.

We present two functions, marshal and unmarshal, that dynamically construct marshaling and unmarshaling code, respectively, given a "format vector," types, that specifies the types of arguments.

```
typedef union { int i; double d; void *p; } type;
typedef enum { INTEGER, DOUBLE, POINTER } type_t; /* Types we expect to marshal */
extern void *alloc();
void cspec mk_marshal(type_t *types, int nargs) {
    int i;
    type *vspec m = local(type *); /* Spec of pointer to result vector */
    void cspec s = `{ m = (type *)alloc(nargs * sizeof(type)); };

    for (i = 0; i < nargs; i++) { /* Add code to marshal each param */
        switch(types[i]) {
        case INTEGER: s = `{ @s; m[$i].i = param($i, int); };      break;
        case DOUBLE:  s = `{ @s; m[$i].d = param($i, double); };  break;
        case POINTER: s = `{ @s; m[$i].p = param($i, void *); };  break;
        }
    }
    /* Return code spec to marshal parameters and return result vector */
    return `{ @s; return m; };
}
```

Figure 3-13: Sample marshaling code in `C.

```
typedef int (*fptr)(); /* Type of the function we will be calling */
void cspec mk_unmarshal(type_t *types, int nargs) {
    int i;
    fptr vspec fp = param(fptr, 0); /* Arg 0: the function to invoke */
    type *vspec m = param(type *, 1); /* Arg 1: the vector to unmarshal */
    void cspec args = push_init(); /* Initialize the dynamic argument list */

    for (i = 0; i < nargs; i++) { /* Build up the dynamic argument list */
        switch(types[i]) {
        case INTEGER: push(args, `m[$i].i); break;
        case DOUBLE:  push(args, `m[$i].d); break;
        case POINTER: push(args, `m[$i].p); break;
        }
    }
    /* Return code spec to call the given function with unmarshaled args */
    return `{ fp(args); };
}
```

Figure 3-14: Unmarshaling code in `C.

The sample code in Figure 3-13 generates a marshaling function for arguments with a particular set of types (in this example, int, void *, and double). First, it specifies code to allocate storage for a byte vector large enough to hold the arguments described by the type format vector. Then, for every type in the type vector, it creates a vspec that refers to the corresponding parameter, and constructs code to store the parameter's value into the byte vector at a distinct run-time constant offset. Finally, it specifies code that returns a pointer to the byte vector of marshaled arguments. After dynamic code generation, the function that has been constructed will store all of its parameters at fixed, non-overlapping offsets in the result vector. Since all type and offset computations have been done during environment binding, the generated code will be efficient. Further performance gains might be achieved if the code were to manage details such as endianness and alignment.

The code in Figure 3-14 generates an unmarshaling function that works with the marshaling code in Figure 3-13. It relies on `C's mechanism for constructing calls to arbitrary functions at run time. The generated code takes a function pointer as its first argument and a byte vector of marshaled arguments as its second argument. It unmarshals the values in the byte vector into their appropriate parameter positions, and then invokes the function pointer. mk_unmarshal works as follows. It creates the specifications for the generated function's two incoming arguments, and

```
typedef double cspec (*dptr)(double vspec);

/* Dynamically create a Newton-Raphson routine. n: max number of iterations;
   tol: maximum tolerance; p0: initial estimate; f: function to solve; fprime: derivative of f. */
double newton(int n, double tol, double usr_p0, dptr f, dptr fprime) {
     void cspec cs = `{
          int i; double p, p0 = usr_p0;
          for (i = 0; i < $n; i++) {
               p = p0 − f(p0) / fprime(p0); /* Compose cspecs returned by f and fprime */
               if (abs(p − p0) < tol) return p; /* Return result if we've converged enough */
               p0 = p; /* Seed the next iteration */
          }
          error("method failed after %d iterations“n", i);
     };
     return (*compile(cs,double))(); /* Compile, call, and return the result. */
}

/* Function that constructs a cspec to compute f(x) = (x+1)^2 */
double cspec f(double vspec x) { return `((x + 1.0) * (x + 1.0)); }
/* Function that constructs a cspec to calculate f'(x) = 2(x+1) */
double cspec fprime(double vspec x) { return `(2.0 * (x + 1.0)); }
/* Call newton to solve an equation */
void use_newton(void) { printf("Root is %f“n", newton(100, .000001, 10., f, fprime)); }
```

Figure 3-15: ‘C code that implements Newton's method for solving polynomials. This example computes the root of the function $f(x) = (x + 1)^2$.

initializes the argument list. Then, for every type in the type vector, it creates a cspec to index into the byte vector at a fixed offset and pushes this cspec into its correct parameter position. Finally, it creates the call to the function pointed to by the first dynamic argument.

3.1.3 Elimination of Call Overhead

‘C makes it easy to inline and compose functions dynamically. This feature is analogous to dynamic inlining through indirect function calls. It improves performance by eliminating function call overhead and by creating the opportunity for optimization across function boundaries.

Parameterized Library Functions

Dynamic function composition is useful when writing and using library functions that would normally be parameterized with function pointers, such as many mathematical and standard C library routines. The ‘C code for Newton's method [59] in Figure 3-15 illustrates its use. The function newton takes as arguments the maximum allowed number of iterations, a tolerance, an initial estimate, and two pointers to functions that return cspecs to evaluate a function and its derivative. In the calls f(p0) and fprime(p0), p0 is passed as a vspec argument. The cspecs returned by these functions are incorporated directly into the dynamically generated code. As a result, there is no function call overhead, and inter-cspec optimization can occur during dynamic code generation.

Network Protocol Layers

Another important application of dynamic inlining is the optimization of networking code. The modular composition of different protocol layers has long been a goal in the networking community [13]. Each protocol layer frequently involves data manipulation operations, such as checksumming and byte-swapping. Since performing multiple data manipulation passes is expensive, it is desirable to compose the layers so that all the data handling occurs in one phase [13].

```
unsigned cspec byteswap(unsigned vspec input) {
    return `((input << 24) | ((input & 0xff00) << 8) |
            ((input >> 8) & 0xff00) | ((input  >> 24) & 0xff));
}
/* "Byteswap" maintains no state and so needs no initial or final code */
void cspec byteswap_initial(void) { return `{}; }
void cspec byteswap_final(void) { return `{}; }
```

Figure 3-16: A sample pipe written in ‘C: byteswap returns a cspec for code that byte-swaps input.

‘C can be used to construct a network subsystem that dynamically integrates protocol data operations into a single pass over memory (e.g., by incorporating encryption and compression into a single copy operation). A simple design for such a system divides each data-manipulation stage into *pipes* that each consume a single input and produce a single output. These pipes can then be composed and incorporated into a data-copying loop. The design includes the ability to specify prologue and epilogue code that is executed before and after the data-copying loop, respectively. As a result, pipes can manipulate state and make end-to-end checks, such as ensuring that a checksum is valid after it has been computed.

The pipe in Figure 3-16 can be used to do byte-swapping. Since a byte swapper does not need to maintain any state, there is no need to specify initial and final code. The byte swapper simply consists of the "consumer" routine that manipulates the data.

To construct the integrated data-copying routine, the initial, consumer, and final cspecs of each pipe are composed with the corresponding cspecs of the pipe's neighbors. The composed initial code is placed at the beginning of the resulting routine; the consumer code is inserted in a loop and composed with code which provides it with input and stores its output; and the final code is placed at the end of the routine. A simplified version would look like the code fragment in Figure 3-17. In a mature implementation of this code, we could further improve performance by unrolling the data-copying loop. Additionally, pipes would take inputs and outputs of different sizes that the composition function would reconcile.

3.1.4 Elimination of Interpretation Overhead

Dynamic compilation can serve to strip away a layer of interpretation required to process bytecodes, database queries, complex data structures, and so forth. Furthermore, ‘C's imperative approach to dynamic code generation makes it well-suited for writing programs—such as compilers and compiling interpreters—that involve compilation. For such programs, an expressive dynamic compilation interface can be not only an optimization mechanism, but also a software design tool. Rather than writing a parser that generates bytecodes and then a bytecode interpreter, or a parser and then a full compiler back end, one need only write a parser that uses ‘C. As the parser processes its input, it can build up a dynamic code specification that represents the input. When the parsing is done, the compile special form does all the work that would normally be done by a back end. Thus, ‘C both generates efficient code and simplifies the overall development task.

Domain-Specific Languages

Small, domain-specific languages can benefit from dynamic compilation. The small query languages used to search databases are one class of such languages [47]. Since databases are large, dynamically compiled queries will usually be applied many times, which can easily pay for the cost of dynamic code generation.

We provide a toy example in Figure 3-18. The function mk_query takes a vector of queries. Each query contains the following elements: a database record field (i.e., CHILDREN or INCOME); a value to compare this field to; and the operation to use in the comparison (i.e., <, >, etc.). Given a query

```
typedef void cspec (*vptr)();
typedef unsigned cspec (*uptr)(unsigned cspec);
/* Pipe structure: contains pointers to functions that return cspecs
   for the initialization, pipe, and finalization code of each pipe */
struct pipe {
     vptr initial, final;          /* initial and final code */
     uptr pipe;                    /* pipe */
};

/* Return cspec that results from composing the given vector of pipes */
void cspec compose(struct pipe *plist, int n) {
     struct pipe *p;
     int vspec nwords = param(int, 0);                /* Arg 0: input size   */
     unsigned * vspec input = param(unsigned *, 1);   /* Arg 1: pipe input   */
     unsigned * vspec output = param(unsigned *, 2);  /* Arg 2: pipe output  */
     void cspec initial = `{}, cspec final = `{}; /* Prologue and epilogue code */
     unsigned cspec pipes = `input[i];/* Base pipe input */

     for (p = &plist[0]; p < &plist[n]; p++) { /* Compose all stages together */
          initial = `{ @initial; @p->initial(); }; /* Compose initial statements */
          pipes =`p->pipe(pipes); /* Compose pipes: one pipe's output is the next one's input */
          final = `{ @final; @p->final(); }; /* Compose final statements */
     }
     /* Create a function with initial statements first, consumer statements
        second, and final statements last. */
     return `{ int i;
               @initial;
               for (i = 0; i < nwords; i++) output[i] = pipes;
               @final;
     };
}
```

Figure 3-17: ‘C code for composing data pipes.

```
typedef enum { INCOME, CHILDREN /* ... */ } query; /* Query type */
typedef enum { LT, LE, GT, GE, NE, EQ } bool_op; /*  Comparison operation */

struct query {
    query record_field;        /* Field to use */
    unsigned val;              /* Value to compare to */
    bool_op bool_op;           /* Comparison operation */
};
struct record { int income; int children; /* ... */ }; /* Simple database record */

/* Function that takes a pointer to a database record and returns 0 or 1,
   depending on whether the record matches the query */
typedef int (*iptr)(struct record *r);

iptr mk_query(struct query *q, int n) {
    int i, cspec field, cspec expr = `1; /* Initialize the boolean expression */
    struct record * vspec r = param(struct record *, 0); /* Record to examine */

    for (i = 0; i < n; i++) { /* Build the rest of the boolean expression */
        switch (q[i].record_field) { /* Load the appropriate field value */
        case INCOME:   field = `(r->income); break;
        case CHILDREN: field = `(r->children); break;
        /* ... */
        }
        switch (q[i].bool_op) { /* Compare the field value to runtime constant q[i] */
        case LT: expr = `(expr && field < $q[i].val); break;
        case EQ: expr = `(expr && field == $q[i].val); break;
        case LE: expr = `(expr && field <= $q[i].val); break;
        /* ... */
        }
    }
    return (iptr)compile(`{ return expr; }, int);
}
```

Figure 3-18: Compilation of a small query language with ‘C.

```
enum { WHILE, IF, ELSE, ID, CONST, LE, GE, NE, EQ };  /* Multi-character tokens */
int expect();             /* Consume the given token from the input stream, or fail if not found */
int cspec expr();         /* Parse unary expressions */
int gettok();             /* Consume a token from the input stream */
int look();               /* Peek at the next token without consuming it */
int cur_tok;              /* Current token */
int vspec lookup_sym(); /* Given a token, return corresponding vspec */

void cspec stmt() {
    int cspec e = `0;  void cspec s = `{}, s1 = `{}, s2 = `{};
    switch (gettok()) {
    case WHILE: /* 'while' '(' expr ')' stmt */
        expect('('); e = expr();
        expect(')'); s = stmt();
        return `{ while(e) @s; };
    case IF: /* 'if' '(' expr ')' stmt { 'else' stmt } */
        expect('('); e = expr();
        expect(')'); s1 = stmt();
        if (look(ELSE)) {
            gettok(); s2 = stmt();
            return `{ if (e) @s1; else @s2; };
        } else return `{ if (e) @s1; };
    case '{': /* '{' stmt* '}' */
        while (!look('}')) s = `{ @s; @stmt(); };
        return s;
    case ';': return `{};
    case ID: { /* ID '=' expr ';' */
        int vspec lvalue = lookup_sym(cur_tok);
        expect('='); e = expr(); expect(';');
        return `{ lvalue = e; };
    }
    default: parse_err("expecting statement");
    }
}
```

Figure 3-19: A sample statement parser from a compiling interpreter written in ‘C.

vector, mk_query dynamically creates a query function which takes a database record as an argument and checks whether that record satisfies all of the constraints in the query vector. This check is implemented simply as an expression which computes the conjunction of the given constraints. The query function never references the query vector, since all the values and comparison operations in the vector have been hardcoded into the dynamic code's instruction stream.

The dynamically generated code expects one incoming argument, the database record to be compared. It then "seeds" the boolean expression: since we are building a conjunction, the initial value is 1. The loop then traverses the query vector and builds up the dynamic code for the conjunction according to the fields, values, and comparison operations described in the vector. When the cspec for the boolean expression is constructed, mk_query compiles it and returns a function pointer. That optimized function can be applied to database entries to determine whether they match the given constraints.

Compiling Interpreters

Compiling interpreters (also known as JIT compilers) are important pieces of technology: they combine the flexibility of an interpreted programming environment with the performance of compiled code. For a given piece of code, the user of a compiling interpreter pays a one-time cost of compilation, which can be roughly comparable to that of interpretation. Every subsequent use of that code employs the compiled version, which can be much faster than the interpreted version. Compiling interpreters are currently very popular in shippable code systems such as Java.

Figure 3-19 contains a fragment of code for a simple compiling interpreter written in ‘C. This interpreter translates a simple subset of C. As it parses the input program, it builds up a cspec that represents it. At the end of the parse, the cspec is passed to compile, which automatically generates executable code.

3.2 Limits of Dynamic Compilation

After this discussion, it is instructive to take a negative approach. In what sorts of problems is dynamic code generation not applicable? Given a static routine or an algorithm, can we immediately determine whether dynamic compilation is not the right solution to provide performance improvements? To some degree, yes. A few important obstacles account for most of the cases in which dynamic code generation is inapplicable:

Frequent data changes. This situation can also be viewed as low data reuse. For instance, the ‘C matrix factorization codes in Section A.1.1 may not provide performance advantages relative to static code if the matrices involved change frequently—most dynamic code must be reused at least a few times before the cumulative time savings obtained thanks to its high level of optimization outweigh the overhead of generating it.

Data-dependent control flow. If control flow is not data-dependent, then it may still be possible to perform some dynamic optimizations (for instance, loop unrolling), even though the data that is being manipulated changes from run to run. In a sense, therefore, this factor is complementary to frequent data changes. For example, the mergesort routine in Section A.2.1 can be specialized to a particular input length because mergesort is not sensitive to its input. Quicksort, on the other hand, could not be optimized in this way because its choice of pivots depends on the values in the array that is being sorted.

Analogously, as long as the data changes rarely enough, it may not matter that control flow is data-dependent. For instance, the specialized Huffman compression code in Section A.2.5 has data-dependent control flow, but the data—the values in the Huffman code—do not change frequently and can be reused for many compression runs. By contrast, the data processed by quicksort changes every time (presumably, a given array only needs to be sorted once), which makes the algorithm unsuitable for dynamic optimization.

Big loops. Data-dependent loop unrolling is an effective optimization enabled by dynamic code generation. However, big loops have negligible loop overhead, may cause cache misses when unrolled, and may require considerable overhead to generate dynamically. It therefore helps to focus on small, tight loops when converting existing static code to use dynamic compilation.

Floating point constants. Unfortunately, most processors cannot encode floating point instruction operands as immediates, making it impossible to benefit from run-time constant floating point values. This fact is one of the greatest limitations to the applicability of dynamic optimizations in numerical code.

The first three problems are inherent to dynamic compilation—one must always take into account the tradeoff between dynamic code performance improvement and generation overhead. The fourth problem is caused by current instruction set architectures—if it were possible to encode floating point immediates in an instruction word, dynamic compilation would be dramatically more effective for numerical code. As we have seen, however, despite these limitations, ‘C and dynamic compilation in general can be used effectively in a variety of practical situations.

Chapter 4

Implementation and Performance

The 'C compiler, tcc, tries to achieve two goals: high-quality dynamic code and low dynamic compilation overhead. The challenge is to reconcile these two goals with 'C's imperative code generation model, which makes it difficult to optimize dynamic code at static compilation time. This chapter describes how tcc handles this tradeoff and evaluates tcc's performance.

4.1 Architecture

The tcc compiler is based on lcc [31, 30], a portable compiler for ANSI C. lcc performs common subexpression elimination within extended basic blocks, and uses lburg [32] to find the lowest-cost implementation of a given IR-level construct. Otherwise, it performs few optimizations.

Figure 4-1 illustrates the interaction of static and dynamic compilation in tcc. All parsing and semantic checking of dynamic expressions occurs at static compile time. Semantic checks are performed at the level of dynamically generated expressions. For each cspec, tcc performs internal type checking. It also tracks goto statements and labels to ensure that a goto does not transfer control outside the body of the containing cspec.

Unlike traditional static compilers, tcc uses two types of back ends to generate code. One is the static back end, which compiles the non-dynamic parts of 'C programs, and emits either native assembly code or C code suitable for compilation by an optimizing compiler. The other, referred to as the dynamic back end, emits C code to *generate* dynamic code. Once produced by the dynamic back end, this C code is in turn compiled by the static back end.

As mentioned in Section 1.3, tcc provides two dynamic code generation run-time systems so as to trade code generation speed for dynamic code quality. Both systems provide a RISC-like interface that can be used to specify instructions to be generated. The more lightweight run-time system is VCODE [22], which emits code quickly in one pass and requires as few as 10 instructions per generated instruction in the best case. Although it is fast, it only has access to local information about backquote expressions. However, generating efficient code from composed cspecs requires, in general, optimization analogous to function inlining and interprocedural optimization. As a result, the quality of code generated by VCODE could often be improved. The second run-time system, ICODE, produces better code at the expense of additional dynamic compilation overhead. Rather than emitting code in one pass, it builds and optimizes an intermediate representation prior to code generation.

lcc is not an optimizing compiler. The assembly code emitted by its traditional static back ends is usually significantly slower (even three or more times slower) than that emitted by optimizing compilers such as gcc or vendor C compilers. To improve the quality of static code emitted by tcc, we implemented a static back end that generates ANSI C from 'C source; this code can then be compiled by any optimizing compiler. lcc's traditional back ends can thus be used when static compilation must be fast (i.e., during development), and the C back end can be used when the performance of the code is critical.

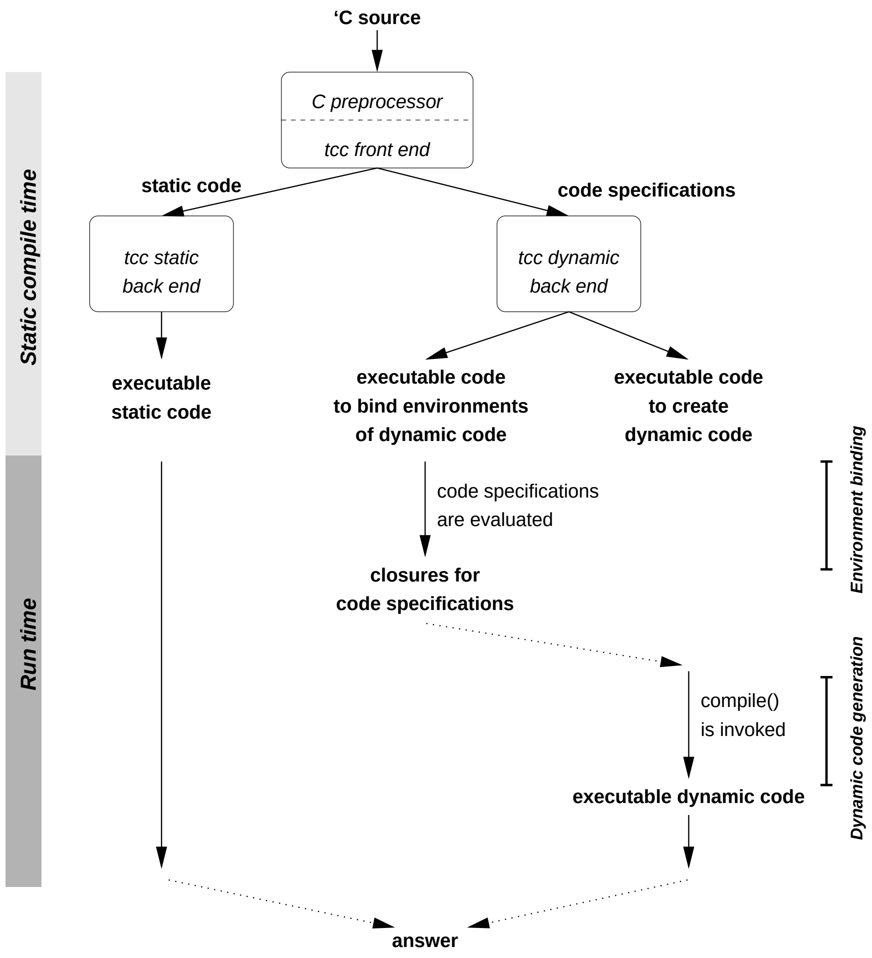

Figure 4-1: Overview of the tcc compilation process.

4.2 The Dynamic Compilation Process

The creation of dynamic code can be divided into three phases: static compilation, environment binding, and dynamic code generation. This section describes how tcc implements these three phases.

4.2.1 Static Compile Time

During static compilation, tcc compiles the static parts of a 'C program just like a traditional C compiler. It compiles each dynamic part—each backquote expression—to a *code-generating function* (CGF), which is invoked at run time to generate code for dynamic expressions.

In order to minimize the overhead of dynamic compilation, tcc performs as much work as possible statically. The intermediate representation of each backquote expression is statically processed by the common subexpression elimination and other local optimizations performed by the lcc front end. Instruction selection that depends on operand types is performed statically. tcc also uses copt [29] to perform static peephole optimizations on the code-generating macros used by CGFs.

However, not all instruction selection and register allocation can occur statically. For instance, it is not possible to determine statically what vspecs or cspecs will be incorporated into other cspecs when the program is executed. Hence, allocation of dynamic lvalues (vspecs) and of results of composed cspecs must be performed dynamically. Dynamic register allocation is discussed further in Section 4.3 and Chapter 5.

```
cspec_t i = ((closure0 = (closure0_t)alloc_closure(4)),
                 (closure0->cgf = cgf0), /* code gen func */
                 (cspec_t)closure0);

cspec_t c = ((closure1 = (closure1_t)alloc_closure(16)),
                 (closure1->cgf = cgf1), /* code gen func */
                 (closure1->cs_i = i), /* nested cspec */
                 (closure1->rtc_j = j), /* run-time const */
                 (closure1->fv_k = &k),/* free variable */
                 (cspec_t)closure1);
```

Figure 4-2: Sample closure assignments.

```
unsigned int cgf0 (closure0_t c) {
  vspec_t itmp0 = local (INT); /* int temporary */
  seti (itmp0, 5); /* set it to 5 */
  return itmp0; /* return the location */
}

void cgf1 (closure1_t *c) {
  vspec_t itmp0 = local (INT); /* some temporaries */
  vspec_t itmp1 = local (INT);
  ldii (itmp1,zero,c->fv_k); /* load from addr of free variable k */
  mulii (itmp1,itmp1,c->rtc_j); /* multiply by run-time const j */
  /* now apply i's CGF to i's closure: cspec composition! */
  itmp0 = (*c->cs_i->cgf)(c->cs_i);
  addi (itmp1,itmp0,itmp1); /* add the result to the total */
  reti (itmp1); /* emit a return (not return a value) */
}
```

Figure 4-3: Sample code-generating functions.

State for dynamic code generation is maintained in CGFs and in dynamically allocated closures. Closures are data structures that store five kinds of necessary information about the run-time environment of a backquote expression: (1) a function pointer to the corresponding statically generated CGF; (2) information about inter-cspec control flow (i.e., whether the backquote expression is the destination of a jump); (3) the values of run-time constants bound via the $ operator; (4) the addresses of free variables; (5) pointers to the run-time representations of the cspecs and vspecs used inside the backquote expression. Closures are necessary to reason about composition and out-of-order specification of dynamic code.

For each backquote expression, tcc statically generates both its code-generating function and the code to allocate and initialize closures. A new closure is initialized each time a backquote expression is evaluated. Cspecs are represented by pointers to closures.

For example, consider the following code:

```
int j, k;
int cspec i = `5;
void cspec c = `{ return i+$j*k; };
```

tcc implements the assignments to these cspecs by assignments to pointers to closures, as illustrated in Figure 4-2. i's closure contains only a pointer to its code-generating function. c has more dependencies on its environment, so its closure also stores other information.

Simplified code-generating functions for these cspecs appear in Figure 4-3. cgf0 allocates a temporary storage location, generates code to store the value 5 into it, and returns the location. cgf1 must do a little more work: the code that it generates loads the value stored at the address of free variable k into a register, multiplies it by the value of the run-time constant j, adds this to the dynamic value of i, and returns the result. Since i is a cspec, the code that computes the dynamic value of i is generated by calling i's code-generating function.

4.2.2 Run Time

At run time, the code that initializes closures and the code-generating functions run to create dynamic code. As illustrated in Figure 4-1, this process consists of two parts: environment binding and dynamic code generation.

Environment Binding

During environment binding, code such as that in Figure 4-2 builds a closure that captures the environment of the corresponding backquote expression. Closures are heap-allocated, but their allocation cost is greatly reduced (down to a pointer increment, in the normal case) by using arenas [28].

Dynamic Code Generation

During dynamic code generation, the 'C run-time system processes the code-generating functions. The CGFs use the information in the closures to generate code, and they perform various dynamic optimizations.

Dynamic code generation begins when the compile special form is invoked on a cspec. Compile calls the code-generating function for the cspec on the cspec's closure, and the CGF performs most of the actual code generation. In terms of our running example, the code int (*f)() = compile(j, int); causes the run-time system to invoke closure1→cgf(closure1).

When the CGF returns, compile links the resulting code, resets the information regarding dynamically generated locals and parameters, and returns a pointer to the generated code. We attempt to minimize poor cache behavior by laying out the code in memory at a random offset modulo the i-cache size. It would be possible to track the placement of different dynamic functions to improve cache performance, but we do not do so currently.

Cspec composition—the inlining of code corresponding to one cspec, b, into that corresponding to another cspec, a, as described in Section 2.1.3—occurs during dynamic code generation. This composition is implemented simply by invoking b's CGF from within a's CGF. If b returns a value, the value's location is returned by its CGF, and can then be used by operations within a's CGF.

The special forms for inter-cspec control flow, jump and label, are implemented efficiently. Each closure, including that of the empty void cspec '{}, contains a field that marks whether the corresponding cspec is the destination of a jump. The code-generating function checks this field, and if necessary, invokes a VCODE or ICODE macro to generate a label, which is eventually resolved when the run-time system links the code. As a result, label can simply return an empty cspec. jump marks the closure of the destination cspec appropriately, and then returns a closure that contains a pointer to the destination cspec and to a CGF that contains an ICODE or VCODE unconditional branch macro.

Within a CGF, tcc also generates code that implements some inexpensive dynamic optimizations. These optimizations do not depend on elaborate run-time data structures or intermediate forms; rather, they are performed directly by the code that is generated at static compile time. They significantly improve code quality without greatly increasing dynamic compilation overhead. In fact, in some cases they even decrease compilation overhead.

tcc implements several simple optimizations of this type:

Loop unrolling. Some versions of tcc emit code-generating code that automatically unrolls loops based on run-time constants. If a loop is bounded by constants or run-time constants, then control flow can be determined at dynamic code generation time.

For example, consider the simple code:

```
`{ for (i = $l; i < $h; i++} { ... /* loop body */ ... } };
```

Since the loop is bounded by run-time constants (l and h), looping can occur at dynamic compile time to produce loop-free dynamic code. The code-generating function looks like this:

```
for (i = rtc_l; i < rtc_h; i++) {
  /* code to generate loop body, with i treated as a constant */
}
```

The only dynamic compile-time overhead of this loop unrolling—other than the inevitable need to emit multiple loop bodies—is the loop bounds check and increment of i.

Run-time constant information can also propagate down loop nesting levels: for example, if a loop induction variable is bounded by run-time constants, and it is in turn used to bound a nested loop, then the induction variable of the nested loop is considered run-time constant too, within each unrolled iteration of the nested loop.

Dead code elimination. Dead code elimination works on the same principle as loop unrolling. The dynamic code

```
`{ if ($condition) { ... /* do X */ } };
```

is generated by the following code:

```
if (rtc_condition) { ... /* generate code to do X */ }
```

Since the if statement tests a run-time constant, it can be performed at run-time. If the condition is true, the generated code contains no test. If it is false, the code is not generated, which improves the performance of both the dynamic code and the dynamic compilation process.

Although tcc implements only the basic functionality, this scheme can be extended to more sophisticated tests. For instance,

```
`{ if ($x && y || $w && z) { ... }
```

could be compiled to

```
if (rtc_x) { /* emit "if (y) goto X;" */ }
if (rtc_w) { /* emit "if (z) goto X;" */ }
if (rtc_x || rtc_w) {
  /* emit "goto Y;" */
  /* emit "X: body of if statement" */
  /* emit "Y:" */
}
```

Constant folding. The code-generating functions evaluate at dynamic compile time any parts of an expression that consist of static and run-time constants. The dynamically emitted instructions then encode these values as immediates.

It would not be difficult to rearrange commutative expressions to improve constant folding even further. For instance, tcc folds the expression a+$x+3+$y into just one addition, but $x+a+3+$y is folded to two additions, because the run-time constants are in non-adjacent branches of the expression tree.

Strength reduction of multiplication. The code-generating functions automatically replace multiplication by a run-time constant integer with a minimal series of shifts and adds, as described in Briggs and Harvey [10]. Multiplications by zero are replaced by a clear instruction, and multiplications by one by a move instruction.

The structure of lcc, on which tcc is based, makes it awkward to implement data flow analyses. As a result, all of these optimizations in tcc are local and rather ad-hoc. A better solution would be to implement a data flow analysis pass to propagate information about run-time constants in dynamic code. It would enable tcc to identify derived run-time constants and to stage aggressive constant and copy propagation based on run-time constants. At present, some of these optimizations can be performed by the 'C programmer by cleverly composing cspecs and run-time constants, but more aggressive staged data flow analysis would improve dynamic code and probably also speed up dynamic compilation. DyC, for example, benefits substantially from staged data flow analysis and optimization [39]. Even without compile-time data flow analysis, however, the staged optimizations improve dynamic code and incur negligible dynamic compile-time overhead.

4.3 Run-Time Systems

As described earlier, tcc provides two run-time systems for generating code. VCODE emits code locally, with no global analysis or optimization. ICODE builds up an intermediate representation that allows it to optimize the code after the order of composition of cspecs has been completely determined.

These two run-time systems allow programmers to choose the appropriate level of run-time optimization. The choice is application-specific: it depends on the number of times the code will be used and on the code's size and structure. Programmers can select which run-time system to use when they compile a 'C program.

4.3.1 VCODE

To reduce code generation overhead, the user can make tcc generate CGFs that contain VCODE macros. VCODE provides an idealized load/store RISC interface. Each instruction in this interface is a C macro which directly emits the corresponding instruction (or series of instructions) for the target architecture.

VCODE generates code and allocates registers all in one pass. It uses a simple and fast one-pass register allocation algorithm, which is described in detail Section 5.1. To further reduce register allocation overhead, tcc reserves a limited number of physical registers. These registers are not allocated dynamically, but instead are managed at static compile time by tcc's dynamic back end. They can only be used for values whose live ranges do not span composition with a cspec, and are typically employed for expression temporaries. As a result of these optimizations, VCODE register allocation is quite fast. However, it does not always produce the best code, especially when many cspecs composed together create high register pressure.

VCODE can be optionally followed by a peephole optimization pass. The peephole optimizer, bpo, is described in Chapter 7. It is designed to efficiently perform local optimizations on binary code. For instance, it effectively removes redundant register moves introduced when vspecs are composed in cspecs. bpo is conceptually the last stage of dynamic compilation in tcc, but it runs in an incremental manner. It rewrites and optimizes basic blocks one by one as they are emitted by VCODE. This strategy limits the necessary temporary storage to the size of the biggest basic block, and allows linking—resolution of jumps and jump targets—to be done more easily and quickly than after a complete peephole optimization pass over the entire dynamic code.

4.3.2 ICODE

To improve code quality, the user can make tcc generate CGFs that contain ICODE macros. ICODE generates an intermediate representation on which optimizations can be performed. For example, it can perform global register allocation on dynamic code more effectively than VCODE in the presence of cspec composition.

ICODE provides an interface similar to that of VCODE, with two main extensions: an infinite number of registers and primitives to express changes in estimated usage frequency of code. The first extension allows ICODE clients to emit code that assumes no spills, leaving the work of global, inter-cspec register allocation to ICODE. The second allows ICODE to obtain estimates of code execution frequency at low cost. For instance, prior to invoking ICODE macros that correspond to a loop body, the ICODE client could invoke refmul(10); this tells ICODE that all variable references occurring in the subsequent macros should be weighted as occurring 10 times (an estimated average number of loop iterations) more than the surrounding code. After emitting the loop body, the ICODE client should invoke a corresponding refdiv(10) macro to correctly weight code outside the loop. The estimates obtained in this way are useful for several optimizations; they currently provide approximate variable usage counts that help to guide register allocation.

ICODE's intermediate representation is designed to be compact (two four-byte machine words per ICODE instruction) and easy to parse in order to reduce the overhead of subsequent passes. When compile is invoked in ICODE mode, ICODE builds a flow graph, performs register allocation, and finally generates executable code.

ICODE builds a control-flow graph in one pass after all CGFs have been invoked. The flow graph is a single array that uses pointers for indexing. In order to allocate all required memory at the same time, ICODE computes an upper bound on the number of basic blocks by summing the numbers of labels and jumps emitted by ICODE macros. After allocating space for an array of this size, it traverses the buffer of ICODE instructions and adds basic blocks to the array in the same order in which they exist in the list of instructions. Forward references are initially stored in an array of pairs of basic block addresses; when all the basic blocks are built, the forward references are resolved by traversing this array and linking the pairs of blocks listed in it. As it builds the flow graph, ICODE also collects a minimal amount of local data-flow information (def and use sets for each basic block). All memory management occurs through arenas [28], which ensures low amortized cost for memory allocation and essentially free deallocation.

Good register allocation is the main benefit that ICODE provides over VCODE. ICODE currently implements three different register allocation algorithms: graph coloring, linear scan, and a simple scheme based on estimated usage counts. Linear scan, in turn, can obtain live intervals from full live variable information or via a faster, approximate method that coarsens the flow graph to the level of strongly connected components. All three register allocation algorithms are described in Section 5.2. They allow ICODE to provide a variety of tradeoffs of compile-time overhead versus quality of code. Graph coloring is most expensive and usually produces the best code, linear scan is considerably faster but sometimes produces worse code, and the usage count allocator is faster than linear scan but can produce considerably worse code. However, given the relatively small size of most 'C dynamic code, the algorithms perform similarly on the benchmarks presented in this chapter. As a result, Section 4.5 presents measurements only for a representative case, the linear scan allocator using live intervals derived from full live variable information.

After register allocation, ICODE translates the intermediate representation into executable code. The code emitter makes one pass through the ICODE intermediate representation; it invokes the VCODE macro that corresponds to each ICODE instruction, and prepends and appends spill code as necessary. Peephole optimizations, if any, are performed at this time.

ICODE has several hundred instructions (the Cartesian product of operation kinds and operand types), so a translator for the entire instruction set is quite large. Most 'C programs, however, use only a small subset of all ICODE instructions. tcc therefore keeps track of the ICODE instructions used by an application. It encodes this usage information for a given 'C source file in dummy symbol names in the corresponding object file. A pre-linking pass then scans all the files about to be linked and emits an additional object file containing an ICODE-to-binary translator tailored specifically to the ICODE macros present in the executable. This simple trick significantly reduces the size of the ICODE code generator; for example, for the benchmarks presented in this chapter it usually shrank the code generators by a factor of five or six.

ICODE is designed to be a generic framework for dynamic code optimization; it is possible to extend it with additional optimization passes, such as copy propagation, common subexpression elimination, and so forth. However, much dynamic optimization beyond good register allocation is

Benchmark	Description	Section	Page
ms	Scale a 100x100 matrix by the integers in $[10, 100]$	3.1.1	46
hash	Hash table, constant table size, scatter value, and hash table pointer: one hit and one miss	3.1.1	45
dp	Dot product with a run-time constant vector: length 40, one-third zeroes	3.1.1	47
binary	Binary search on a 16-element constant array; one hit and one miss	3.1.1	47
pow	Exponentiation of 2 by the integers in $[10, 40]$	3.1.1	46
dfa	Finite state machine computation 6 states, 13 transitions, input length 16	3.1.1	49
heap	Heapsort, parameterized with a specialized swap: 500-entry array of 12-byte structures	3.1.2	50
mshl	Marshal five arguments into a byte vector	3.1.2	52
unmshl	Unmarshal a byte vector, and call a function of five arguments	3.1.2	52
ntn	Root of $f(x) = (x+1)^2$ to a tolerance of 10^{-9}	3.1.3	53
ilp	Integrated copy, checksum, byteswap of a 16KB buffer	3.1.3	55
query	Query 2000 records with seven binary comparisons	3.1.4	56

Table 4.1: Descriptions of benchmarks.

probably not practical: the increase in dynamic compile time is usually not justified by sufficient improvements in the speed of the resulting code.

4.4 Evaluation Methodology

To evaluate the performance of tcc and of specific dynamic compilation algorithms, we use some of the routines presented in Chapter 3. Table 4.1 briefly summarizes each of these benchmarks, and lists the section in which it is described.

4.4.1 Benchmark-Specific Details

Since the performance improvements of dynamic code generation hinge on customizing code to data, the performance of all of these benchmarks is data-dependent to some degree. In particular, the amount of code generated by the benchmarks is in some cases dependent on the input data. For example, since dp generates code to compute the dot product of an input vector with a run-time constant vector, the size of the dynamic code (and hence its i-cache performance) is dependent on the size of the run-time constant vector. Its performance relative to static code also depends on the density of 0s in the run-time constant vector, since those elements are optimized out when generating the dynamic code. Similarly, binary and dfa generate more code for larger inputs, which generally improves their performance relative to equivalent static code until negative i-cache effects come into play.

Some other benchmarks—ntn, ilp, and query—involve dynamic function inlining that is affected by input data. For example, the code inlined in ntn depends on the function to be computed, that in ilp on the nature of the protocol stack, and that in query on the type of query submitted by the user. The advantage of dynamic code over static code increases with the opportunity for inlining and cross-function optimization. For example, an ilp protocol stack composed from many small passes will perform relatively better in dynamic code that one composed from a few larger passes.

Lastly, a few benchmarks are relatively data-independent. pow, heap, mshl, and unmshl generate varying amounts of code, but the differences are small for most reasonable inputs. ms obtains performance improvements by hardwiring loop bounds and strength-reducing multiplication by the scale factor. hash makes similar optimizations when computing a hash function. The values of run-time constants may affect performance to some degree (e.g., excessively large constants are not useful for this sort of optimization), but such effects are much smaller than those of more large-scale dynamic optimizations.

4.4.2 Measurement Infrastructure

The machine used for all measurements of ‘C benchmarks is a Sun Ultra 2 Model 2170 workstation with 384MB of main memory and a 168MHz UltraSPARC-I CPU. The UltraSPARC-I can issue up to two integer and two floating-point instructions per cycle, and has a write-through, nonallocating, direct-mapped, on-chip 16KB cache. It implements the SPARC version 9 architecture [67]. tcc also generates code for the MIPS family of processors. For clarity, and since results on the two architectures are similar, this thesis reports only SPARC measurements.

Run times were obtained by summing the system and user times reported by the UNIX getrusage system call. The time for each benchmark trial was computed by timing a large number of iterations (between 100 and 100,000, depending on the benchamrk, so as to provide several seconds of granularity), and dividing the result by the number of iterations so as to obtain the average overhead of a single iteration. This form of measurement ignores the effects of cache refill misses, but is representative of how these applications would likely be used (e.g., in tight inner loops). Reported values for each benchmark were obtained by taking the mean of ten such trials. The standard deviation for each set of trials was negligible.

Certain aspects of linear scan register allocation (Section 5.2) and fast peephole optimizations (Chapter 7) were evaluated on other platforms, specifically a Digital Alpha-based workstation and a Pentium-based PC. Details about these platforms and measurements appear in the relevant sections.

4.4.3 Comparison of Static and Dynamic Code

To compare the performance of dynamic code with that of static code, each benchmark was written both in ‘C and in equivalent static C. The speedup due to dynamic code generation is computed by dividing the time required to run the static code by the time required to run the corresponding dynamic code. Overhead is measured in terms of each benchmark's cross-over point, if one exists. As explained in Section 1.3, this point is the number of times that dynamic code must be used so that the time gained by running the dynamic code equals or exceeds the overhead of dynamic code generation.

The ‘C programs were compiled with both the VCODE and the ICODE-based tcc back ends. Performance numbers for the ICODE run-time system were obtained using linear scan register allocation with live intervals derived from live variable information. Comparative data for the other register allocation algorithms appears in Chapter 5. Neither VCODE nor ICODE were followed by peephole optimization. Therefore, the measurements reported for VCODE correspond to the fastest code generation and poorest dynamic code obtainable with tcc on these benchmarks.

The static C programs were compiled both with the lcc compiler and with the GNU C compiler, gcc. The code-generating functions used for dynamic code generation are created from the lcc intermediate representation, using that compiler's code generation strategies. As a result, the performance of lcc-generated code should be used as the baseline to measure the impact of dynamic code generation. Measurements collected using gcc serve to compare tcc to an optimizing compiler of reasonable quality.

4.5 Performance Evaluation

‘C and tcc support efficient dynamic code generation. In particular, the measurements in this section demonstrate the following results:

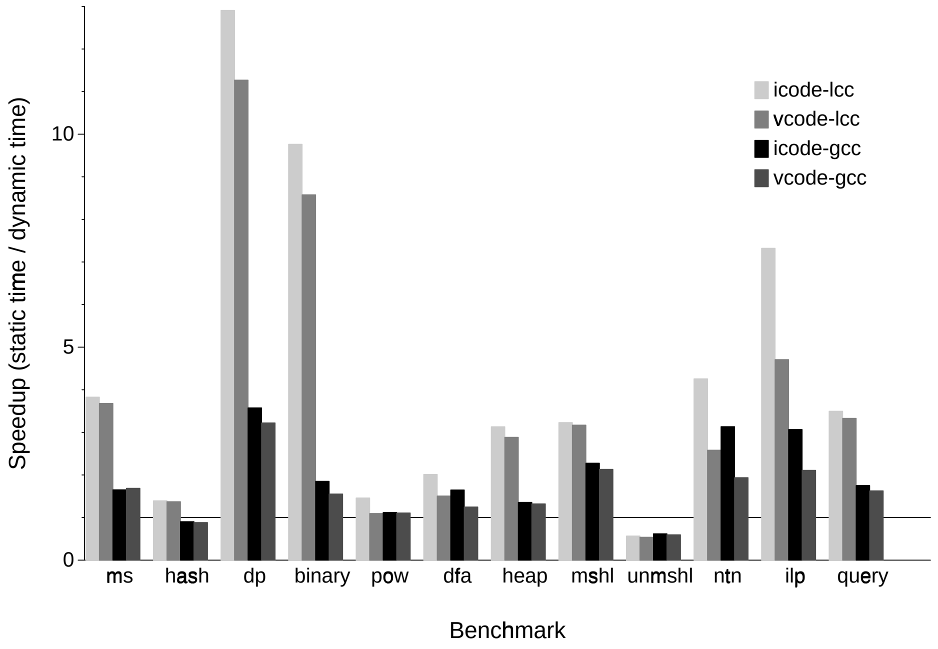

Figure 4-4: Speedup of dynamic code over static code.

- By using ‘C and tcc, we can achieve good speedups relative to static C. Speedups by a factor of two to four are common for the programs that we have described.

- ‘C and tcc do not impose excessive overhead on performance. The cost of dynamic compilation is usually recovered in under 100 runs of a benchmark; sometimes this cost can be recovered in one run of a benchmark.

- The tradeoff between dynamic code quality and dynamic code generation speed must be made on a per-application basis. For some applications, it is better to generate code faster; for others, it is better to generate better code.

- Dynamic code generation can result in large speedups when it enables large-scale optimization: when interpretation can be eliminated, or when dynamic inlining enables further optimization. It provides smaller speedups if only local optimizations, such as strength reduction, are performed dynamically. In such cases, the cost of dynamic code generation may outweigh its benefits.

4.5.1 Performance

This section presents results for the benchmarks in Table 4.1 and for xv, a freely available image manipulation package. The results show that dynamic compilation can frequently improve code performance by a factor of two or more, and that code generation overhead can often be amortized with fewer than 100 runs of the dynamic code. VCODE generates code approximately three to eight times more quickly than ICODE. Nevertheless, the code generated by ICODE can be considerably faster than that generated by VCODE. A programmer can choose between the two systems to trade code quality for code generation speed, depending on the needs of the application.

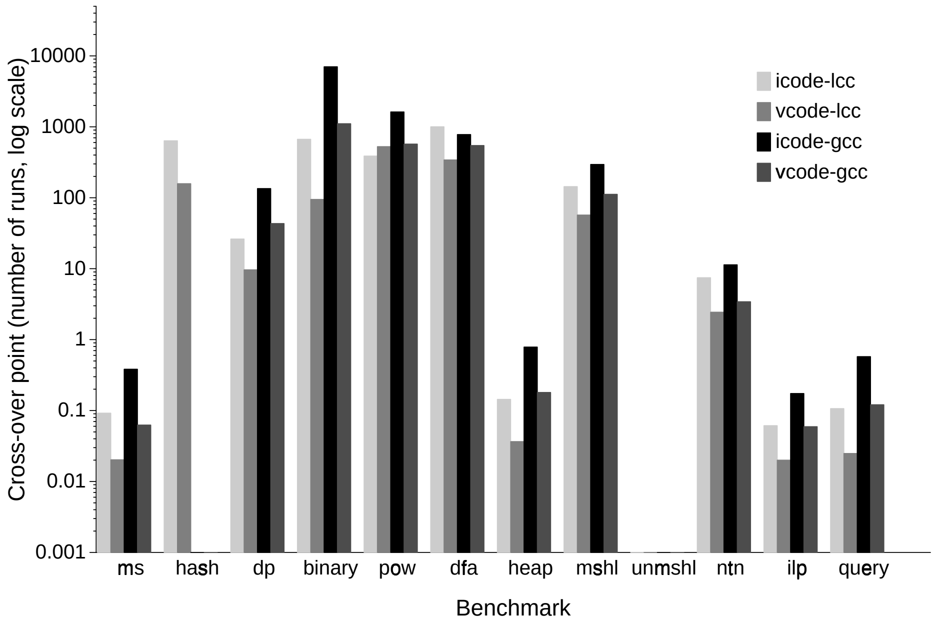

Figure 4-5: Cross-over points, in number of runs. If the cross-over point does not exist, the bar is omitted.

Speedup

Figure 4-4 shows that using 'C and tcc improves the performance of almost all of our benchmarks. Both in this figure and in Figure 4-5, the legend indicates which static and dynamic compilers are being compared. icode-lcc compares dynamic code created with ICODE to static code compiled with lcc; vcode-lcc compares dynamic code created with VCODE to static code compiled with lcc. Similarly, icode-gcc compares ICODE to static code compiled with gcc, and vcode-gcc compares VCODE to static code compiled with gcc.

In general, dynamic code is significantly faster than static code: speedups by a factor of two relative to the best code emitted by gcc are common. Unsurprisingly, the code produced by ICODE is faster than that produced by VCODE, by up to 50% in some cases. Also, the GNU compiler generates better code than lcc, so the speedups relative to gcc are almost always smaller than those relative to lcc. As mentioned earlier, however, the basis for comparison should be lcc, since the code-generating functions are generated by an lcc-style back end, which does not perform static optimizations.

The benchmarks that achieve the highest speedups are those in which dynamic information allows the most effective restructuring of code relative to the static version. The main classes of such benchmarks are numerical code in which particular values allow large amounts of work to be optimized away (e.g., dp), code in which an expensive layer of data structure interpretation can be removed at run time (e.g., query), and code in which inlining can be performed dynamically but not statically (e.g., ilp). Dynamic code generation does not pay off in only one benchmark, unmshl. In this benchmark, 'C provides functionality that does not exist in C. The static code used for comparison implements a special case of the general functionality provided by the 'C code, and it is very well tuned.

Cross-over

Figure 4-5 indicates that the cost of dynamic code generation in tcc is reasonably low. Remember that the cross-over point on the vertical axis is the number of times that the dynamic code must

Convolution mask (pixels)	Times (seconds)			
	lcc	gcc	tcc (ICODE)	DCG overhead
3×3	5.79	2.44	1.91	2.5×10^{-3}
7×7	17.57	6.86	5.78	3.5×10^{-3}

Table 4.2: Performance of convolution on an 1152 x 900 image in xv.

be used in order for the total overhead of its compilation and uses to be equal to the overhead of the same number of uses of static code. This number is a measure of how quickly dynamic code "pays for itself." For all benchmarks except query, one use of dynamic code corresponds to one run of the dynamically created function. In query, however, the dynamic code is used as a small part of the overall algorithm: it is the test function used to determine whether a record in the database matches a particular query. As a result, in that case one use of dynamic code is defined to be one run of the search algorithm, which corresponds to many invocations (one per database entry) of the dynamic code. This methodology realistically measures how specialization is used in these cases.

In the case of unmshl, the dynamic code is slower than the static one, so the cross-over point never occurs. Usually, however, the performance benefit of dynamic code generation occurs after a few hundred or fewer runs. In some cases (ms, heap, ilp, and query), the dynamic code pays for itself after only one run of the benchmark. In ms and heap, this occurs because a reasonable problem size is large relative to the overhead of dynamic compilation, so even small improvements in run time (from strength reduction, loop unrolling, and hardwiring pointers) outweigh the code generation overhead. In addition, ilp and query exemplify the types of applications in which dynamic code generation can be most useful: ilp benefits from extensive dynamic function inlining that cannot be performed statically, and query dynamically removes a layer of interpretation inherent in a database query language.

Figures 4-4 and 4-5 show how dynamic compilation speed can be exchanged for dynamic code quality. VCODE can be used to perform fast, one-pass dynamic code generation when the dynamic code will not used very much. However, the code generated by ICODE is often considerably faster than that generated by VCODE; hence, ICODE is useful when the dynamic code is run more times, so that the code's performance is more important than the cost of generating it.

xv

xv is a popular image manipulation package that consists of approximately 60,000 lines of code. It serves as an interesting test of the performance of tcc on a relatively large application. For the benchmark, one of xv's image-processing algorithms was rewritten in 'C to make use of dynamic code generation. One algorithm is sufficient, since most of the algorithms are implemented similarly. The algorithm, *Blur*, applies a convolution matrix of user-defined size that consists of all 1's to the source image. The original algorithm was implemented efficiently: the values in the convolution matrix are known statically to be all 1's, so convolution at a point is simply the average of the image values of neighboring points. Nonetheless, the inner loop contains image-boundary checks based on run-time constants, and is bounded by a run-time constant, the size of the convolution matrix.

Results from this experiment appear in Table 4.2. For both a 3×3 and 7×7 convolution mask, the dynamic code obtained using tcc and ICODE is approximately 3 times as fast as the static code created by lcc, and approximately 20% faster than the static code generated by gcc with all optimizations turned on. Importantly, the overhead of dynamic code generation is almost three orders of magnitude less than the performance benefit it provides.

xv is an example of the usefulness of dynamic code generation in the context of a well-known application program. Two factors make this result significant. First, the original static code was quite well tuned. Second, the tcc code generator that emits the code-generating code is derived from an lcc code generator, so that the default dynamic code, barring any dynamic optimizations, is considerably less well tuned than equivalent code generated by the GNU compiler. Despite all this,

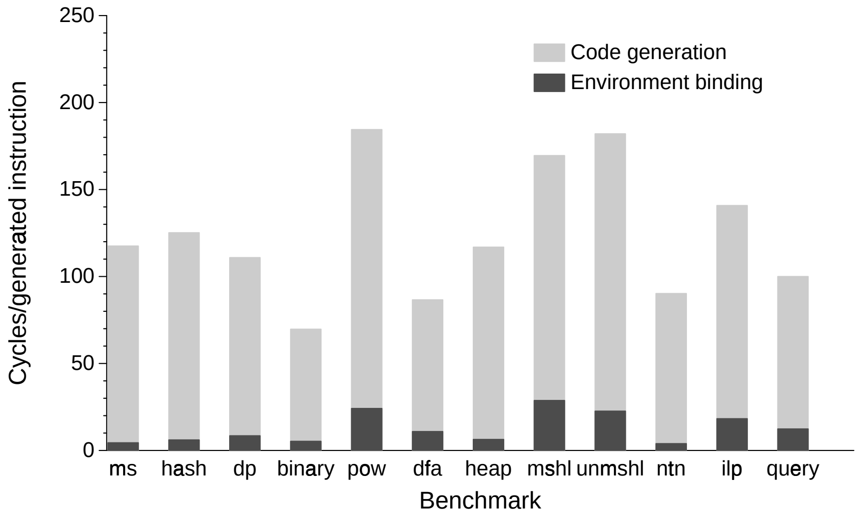

Figure 4-6: Dynamic code generation overhead using VCODE.

the dynamic code is faster than even aggressively optimized static code, and the cost of dynamic code generation is insignificant compared to the benefit obtained.

4.5.2 Analysis

This section analyzes the code generation overhead of VCODE and ICODE. VCODE generates code at a cost of approximately 100 cycles per generated instruction. Most of this time is taken up by register management; just laying out the instructions in memory requires much less overhead. ICODE generates code roughly three to eight times more slowly than VCODE. Much of this overhead is due to register allocation: the choice of register allocator can significantly influence ICODE performance.

VCODE Overhead

Figure 4-6 breaks down the code generation overhead of VCODE for each of the benchmarks. The VCODE back end generates code at approximately 100 cycles per generated instruction: the geometric mean of the overheads for the benchmarks in this chapter is 119 cycles per instruction. The cost of environment binding is small—almost all the time is spent in code generation.

The code generation overhead has several components. The breakdown is difficult to measure precisely and varies slightly from benchmark to benchmark, but there are some broad patterns:

- Laying instructions out in memory (bitwise operations to construct instructions, and stores to write them to memory) accounts for roughly 15% of the overhead.
- Dynamically allocating memory for the code, linking, delay slot optimizations, and prologue and epilogue code add approximately another 25%.
- Register management (VCODE's putreg/getreg operations) accounts for about 50% of the overhead.
- Approximately 10% of the overhead is due to other artifacts, such as checks on the storage class of dynamic variables and calls to code-generating functions.

These results indicate that dynamic register allocation, even in the minimal VCODE implementation, is a major source of overhead. This cost is unavoidable in 'C's dynamic code composition

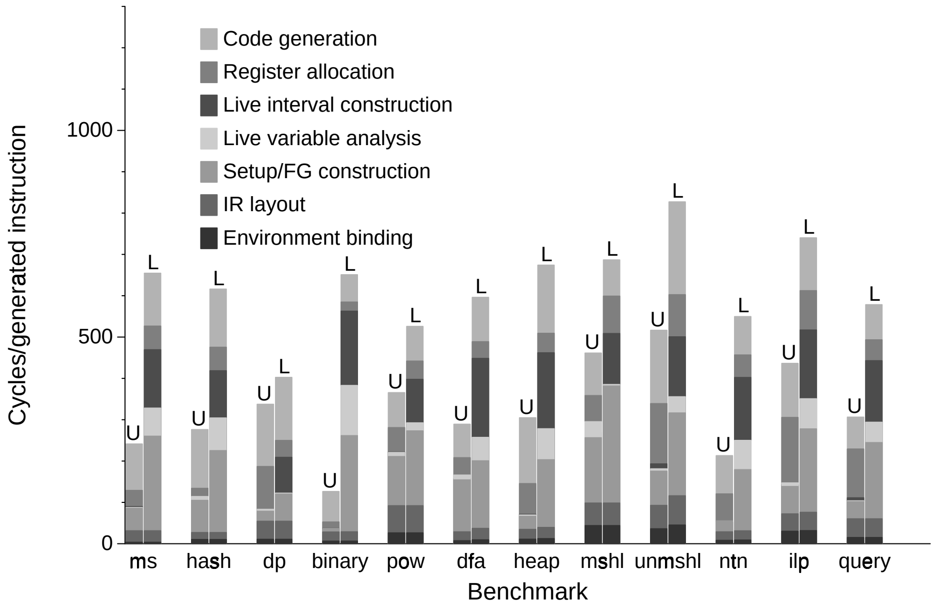

Figure 4-7: Dynamic code generation overhead using ICODE. Columns labeled L denote ICODE with linear scan register allocation; those labeled U denote ICODE with a simple allocator based on usage counts.

model; systems that can statically allocate registers for dynamic code should therefore have a considerable advantage over 'C in terms of dynamic compile-time performance.

ICODE Overhead

Figure 4-7 breaks down the code generation overhead of ICODE for each of the benchmarks. For each benchmark we report two costs. The columns labeled L represent the overhead of using ICODE with linear scan register allocation based on precise live variable information. For comparison, the columns labeled U represent the overhead of using ICODE with the simple allocator that places the variables with the highest usage counts in registers. Data on the performance of both of these algorithms, independently of other code generation costs, appears in Section 5.2.

For each benchmark and type of register allocation, we report the overhead due to environment binding, laying out the ICODE intermediate representation, creating the flow graph and doing some setup (allocating memory for the code, initializing VCODE, etc.), performing various phases of register allocation, and finally generating code.

In addition to register allocation and related liveness analyses, the main sources of overhead are flow graph construction and code generation. The latter roughly corresponds to the VCODE code generation overhead. Environment binding and laying out the ICODE intermediate representation are relatively inexpensive operations.

ICODE's code generation speed ranges from about 200 to 800 cycles per instruction, depending on the benchmark and the type of register allocation. The geometric mean of the overheads for the benchmarks in this chapter, when using linear scan register allocation, is 615 cycles per instruction.

4.6 Summary

This section has described tcc, an experimental implementation of 'C. tcc generates traditional machine code for the static parts of 'C programs as well code-generating code that creates dynamic code at run time. Performance measurements indicate that 'C can improve the performance of several realistic benchmarks by a factor of two to four, and that the overhead of dynamic compilation is usually low enough—about 100 to 600 cycles per generated instruction, depending on the level of optimization—to be recouped within 100 uses of the dynamic code.

The next three chapters will focus on specific algorithms that can improve the performance of dynamic code without incurring excessive dynamic compilation overhead. Chapter 5 discusses three efficient register allocation algorithms. The first is very fast but generates relatively poor code, and is implemented in the VCODE run-time system; the second is slower but generates better code, and is implemented in the ICODE run-time system; the third is still unimplemented, but it promises to merge the best performance features of each of the existing algorithms. Chapter 6 presents some fast approximate data flow analysis algorithms. They are still too slow to be useful in a lightweight dynamic compilation environment like that of 'C, so they are not a part of tcc. However, they are several times faster than traditional data flow analyses, so they may be useful in other dynamic code applications. Finally, Chapter 7 covers fast peephole optimizations. It presents bpo, a peephole optimizer that is part of tcc, and shows that peephole optimizations can be efficient and can significantly improve dynamic code.

Chapter 5

Register Allocation

Good register allocation is an important optimization. It can improve the performance of code by an order of magnitude relative to no or poor register allocation. Unfortunately, as explained in Section 1.3, the code composition inherent to 'C's programming model makes it difficult to perform good register allocation on dynamic code statically. Therefore, an effective 'C implementation requires a good but fast run-time register allocator.

This chapter presents three different register allocation algorithms. The first is a simple one-pass greedy algorithm used in tcc's VCODE run-time system. It is fast but easily makes poor spill decisions. The second algorithm, linear scan, coarsens variable liveness information into approximate live intervals, and allocates registers in one pass over a sorted list of intervals. It is more expensive than the first algorithm, but usually produces better code. It is used in tcc's ICODE optimizing run-time system. The third algorithm is a compromise between the two: it should handle dynamic code composition more gracefully than the first algorithm, but at lower overhead than the second. This third algorithm has not yet been implemented.

5.1 One-Pass Register Allocation

The simplest way to allocate registers at run time is to track available registers with a bit vector. If a physical register is available, it is represented as a one in the bit vector; if it is being used, it is represented as a zero. If no register is available when one is needed, the corresponding variable is placed in a stack location. Once a register is no longer needed, the corresponding bit in the bit vector is turned back on.

This simple algorithm is used by tcc's VCODE run-time system, described in Chapter 4. VCODE maintains a bit vector for each register class (for instance, integer registers and floating point registers), and provides getreg and putreg operations that operate on these bit vectors by allocating and deallocating registers, respectively.

As explained in Chapter 4, tcc creates one code-generating function (CGF) for every cspec. Each CGF contains macros and calls to one of the two run-time systems, VCODE or ICODE, that actually emit dynamic code. The code-generating code that uses VCODE calls the getreg and putreg functions as well as macros that generate machine instructions. For instance, the simple dynamic code

```
'{ int a, b, c; a = b+c; };
```

is generated by VCODE macros that look like this:

```
reg_t r1, r2, r3; /* These typically store values in 0-31 */
v_getreg(&r1, INT, VAR); /* Get a register for int variable a */
v_getreg(&r2, INT, VAR); /* Get a register for int variable b */
v_getreg(&r3, INT, VAR); /* Get a register for int variable c */
v_addi(r1, r2, r3); /* This macro generates an integer add instruction  */
v_putreg(r1, INT); /* Free the register whose number is in r1 */
v_putreg(r2, INT); /* Free the register whose number is in r2 */
v_putreg(r3, INT); /* Free the register whose number is in r3 */
```

cspec composition is implemented by calling the nested cspec's code-generating function from within the outer cspec's CGF. Unfortunately, cspec composition interacts poorly with the simple greedy register allocation scheme. If a register is allocated in the outer cspec and not deallocated until after the call to the nested cspec's CGF, then it is unavailable to the nested cspec and other cspecs nested within it. In this situation, spills will occur after only a few levels of cspec nesting, even though a more global view could avoid them. Furthermore, since deeply nested cspecs frequently correspond to more heavily executed code (fragments of loop bodies, for example), variables in those cspecs may be particularly poor spill candidates.

Another drawback of VCODE's one-pass approach is that it cannot determine the end of the live range of a dynamic variable (an explicitly declared vspec) that is live across multiple cspecs. For instance, if one declares

```
int vspec v = local(int);
```

v is considered live across the entire dynamic function that is being generated. With the exception of the topmost (enclosing) CGF, none of the CGFs that generate code that uses v can reclaim v's dynamic storage location, because v may be used later in code generated by another CGF. As a result, v can end up occupying a valuable register even when it is no longer live.

tcc follows some simple heuristics to improve code quality. First, expression trees are rearranged so that cspec operands of instructions are evaluated before non-cspec operands. This minimizes the number of temporaries that span cspec references, and hence the number of registers allocated by the code-generating function of one cspec during the execution of the CGF of a nested cspec. Second, a cspec does not allocate a register for the return value of a nested non-void cspec. Rather, the CGF for the nested cspec is in charge of allocating the register for the result, and then returns this register name to the CGF of the enclosing cspec. Third, explicitly declared vspecs, such as v in the previous paragraph, are allocated to stack locations unless one explicitly declares them as register variables (for instance, with v = local(register int)).

If after all this there are still no unallocated registers when getreg is invoked, getreg could return a spilled location designated by a negative number; VCODE macros could recognize this number as a stack offset and emit the necessary loads and stores around each instruction. However, such spilling results in poor dynamic code, and the register checks around every VCODE macro decrease compilation performance. Therefore, the current VCODE implementation simply halts dynamic compilation when getreg cannot find a register, and the user can then switch to ICODE.

Section 4.5 discussed the performance of this simple register allocation algorithm. Although the algorithm is quite fast and the heuristics described above make it generate passable code, it often makes rather poor spilling decisions. Linear scan register allocation was developed to solve this problem.

5.2 Linear Scan Register Allocation

The drawback of the algorithm described in the previous section is that local allocation decisions (for example, allocating a register to an expression temporary) can have undesirable global effects (for example, making registers unavailable in cspecs nested in the same expression). An alternative is to use an algorithm that takes a more global view of liveness information.

Unfortunately, global register allocation algorithms for most static optimizing compilers are heavily skewed toward performance of compiled code rather than small compilation overhead. These compilers usually employ a variant of graph coloring register allocation [11], in which the interference graph can have a worst-case size that is quadratic in the number of live ranges.

Linear scan register allocation, by contrast, is a global register allocation algorithm that is not based on graph coloring, but rather allocates registers in a greedy fashion in just one pass of the program's live range representation. The algorithm is simple, efficient, and produces relatively good code: it is up to several times faster than even a fast graph coloring register allocator that performs no coalescing, yet the resulting code is within 12% as fast as code generated by an aggressive graph coloring algorithm for all but two of our benchmarks. Linear scan register allocation is used in the ICODE optimizing run-time system of tcc, and is should be well-suited also to other applications where compile time and code quality are important, such as "just-in-time" compilers and interactive development environments.

The rest of this section first presents some related work on global register allocation, and then describes the algorithm, evaluates its performance, and outlines possible extensions.

5.2.1 Background

Global register allocation has been studied extensively in the literature. The predominant approach, first proposed by Chaitin [11], is to abstract the register allocation problem as a graph coloring problem. Nodes in the graph represent live ranges (variables, temporaries, virtual registers) that are candidates for register allocation. Edges connect live ranges that *interfere*—those that are live simultaneously in at least one program point. Register allocation then reduces to the graph coloring problem in which colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color. If the graph is not colorable, some nodes are deleted from the graph until the reduced graph becomes colorable. The deleted nodes are said to be *spilled* because they are not assigned to registers. The basic goal of register allocation by graph coloring is to find a legal coloring after deleting the minimum number of nodes (or more precisely, after deleting a set of nodes with minimum total spill cost).

Chaitin's algorithm also features *coalescing*, a technique that can be used to eliminate redundant moves. When the source and destination of a move instruction do not share an edge in the interference graph, the corresponding nodes can be coalesced into one, and the move eliminated. Unfortunately, aggressive coalescing can lead to uncolorable graphs, in which additional live ranges need to be spilled to memory. More recent work on graph coloring ([10], [36]) has focused on removing unnecessary moves in a conservative manner so as to avoid introducing spills.

Some simpler heuristic solutions also exist for the global register allocation problem. For example, lcc [31] allocates registers to the variables with the highest estimated usage counts, places all others on the stack, and allocates temporary registers within an expression by doing a tree walk.

Linear scan can be viewed as a global extension of a special class of local register allocation algorithms that have been considered in the literature [35, 42, 31, 53], which in turn take their inspiration from an optimal off-line replacement algorithm that was studied for virtual memory [5].

Since the original description of the linear scan algorithm in Poletto et al. [56], Traub et al. have proposed a more complex linear scan algorithm, which they call *second-chance binpacking* [73]. This algorithm is an evolution and refinement of *binpacking*, a technique used for several years in the DEC GEM optimizing compiler [9]. At a high level, the binpacking schemes are similar to linear scan, but they invest more time in compilation in an attempt to generate better code. The second-chance binpacking algorithm both makes allocation decisions and rewrites code in one pass. The algorithm allows a variable's lifetime to be split multiple times, so that the variable resides in a register in some parts of the program and in memory in other parts. It takes a lazy approach to spilling, and never emits a store if a variable is not live at that particular point or if the register and memory values of the variable are consistent. At every program point, if a register must be used to hold the value of a variable v_1, but v_1 is not currently in a register and all registers have been allocated to other variables, the algorithm evicts a variable v_2 that is allocated to a register. It tries to find a v_2 that is not currently live (to avoid a store of v_2), and that will not be live before the end of v_1's live range (to avoid evicting another variable when both v_1 and v_2 become live).

Binpacking can emit better code than linear scan, but it does more work at compile time. Unlike linear scan, binpacking keeps track of the "lifetime holes" of variables and registers (intervals when a variable maintains no useful value, or when a register can be used to store a value), and maintains information about the consistency of the memory and register values of a reloaded variable. The algorithm analyzes all this information whenever it makes allocation or spilling decisions. Furthermore, unlike linear scan, it must perform an additional "resolution" pass to resolve any conflicts between the non-linear structure of the control flow graph and the assumptions made during the linear register allocation pass. Section 5.2.3 compares the performance of the two algorithms and of the code that they generate.

5.2.2 The Algorithm

First, we must establish some conventions. This section assumes a program intermediate representation that consists of RTL-like quads or pseudo-instructions. Register candidates (live ranges) are represented by an unbounded set of "virtual registers" that can be used directly in arithmetic operations. Variables are not live on entry to the start node in the flow graph; initialization of procedure parameters is captured by explicit assignment statements.

The linear scan algorithm assumes that the intermediate representation pseudo-instructions are numbered according to some order. One possible ordering is that in which the pseudo-instructions appear in the intermediate representation. Another is depth-first ordering, the reverse of the order in which nodes are last visited in a preorder traversal of the flow graph [1]. This section uses depth-first order. The choice of instruction ordering does not affect the correctness of the algorithm, but it may affect the quality of allocation. Section 5.2.4 discusses alternative orderings.

Central to the algorithm is the notion of a *live interval.* Given some numbering of the intermediate representation, $[i, j]$ is said to be a live interval for variable v if there is no instruction with number $j' > j$ such that v is live at j', and there is no instruction with number $i' < i$ such that v is live at i'. This information is a conservative approximation of live ranges: there may be subranges of $[i, j]$ in which v is not live, but they are ignored. The "trivial" live interval for any variable v is $[1, N]$, where N is the number of pseudo-instructions in the intermediate representation: this live interval is correct and takes no time to compute, but it also yields no information. All other live intervals lie on the spectrum between the trivial live interval and accurate live interval information. The order chosen for numbering pseudo-instructions influences the extent and accuracy of live intervals, and hence the quality of register allocation, but the definition of live intervals does not rely on or make assumptions about a particular numbering.

Given live variable information (obtained, for example, via data-flow analysis [1]), live intervals can be computed easily with one pass through the intermediate representation. Interference among live intervals is captured by whether or not they overlap. Given R available registers and a list of live intervals, the linear scan algorithm must allocate registers to as many intervals as possible, but such that no two overlapping live intervals are allocated to the same register. If $n > R$ live intervals overlap at any point, then at least $n - R$ of them must reside in memory.

Details

The number of overlapping intervals changes only at the start and end points of an interval. Live intervals are stored in a list that is sorted in order of increasing start point. Hence, the algorithm can quickly scan forward through the live intervals by skipping from one start point to the next.

At each step, the algorithm maintains a list, *active*, of live intervals that overlap the current point and have been placed in registers. The *active* list is kept sorted in order of increasing end point. For each new interval, the algorithm scans *active* from beginning to end. It removes any "expired" intervals—those intervals that no longer overlap the new interval because their end point precedes the new interval's start point—and makes the corresponding register available for allocation. Since *active* is sorted by increasing end point, the scan needs to touch exactly those elements that need to be removed, plus at most one: it can halt as soon as it reaches the end of *active* (in which case *active* remains empty) or encounters an interval whose end point follows the new interval's start point.

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Figure 5-1: Linear scan register allocation. Indentation denotes nesting level. Live intervals (including *startpoint* and *endpoint* information) have been computed by a prior liveness analysis phase.

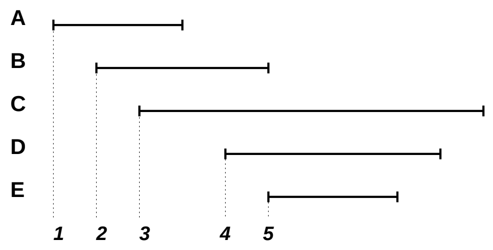

Figure 5-2: An example set of live intervals. Letters on the left are variable names; the corresponding live intervals appear to the right. Numbers in italics refer to steps in the linear scan algorithm described in the text.

The length of the *active* list is at most R. The worst case scenario is that *active* has length R at the start of a new interval and no intervals from *active* are expired. In this situation, one of the current live intervals (from *active* or the new interval) must be spilled. There are several possible heuristics for selecting a live interval to spill. The heuristic described here is based on the remaining length of live intervals. The algorithm spills the interval that ends last, furthest away from the current point. It can find this interval quickly because *active* is sorted by increasing end point: the interval to be spilled is either the new interval or the last interval in *active*, whichever ends later. In straight-line code, and when each live interval consists of exactly one definition followed by one use, this heuristic produces code with the minimal possible number of spilled live ranges [5, 53]. Although in our case a live interval may cover arbitrarily many definitions and uses spread over different basic blocks, the heuristic still appears to work well. Figure 5-1 contains the pseudocode for the linear scan algorithm with this heuristic. All results in Section 5.2.3 are also based on this heuristic.

An Example

Consider, for example, the live intervals in Figure 5-2 for the case when the number of available registers is $R = 2$. The algorithm performs allocation decisions 5 times, once per live interval, as denoted by the italicized numbers at the bottom of the figure. By the end of step *2*, $active = \langle A, B \rangle$ and both A and B are therefore in registers. At step *3*, three live intervals overlap, so one variable must be spilled. The algorithm therefore spills C, the one whose interval ends furthest away from the current point, and does not change *active*. As a result, at step *4*, A is expired from *active*, making a register available for D, and at step *5*, B is expired, making a register available for E. Thus, in the end, C is the only variable not allocated to a register. Had the algorithm not spilled the longest interval, C, at step *3*, both one of A and B and one of D and E would have been spilled to memory.

Complexity

Let V be the number of variables (live intervals) that are candidates for register allocation, and R be the number of registers available for allocation. As can be seen from the pseudocode in Figure 5-1, the length of *active* is bounded by R, so the linear scan algorithm takes $O(V)$ time if R is assumed to be a constant.

Since R can be large in some current or future processors, it is worthwhile understanding how the complexity depends on R. Recall that the live intervals in *active* are sorted in order of increasing endpoint. The worst-case execution time complexity of the linear scan algorithm is dictated by the time taken to insert a new interval into *active*. If a balanced binary tree is used to search for the insertion point, then the insertion takes $O(\log R)$ time and the entire algorithm takes $O(V \times \log R)$ time. An alternative is to do a linear search for the insertion point, which takes $O(R)$ time, thus leading to a worst case complexity of $O(V \times R)$ time. This is asymptotically slower than the previous result, but may be faster for moderate values of R because the data structures involved are much simpler. The implementations evaluated in Section 5.2.3 use a linear search.

5.2.3 Evaluation

This section evaluates linear scan register allocation in terms of both compile-time performance and the quality of the resulting code. We first describe our methodology, then evaluate the compile-time performance of the algorithm, and finally discuss the run-time performance of the generated code. As described below, each part of the evaluation was performed with two different implementations and benchmark infrastructures.

Methodology

The data for this evaluation was obtained using two systems. ICODE, tcc's optimizing run-time system, was used primarily to measure compile-time performance. SUIF [2], on the other hand, provided a good platform for measuring run-time performance of the generated code.

Performance of the Register Allocation Algorithm. Linear scan is the default register allocation algorithm in ICODE, tcc's optimizing run-time system. Evaluating the algorithm in ICODE is a convincing benchmark of compile-time performance because ICODE is already well-tuned for efficient compilation. We use two sets of benchmarks to evaluate our ICODE implementation. The first set consists of the 'C applications described in Section 4.4. For each of these benchmarks, we compare linear scan register allocation against two other algorithms implemented in ICODE: a well-tuned graph coloring algorithm and a simple "usage count" register allocation scheme. The graph coloring algorithm tries to be fast without overly penalizing code quality: it does not do coalescing, but employs estimates of usage counts to guide spilling. The live variable information used by this allocator is obtained by an iterative data-flow pass over the ICODE flow graph. Both the liveness analysis and the register allocation pass were carefully implemented for speed. The "usage count" algorithm ignores liveness information altogether. It allocates the k available registers to the k variables and compiler-generated temporaries with the highest estimated usage counts, and places all others on the stack.

The second set of benchmarks consists of pathological programs that perform no useful computation but have huge numbers of simultaneously live variables that make register allocation difficult. We use these benchmarks to compare the performance of graph coloring and linear scan as the size of the allocation problem increases.

All these measurements were performed on the hardware and with the methodology described in Section 4.4.

Performance of the Resulting Code. The 'C benchmarks are all relatively small, so their run-time performance is similar for all the register allocation algorithms. In order to measure the effect of linear scan on the performance of larger, well-known programs, it was necessary to implement it in Machine SUIF [66], an optimizing scalar back end infrastructure for SUIF [2]. This implementation served to compile various SPEC benchmarks (from both the SPEC92 and SPEC95 suites) and two UNIX utilities. As before, linear scan register allocation is compared against a graph coloring algorithm and the simple algorithm based on usage counts. It is also compared against second-chance binpacking [73]. The graph coloring allocator is an implementation of iterated register coalescing [36] developed at Harvard. For completeness, this section also reports the compile-time performance of the SUIF implementation of binpacking and linear scan in Section 5.2.3, even though the underlying SUIF infrastructure has not been designed for efficient compile times.

All benchmarks were compiled with SUIF and Machine SUIF. Measurements are the user time from the best of ten runs on an unloaded DEC Alpha workstation with a 500MHz Alpha 21164 processor and 128MB of RAM.

Compile-Time Performance

ICODE Implementation. Figure 5-3 illustrates the overhead of register allocation for the dynamic code kernels described in Section 5.2.3. The vertical axis measures compilation overhead, in cycles per generated instruction. Larger values indicate larger overhead. The horizontal axis of the

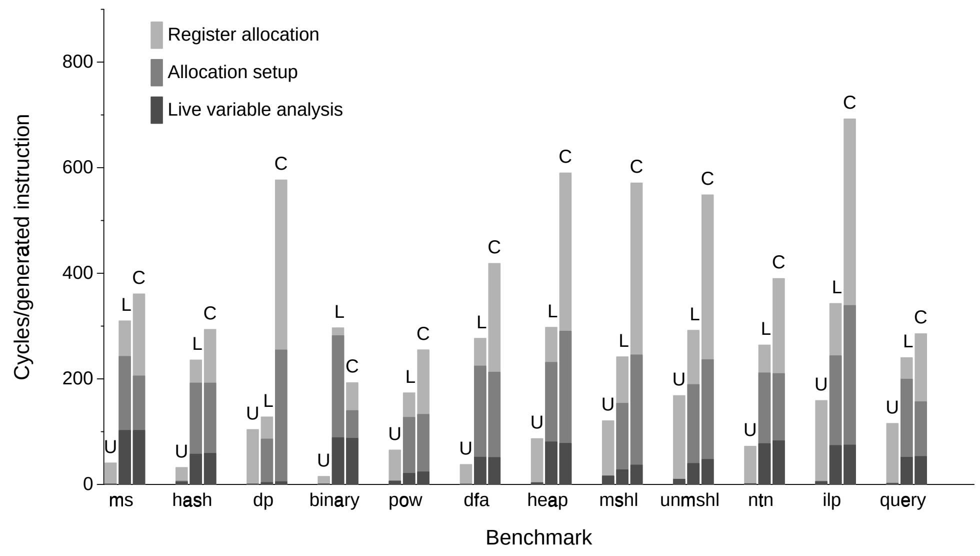

Figure 5-3: Register allocation overhead for 'C benchmarks (ICODE infrastructure). U denotes a simple algorithm based on usage counts. L denotes linear scan. C denotes graph coloring.

figure denotes different benchmarks written in 'C. For each benchmark, there are three bars: U refers to the usage count algorithm; L refers to linear scan register allocation; C refers to graph coloring.

Each bar contains up to three different regions:

1. **Live variable analysis:** refers to traditional iterative live variable analysis, and hence does not appear in the U column.
2. **Allocation setup:** refers to work necessary prior to register allocation. It does not apply to U. In the case of L, it refers to the construction of live intervals by coarsening live variable information obtained through live variable analysis. In the case of C, it refers to construction of the interference graph.
3. **Register allocation:** in the case of U, it involves sorting variables by usage count, and allocating registers to the most used ones until none are left. In the case of L, it refers to linear scan of the live intervals. In the case of C, it refers to coloring the interference graph.

Liveness analysis and allocation setup for the U case are essentially null function calls. Small positive values for these two phases, as well as small differences in the live variable analysis overheads in the L and C cases, are due to slight variability in the getrusage measurements. Times for individual compilation phases were obtained by repeatedly interrupting compilation after the phase of interest, subtracting the time required up to the previous phase, and dividing by the number of (interrupted) compiles.

The figure indicates that linear scan allocation (L) can be considerably faster than even a simple and fast graph coloring algorithm (C). In particular, although creating live intervals from live variable information is roughly similar to building an interference graph from live variable information, linear scan of live intervals is always much faster than coloring the interference graph. The one benchmark in which graph coloring is faster than linear scan is binary. In this case, the code uses few variables but consists of many basic blocks, so it is faster to build the small interference graph than to extract live intervals from liveness information at each basic block. However, note that even for binary the actual time spent on register allocation is smaller for linear scan (L) than for graph coloring (C).

SUIF Implementation. Table 5.1 compares the compile-time performance of the SUIF implementation of binpacking and linear scan on representative files from the benchmark set. It does not

File (Benchmark)	Time in seconds		Ratio
	Linear scan	Binpacking	(Binpacking / linear scan)
swim.f (swim)	0.42	1.07	2.55
xllist.c (li)	0.31	0.60	1.94
xleval.c (li)	0.14	0.29	2.07
tomcatv.f (tomcatv)	0.19	0.48	2.53
compress.c (compress)	0.14	0.32	2.29
cvrin.c (espresso)	0.61	1.14	1.87
backprop.c (alvinn)	0.07	0.19	2.71
fpppp.f (fpppp)	3.35	4.26	1.27
twldrv.f (fpppp)	1.70	3.49	2.05

Table 5.1: Register allocation overhead for static C benchmarks using linear scan and binpacking (SUIF infrastructure).

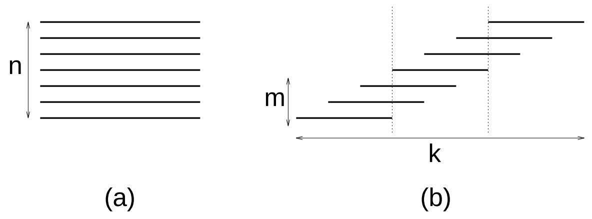

Figure 5-4: Two types of pathological programs.

present data for graph coloring: Traub et al. [73] and Section 5.2.3 provide convincing evidence that both binpacking and linear scan are much faster than graph coloring, especially as the number of register candidates grows.

The times in Table 5.1 refer to only the core allocation routines: they include neither setup activities such as CFG construction and liveness analysis, nor any compilation phase after allocation. In most cases, linear scan is roughly two to three times faster than binpacking.

These results, however, under-represent the difference between the two algorithms. For simplicity, the linear scan implementation uses the binpacking routine for computing "lifetime holes" [73]. However, linear scan does not need or use full information on lifetime holes—it just considers the start and end of each variable's life interval. As a result, an aggressive linear scan implementation could be considerably faster. For example, if one does not count lifetime hole computation, the compilation overhead for fpppp.f is 2.79s with binpacking and 1.88s with linear scan, and that of twldrv.f is 2.28s with binpacking and 0.49s with linear scan.

Pathological Cases. The ICODE framework was also used to compile pathological programs intended to stress register allocators. This study considers programs with two different kinds of structure, as illustrated in Figure 5-4. One kind, labeled (a) in the figure, contains some number, n, of overlapping live intervals (simultaneously live variables). The other kind, labeled (b), contains k staggered "sets" of live intervals in which no more than m live intervals overlap.

Figure 5-5 illustrates the overhead of graph coloring and linear scan as a function of the number n of overlapping live intervals in code of type (a). Both axes are logarithmic. The horizontal axis indicates problem size; the vertical axis indicates time. Although the costs of graph coloring and linear scan are comparable when the number of overlapping live intervals is small, linear scan scales much more gracefully to large problem sizes. With 512 simultaneously live variables, linear scan is over 600 times faster than graph coloring. Unlike linear scan, graph coloring appears to suffer

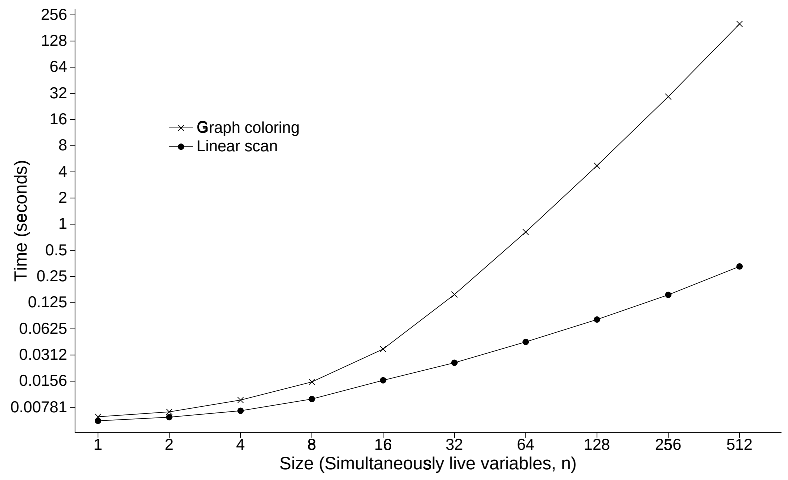

Figure 5-5: Overhead of graph coloring and linear scan as a function of the number of simultaneously live variables for programs of type (a) (ICODE infrastructure). Both algorithms generate the same number of spills.

from the $O(n^2)$ time required to build and color the interference graph. Importantly, the reported overhead is for the entire code generation process—not just allocating registers, but also setting up the intermediate representation, computing live variables, and generating code, so both algorithms share a common fixed cost that reduces the relative performance gap between them. Furthermore, the code generated by both allocators for this pathological case contains the same number of spills.

Figure 5-6 compares the overhead of graph coloring and linear scan for programs with live interval patterns of type (b). As in the previous experiment, linear scan in this case generates the same number of spills as graph coloring. Again, the axes are logarithmic and the vertical axis indicates time. The horizontal axis denotes the number of successive staggered sets of live intervals, k in Figure 5-4b. Different curves denotes different numbers of simultaneously live variables (m in Figure 5-4b): for example, "Linear Scan (m=24)" refers to linear scan allocation with $m = 24$. With increasing k, the overhead of graph coloring grows more quickly than that of linear scan. Moreover, the vertical space between graph coloring curves for increasing m grows more quickly than for the corresponding linear scan curves. This data is consistent with the results in Figure 5-5: the performance of graph coloring degrades as the number of simultaneously live variables increases.

Run-time Performance

ICODE Implementation. Figure 5-7 shows the run-time performance of code compiled with the ICODE implementation of the algorithms. As before, the horizontal axis denotes different benchmarks, and for each benchmark the different bars denote different register allocation algorithms, labeled as in Figure 5-3. The vertical axis is logarithmic, and indicates run time in seconds. Unfortunately, these dynamic code kernels are small, and do not have enough register pressure to illustrate the differences among the allocation algorithms. The three algorithms generate code of similar quality for all benchmarks other than dfa and heap. In these two cases, the code emitted by the simple allocator based on usage count is considerably slower than that created by graph coloring or linear scan.

SUIF Implementation. Figure 5-8 presents the run time of several large benchmarks compiled with the SUIF implementation of the algorithms. Once again, the horizontal axis denotes different benchmarks, and the logarithmic vertical axis measures run time in seconds. In addition to the three algorithms (U, L, and C) measured so far, the figure also presents data for second-chance

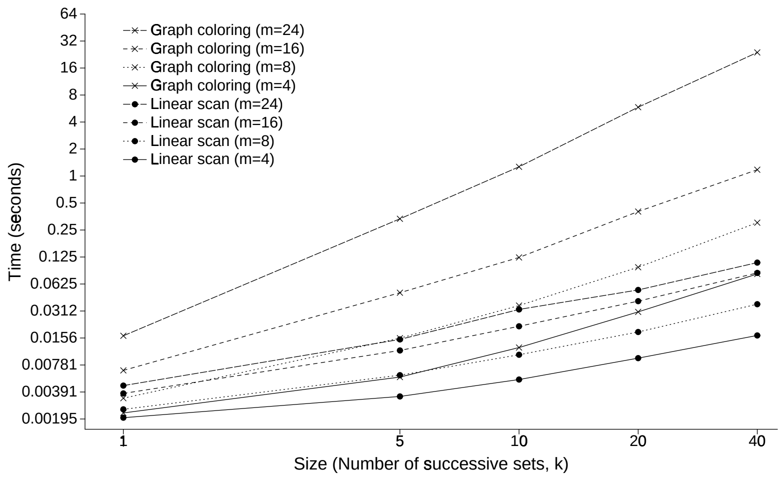

Figure 5-6: Overhead of graph coloring and linear scan as a function of program size for programs of type (b) (ICODE infrastructure). The horizontal axis denotes the number of staggered sets of intervals (k in Figure 4b). Different curves denote values for different numbers of simultaneously live variables (m in Figure 4b). Both algorithms generate the same number of spills.

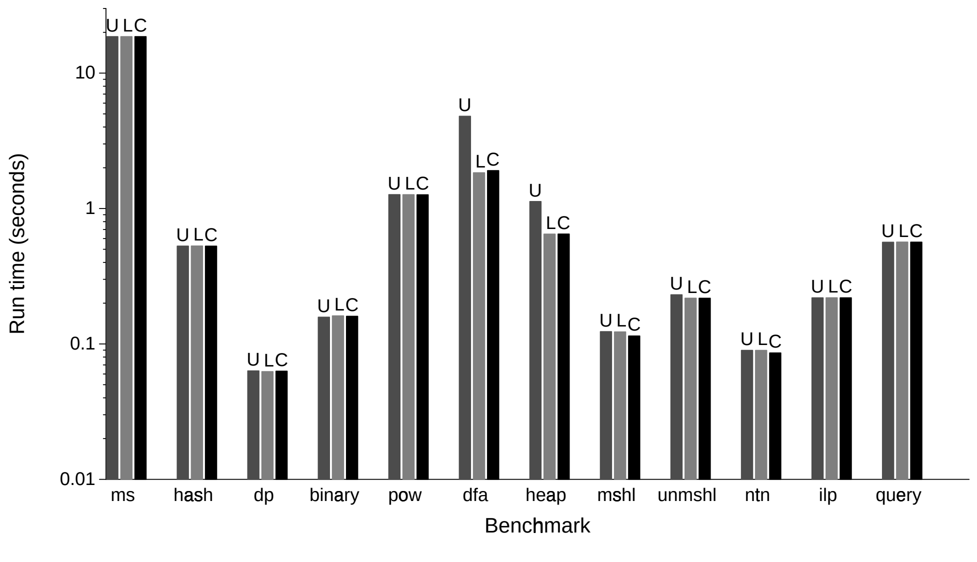

Figure 5-7: Run time of ‘C benchmarks compiled with different register allocation algorithms (ICODE infrastructure). U denotes the simple scheme based on usage counts. L denotes linear scan. C denotes graph coloring.

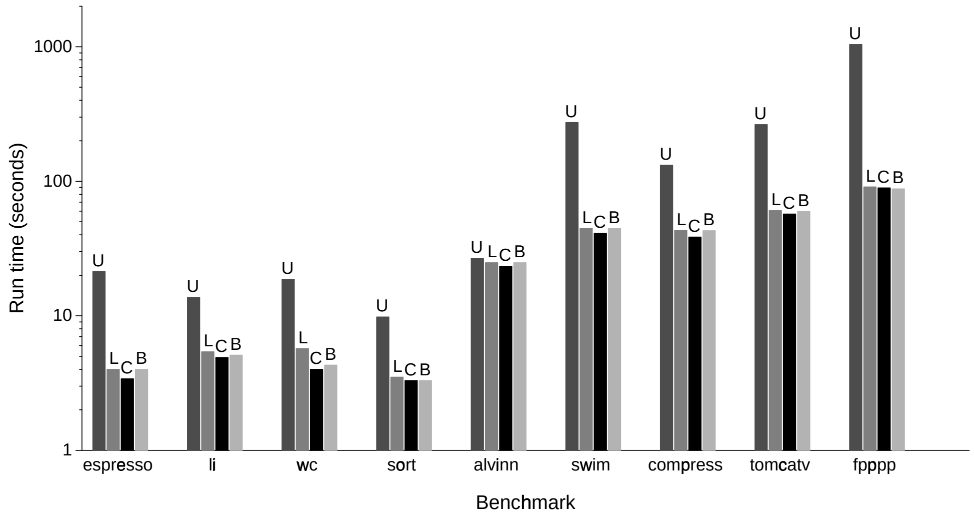

Figure 5-8: Run times of static C benchmarks compiled with different register allocation algorithms (SUIF infrastructure). U, L, and C are as before. B denotes second-chance binpacking.

binpacking [73], labeled B. Table 5.2 contains the same data, and also provides the ratio of the run time of each benchmark compiled with each register allocation method relative to the run time of that benchmark compiled with graph coloring.

The measurements in Figure 5-8 and Table 5.2 indicate that linear scan makes a fair performance tradeoff. It is considerably simpler and faster than graph coloring and binpacking, yet it usually generates code that runs within 10% of the speed of that generated by the two more complicated algorithms, and several times faster than that generated by the simple usage count allocator.

5.2.4 Discussion

Since it is so simple, linear scan allocation lends itself to several extensions and optimizations. This section describes a fast algorithm for conservative (approximate) live interval analysis, discusses the effect of different flow graph numberings and spilling heuristics, mentions some architectural considerations, and outlines possible future refinements to linear scan allocation.

Benchmark	**Time in seconds (ratio to graph coloring)**			
	Usage counts	Linear scan	Graph coloring	Binpacking
espresso	21.3 (6.26)	4.0 (1.18)	3.4 (1.00)	4.0 (1.18)
compress	131.7 (3.42)	43.1 (1.12)	38.5 (1.00)	42.9 (1.11)
li	13.7 (2.80)	5.4 (1.10)	4.9 (1.00)	5.1 (1.04)
alvinn	26.8 (1.15)	24.8 (1.06)	23.3 (1.00)	24.8 (1.06)
tomcatv	263.9 (4.62)	60.5 (1.06)	57.1 (1.00)	59.7 (1.05)
swim	273.6 (6.66)	44.6 (1.09)	41.1 (1.00)	44.5 (1.08)
fpppp	1039.7 (11.64)	90.8 (1.02)	89.3 (1.00)	87.8 (0.98)
wc	18.7 (4.67)	5.7 (1.43)	4.0 (1.00)	4.3 (1.07)
sort	9.8 (2.97)	3.5 (1.06)	3.3 (1.00)	3.3 (1.00)

Table 5.2: Run time of static C benchmarks, as a function of register allocation algorithm (SUIF infrastructure).

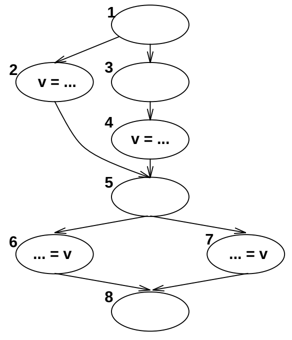

Figure 5-9: An acyclic flow graph. Nodes are labeled with their depth-first numbers.

Fast Live Interval Analysis

Figure 5-3 shows that most of the overhead of linear scan register allocation is due to live variable analysis and "allocation setup," the coarsening of live variable information into live intervals. This section considers an alternative algorithm that trades accuracy for speed, and quickly builds a conservative approximation of live intervals without requiring full iterative live variable analysis.

The algorithm is based on the decomposition of the flow graph into strongly connected components, so let us call it "SCC-based liveness analysis." It relies on two simple observations. First, consider an *acyclic* flow graph in which nodes are numbered in depth-first order (also known as "reverse postorder" [1]), as shown in the example in Figure 5-9. Recall that this order is the reverse of the order in which nodes are last visited, or "finished" [15], in a preorder traversal of the graph. If the assignment to a variable v with the smallest depth-first number (DFN) has DFN i, and the use with the greatest DFN has DFN j, then $[i, j]$ is a live interval of v. For example, in Figure 5-9, a conservative live interval of v is $[2, 7]$. The second observation pertains to cyclic flow graphs: when all the definitions and uses of a variable v appear within a single strongly connected component, C, of the flow graph, the live interval of v will span at most exactly C.

As a result, the algorithm can compute conservative live intervals as follows. (1) Compute SCCs of the flow graph, and for each SCC, construct the set of variables used or defined in it. Also obtain each SCC's DFN in the (acyclic) SCC graph. (2) Traverse the SCCs once, extending the live interval of each variable v to $[i, j]$, where i and j are, respectively, the smallest and largest DFNs of any SCCs that use or define v.

This algorithm is appealing because it is simple and it minimizes expensive bit-vector operations common in live variable analysis. The improvements in compile-time relative to linear scan are impressive, as illustrated in Figure 5-10.

Unfortunately, however, the quality of generated code suffers from this approximate analysis. The difference is minimal for the small 'C benchmarks, but becomes prohibitive for large benchmarks. Table 5.3 compares the run time of applications compiled with full live variable analysis to that of applications compiled with SCC-based liveness analysis. These results indicate that SCC-based liveness analysis may be of interest for quickly compiling small functions, but that it is not suitable as a replacement for full live variable analysis in large programs.

Numbering Heuristics

As mentioned in Section 5.2.2, the definition of live intervals used in linear scan allocation holds for any numbering of flow graph nodes, not just the depth-first numbering discussed so far.

This section used depth-first order because it is the most natural, and it supports SCC-based liveness analysis. Another reasonable alternative is linear, or layout, order, i.e. the order in which the pseudo-instructions appear in the intermediate representation. As shown in Table 5.4, linear and depth-first order produce roughly similar code for our set of benchmarks.

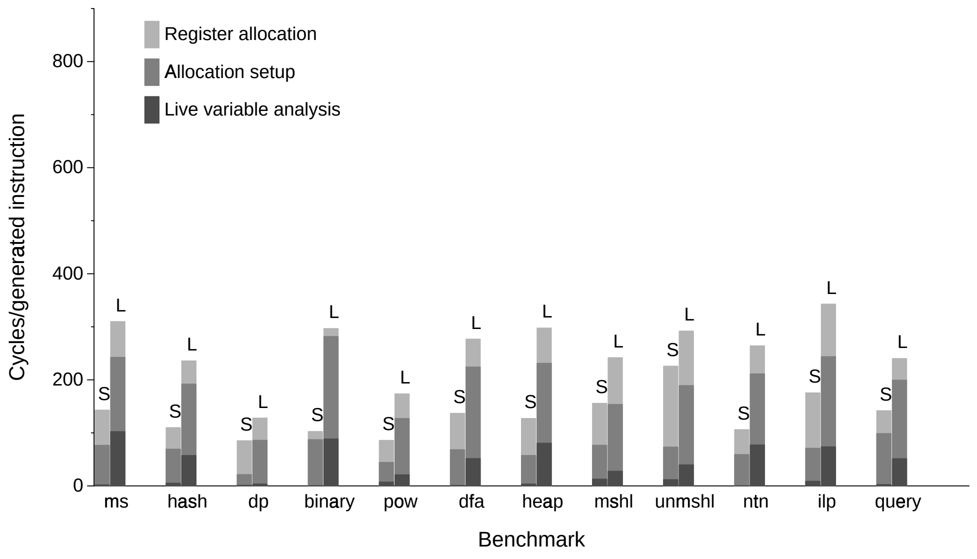

Figure 5-10: Comparison of register allocation overhead of linear scan with full live variable analysis (L) and SCC-based liveness analysis (S) (ICODE infrastructure).

Benchmark	**Time in seconds (ratio to graph coloring)**	
	SCC-based analysis	Full liveness analysis
espresso	22.7 (6.68)	4.0 (1.18)
compress	134.4 (3.49)	43.1 (1.12)
li	14.2 (2.90)	5.4 (1.10)
alvinn	40.2 (1.73)	24.8 (1.06)
tomcatv	290.8 (5.09)	60.5 (1.06)
swim	303.5 (7.38)	44.6 (1.09)
fpppp	484.7 (5.43)	90.8 (1.02)
wc	23.2 (5.80)	5.7 (1.43)
sort	10.6 (3.21)	3.5 (1.06)

Table 5.3: Run time of static C benchmarks compiled with linear scan allocation, as a function of liveness analysis technique (SUIF infrastructure).

Benchmark	**Time in seconds**	
	Depth-first	Linear (layout)
espresso	4.0	4.0
compress	43.3	43.6
li	5.3	5.5
alvinn	24.9	25.0
tomcatv	60.9	60.4
swim	44.8	44.4
fpppp	90.8	91.1
wc	5.7	5.8
sort	3.5	3.6

Table 5.4: Run time of static C benchmarks compiled with linear scan allocation, as a function of flow graph numbering (SUIF infrastructure).

Benchmark	**Time in seconds**	
	Interval length	Interval weight
espresso	4.0	4.0
compress	43.1	43.0
li	5.4	5.4
alvinn	24.8	24.8
tomcatv	60.5	60.2
swim	44.6	44.6
fpppp	90.8	198.6
wc	5.7	5.7
sort	3.5	3.5

Table 5.5: Run time of static C benchmarks compiled with linear scan allocation, as a function of spilling heuristic (SUIF infrastructure).

Spilling Heuristics

The spilling heuristic presented in Section 5.2.2 uses interval length. A potential alternative spilling heuristic is based on interval weight, or estimated usage count. In this case, the algorithm spills the interval with the least estimated usage count among the new interval and the intervals in *active*.

Table 5.5 compares the run time of programs compiled using interval length and interval weight spilling heuristics. In general, the results are similar; only in one benchmark, fpppp, does the interval length heuristic significantly outperform interval weight. Of course, the relative performance of the two heuristics depends entirely on the structure of the program being compiled. The interval length heuristic has the additional advantage that it is slightly simpler, since it does not require maintaining usage count information.

Architectural Considerations

Many machines place restrictions on the use of registers: for instance, only certain registers may be used to pass arguments or return results, or certain operations must target specific registers.

Operations that target specific registers can be handled by pre-allocating the register candidates that are targets of these instructions, and modifying the allocation algorithm to take the pre-allocation into account. In the case of linear scan, if the scan encounters a pre-allocated live interval, it must spill (or assign to a different register) the interval in *active*, if any, that already uses the register of the pre-allocated interval. However, because live intervals are so coarse, it is possible that two intervals that both need to use a particular register overlap: in this case, the best solution is to extend linear scan so that a variable may reside in different locations throughout its lifetime, as in binpacking [73].

The issue of when to use caller-saved registers is solved elegantly by the binpacking algorithm, which extends the concept of lifetime holes to physical registers [73]. The lifetime hole of a register is any region in which that register is available for allocation; the lifetime holes of caller-saved registers simply do not include function calls. Linear scan does not use lifetime holes. The simplest solution is to use all registers, and insert saves and restores where appropriate around function calls after register allocation. Another solution is to actually introduce the binpacking concept of register lifetime hole, and to allocate a live interval to a register only if it fits entirely within the register's lifetime hole. The ICODE implementation uses the former solution, and the SUIF implementation uses the latter one.

Optimizations

The most beneficial optimization in terms of code quality is probably live interval splitting. Splitting does not involve changes to linear scan itself, but only to the definition of live intervals. With splitting, a variable has one live interval for each region of the flow graph throughout which it is uninterruptedly live, rather than one interval for the entire flow graph. This definition takes advantage of holes in variable lifetimes, and is analogous to binpacking without the second-chance technique [73]. Past work on renaming scalar variables, including renaming into SSA form [16], can be useful in subsuming much of the splitting that might be useful for register allocation.

Another possible optimization is coalescing of register moves. If the live interval of a variable v_1 ends where the live interval of another variable, v_2, begins, and the program at that point contains a copy $v_2 \leftarrow v_1$, then v_2 can be assigned v_1's register, and if v_2 is not subsequently spilled, the move can be eliminated after register allocation. It is not difficult to extend the routine ExpireOldIntervals in Figure 5-1 to enable this optimization. However, in order to be effective, move coalescing depends on live interval splitting: without splitting, the opportunities for coalescing are few and can occur only outside of loops.

Although some of these extensions are interesting, none of them have been implemented. The key advantage of the linear scan algorithm is that it is fast and simple, yet can produce relatively good code. Additions to linear scan would make it slower and more complicated, and may not improve the generated code much. If one is willing to sacrifice some compile-time performance to obtain better code, then the second-chance binpacking algorithm might be a better alternative.

5.3 Two-Pass Register Allocation

The two alternatives described so far are somewhat opposite points of the design space: the simple algorithm of Section 5.1 is very lightweight and uses no global information, whereas the linear scan algorithm of Section 5.2, although faster than traditional algorithms, requires various data structures and is several times more expensive than the simple algorithm. This section describes a compromise solution inspired by the two-pass approach of the Sethi-Ullman register allocation algorithm. This two-pass algorithm has not been implemented yet, but it could be interesting future work. It promises to generate good code at low overhead despite 'C's dynamic code composition model.

The Sethi-Ullman algorithm [65] uses dynamic programming to perform optimal register allocation for expression trees. The first part of the algorithm traverses the tree and computes in bottom-up fashion the number of registers required by each subtree. For instance, consider a node p that has two children, l and r, which require n_l and n_r registers respectively. If $n_l = n_r$, then the number of registers required by p is $n_p = n_l + 1$: one register is needed to store the result of r while l is being evaluated. If $n_l \neq n_r$, then the subtree with greater register needs can be evaluated first. For example, if $n_l > n_r$, then $n_l \geq n_r + 1$, so the extra register needed to temporarily store the result of l while r is being computed is hidden by the overhead of l. As a result, $n_p = \max(n_l, n_r)$. When all register requirements are computed, the second part of the algorithm traverses the expression tree once again, assigning registers and inserting spills according to the register need information computed in the first phase.

In retrospect, many of the benefits of linear scan allocation for ‘C might be obtained without resorting to an intermediate representation such as ICODE's, but simply applying a two-pass approach like that of the Sethi-Ullman algorithm to tcc's code-generating functions. The code generated by the CGFs is not a single expression tree, but the structure of cspec composition is indeed a tree, albeit one with arbitrary and irregular branching factors. This tree is exactly the CGF call tree. Of course, code generated by the CGFs may have side effects, so the call tree cannot be rearranged in the same way that one rearranges expression trees in the Sethi-Ullman algorithm. However, the latter's two-pass approach is still valuable.

The live range of a variable is represented implicitly by the CGF call tree—a temporary defined inside a given cspec cannot be live in an enclosing cspec, and will not be referenced in an enclosed cspec—so it should not require the creation of explicit live intervals or other data structures. The trick is to summarize the register requirements of nested cspecs in a compact way—with integers. The algorithm is based on code-generating functions that use VCODE macros and are extended in simple ways. In the first pass, all CGFs are invoked as normal, but a flag ensures that no VCODE macros are executed and that therefore no code is generated. Instead, each CGF returns to its caller the number of registers that it needs. How to compute the "right" value is not as obvious as in the Sethi-Ullman case. For example, the CGF might return more detailed information, such as the total number of registers it would like and the fraction of those that it absolutely needs for performance. Consider the following dynamic code specification, in which the loop body incorporates a cspec denoted by c:

```
‘{ ... while (condition) { ... @c; ... } ... };
```

Assume that c reports a need for n registers, that m local temporaries are live across the reference to c, and that at most k variables are ever simultaneously live in the rest of the backquote expression (the compiler can discover k statically, since it is information local to one backquote expression). The CGF for this backquote expression might tell its caller that it needs $\max(k, m+n)$ registers; or that it would like that many but really needs only $m+n$ (for the loop); or—in the limit, if it is willing to save and restore—that it needs no registers at all.

At the end of the first pass, each code-generating function has information about the register requirements of every CGF that it calls. In the second pass, the VCODE macros emit code as normal. However, before every call to a CGF, the current CGF must compare the number of free registers (available, for instance, in the VCODE bit vectors) against those requested by the called CGF in the first pass, and generate any necessary spill code. After the call, the calling CGF should emit symmetric code that reloads the values into registers. The more detailed the register usage information returned by the called CGF in the first pass, the more intelligent the spilling decisions that can be made by the calling CGF. Figure 5-11 illustrates the operation of this algorithm in the case of a small set of nested cspecs.

In summary, this scheme requires a few simple changes to code-generating functions:

- VCODE macros that emit code are guarded by checks to prevent code generation in the first pass. To decrease overhead, adjacent macros can all be guarded by one check.
- The epilogue of each CGF contains code that, in the first pass, returns the number of needed registers, computed as a function of local temporaries and the needs of nested CGFs.
- Every call to a CGF is followed by code that, in the first pass, stores the number of required registers returned by that CGF.
- Where necessary—for instance, in code that generates a loop—a call to a CGF is preceded by code that, in the second pass, generates whatever spills are necessary to reconcile the number of available registers with the number of registers requested by the nested CGF. The call is also followed by code that generates the corresponding restores.

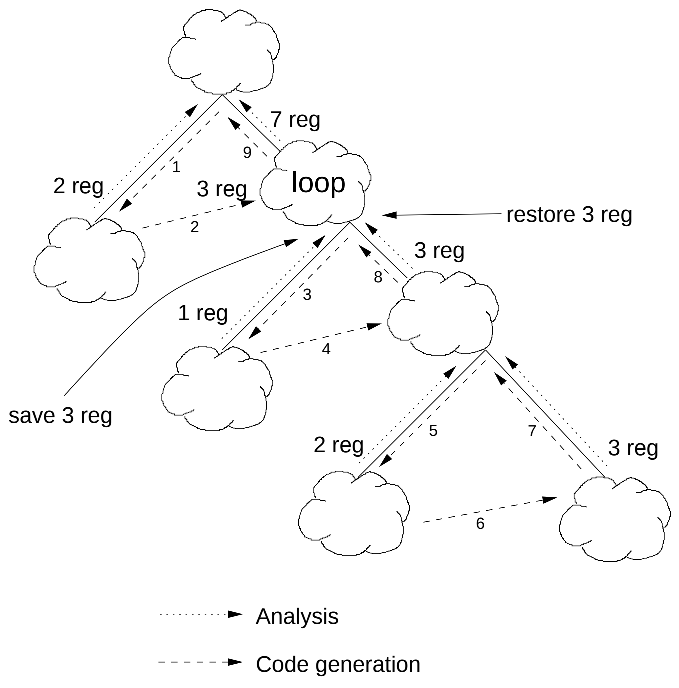

Figure 5-11: The two-pass register allocation process. Each cloud denotes a cspec; solid lines denote cspec composition. During the first pass (dotted lines), CGFs of nested cspecs communicate their register requirements to their parents in the composition tree. During the second pass (dashed lines), CGFs generate normal code and spill code as appropriate. For instance, if composition occurs in a loop, a CGF may generate code to save (and later restore) as many registers as are needed by the nested cspecs.

The advantage of this algorithm over the simple one from Section 5.1 is that allocation decisions are no longer purely local and greedy. The prepass that collects register requirements allows the subsequent code-generating pass to move spills out of frequently used code like inner loops, and to ensure that heavily used variables are always assigned to registers. In fact, the two-pass scheme can never perform more poorly than the greedy one-pass algorithm. If each CGF reports to its parent that it requires no registers, then the code generation and spilling phase obtains no useful information, and the two-pass algorithm simply reduces to the greedy one.

The dynamic compile-time cost of these benefits should be low; since the first pass is lightweight, the whole algorithm should add no more than 30-40% to standard code generation with VCODE. In fact, the algorithm could be made almost no slower than the current VCODE register allocation by folding the first pass into the environment binding phase. Recall that the first pass simply passes register use information up from nested cspecs to enclosing cspecs. Composition of cspecs at environment binding time, described in Chapter 4, also works in this bottom up manner: cspecs are composed together into bigger, higher-level cspecs, and once a cspec exists, nothing more can be added to the tree of cspecs that it encloses. As a result, the register requirements of each cspec can be stored in the corresponding closure and propagated to the closures of enclosing cspecs during cspec composition, all at environment binding time. This solution eliminates a recursive traversal of all CGFs, and should make the overhead of the two-pass algorithm competitive with that of VCODE.

The two-pass algorithm's expected performance makes it an attractive alternative to linear scan for solving the register allocation problem posed by cspec composition. Linear scan may generate slightly better code, because it has finer-grain information about the length and weight of live intervals, but the difference should be small and the dynamic compile-time savings substantial.

Furthermore, the two-pass approach could be extended—at a slight increase in compile-time—to implement linear scan without the overhead of the ICODE intermediate representation. The idea is to change the first pass so that it tracks not just overall register requirements, but the live interval of each local or temporary. To do this, the code would increase a position counter at each VCODE macro while traversing the CGF call tree, and extend the live interval of a variable according to the position of each macro that used or defined the variable. The result would be a set of live intervals that could be allocated using linear scan prior to emitting code in a second traversal of the CGFs. If—as is currently the case—the ICODE run-time system is used primarily to enable register allocation, its full-blown IR is probably not necessary. The two-pass process just described could be used to eliminate most of the overhead of ICODE's intermediate form without affecting register allocation quality.

Finally, it may be possible to extend this approach beyond register allocation, to other kinds of analyses and optimizations. For instance, tcc could statically generate a summary of data flow information for each `cspec`; during dynamic compilation, this information could be used to quickly perform coarse-grain data flow analyses and optimizations. This scheme is similar to region- and structure-based analyses [1], in which analysis performance is improved by summarizing the data flow effects of hierarchical program regions. While simple in principle, such inter-`cspec` analysis is tricky. For instance, any set-based analysis could involve an arbitrary, run-time determined number of elements. Dynamically growing the associated data structures, such as bit vectors, and keeping track of the individual set elements in the presence of composition would be complicated and require considerable overhead. It is not clear that the benefit of such optimizations would be worth the associated overhead. For now, though, using the two-pass algorithm to improve register allocation with minimal run-time overhead is certainly a promising approach.

Chapter 6

One-Pass Data Flow Analyses

The success of the linear scan register allocation described in Chapter 5 suggested that a similar approach might be adaptable to other optimizations, including data flow analyses and optimizations. The linear scan algorithm achieves good performance by coarsening liveness information into conservative live intervals. Such coarsening of data is unlikely to work for data flow analysis, because data flow information is inherently more fine-grained and complex than liveness information. If we were to coarsen data flow information in a similar manner (for example, creating *gen* and *kill* sets for entire strongly connected components of the flow graph), the resulting data flow sets would probably be mostly empty due to conservative assumptions during coarsening. This claim is not backed by measurements—and such a coarsening may be interesting future research—but it is a reasonable assumption.

As a result, the focus for data flow analysis shifted from coarsening information to coarsening action—reducing the number of nodes that are visited during an analysis. One-pass data flow analyses are based on a "90/10" assumption that it may frequently be possible to obtain most of the benefit of data flow analysis while doing only a fraction of the work. The idea is to touch each node in the flow graph exactly once and to make conservative assumptions about data flow information so as to decrease the analysis time at only a slight cost in accuracy and optimization effectiveness.

Other work on fast data flow analysis has focused on structure-based analyses, such as those based on the $T_1 - T_2$ transformation and hierarchical loop decomposition [1]. The goal of these analyses, however, is to improve the efficiency with which data flow information is propagated without affecting the accuracy of the analysis. Extracting the necessary structural information from the control flow graph can be a relatively complicated and time-consuming process. The end of the previous chapter proposed a similar structural approach, in which data flow information is summarized statically for each cspec and then combined at dynamic compile time, but this approach is also complicated and likely to have considerable overhead. The algorithms in this chapter attempt to avoid these problems by compromising on the quality of the analysis.

Although these analyses are faster than traditional ones by a significant constant factor, the overall overhead required to set up the analysis and perform the optimization is probably too high for most dynamic compilation applications. Nonetheless, the results are interesting and may be useful in long-running dynamic code applications.

6.1 The Algorithms

This section describes the overall structure of these one-pass algorithms. We begin with a traditional relaxation algorithm, and move on to more efficient but less accurate versions that touch each node exactly once. We focus only on forward data flow problems, but the discussion can easily be extended to backward problems by reversing control flow edges.

For each algorithm, we assume that we have a flow graph $G = (V, E)$ with a distinct entry node $n_0 \in V$. Nodes may correspond to basic blocks or to individual pseudo-instructions in the compiler IR. Each node n is associated with input and output data flow sets and a (forward) data flow function

f_n such that $out(n) = f_n(in(n))$. These parameters, as well as the data flow meet operator, may be changed to implement different analyses. For example, in the case of the null pointer check removal evaluated in Section 6.2, the data flow information at a given point is the set of pointers for which pointer checks are available along all paths to that point. As a result, the input and output sets are initialized to the empty set, the meet operator is set intersection, and f_n removes any pointers that are modified in n and not checked after the modification, and adds any pointers that are checked in n and not modified after the check.

The following "traditional" algorithm, TRAD, repeatedly traverses the entire flow graph in breadth-first order until the data flow information reaches a fixed point. The traversal order is breadth-first only for the purpose of comparison with later algorithms; it could easily be changed to another order, such as depth-first or layout order.

```
TRAD($G$)
    foreach $n \in G$ do
        $out(n) \leftarrow in(n) \leftarrow \{\}$
    $change \leftarrow$ true
    while ($change$) do
        $visited \leftarrow \{\}$
        $worklist \leftarrow \{n_0\}$
        $change \leftarrow$ false
        while ($worklist \neq \{\}$) do
            $n \leftarrow$ pop($worklist$)
            $change \leftarrow change \vee$ TRAD_NODE(n)
            foreach $s \in succ(n) - visited$ do
                append $s$ to $worklist$
```

```
TRAD_NODE(n)
    $orig \leftarrow out(n)$
    $in(n) \leftarrow \bigcap_{p \in preds(n) \cap visited} out(p)$
    $out(n) \leftarrow f_n(in(n))$
    $visited \leftarrow visited \cup \{n\}$
    return $orig = out(n)$
```

It is easy to do less work than this algorithm, without sacrificing analysis accuracy, by keeping track of nodes that need to be reexamined due to changes in the data flow information of their predecessors. The algorithm below analyzes all the nodes once, and thereafter uses a worklist to only ever touch those nodes that really need to be updated.

```
OPTRAD($G$)
    foreach $n \in G$ do
        $out(n) \leftarrow in(n) \leftarrow \{\}$
    $worklist \leftarrow \{\}$
    foreach $n \in G$ in breadth-first order do
        append $n$ to $worklist$
    while ($worklist \neq \{\}$) do
        $n \leftarrow$ pop($worklist$)
        if OPTRAD_NODE(n) then
            foreach $s \in succ(n)$ do
                append $s$ to $worklist$
```

```
OPTRAD_NODE(n)
    $orig \leftarrow out(n)$
    $in(n) \leftarrow \bigcap_{p \in preds(n) \cap visited} out(p)$
    $out(n) \leftarrow f_n(in(n))$
    $visited \leftarrow visited \cup \{n\}$
    return $orig = out(n)$
```

The OPTRAD algorithm above is an efficient implementation of full data flow analysis. However, it may still touch many nodes several times due to changes in data flow information carried by back edges. The fast approximate algorithms below ignore information from back edges, and traverse the flow graph just once, making conservative assumptions about loops.

The BFS algorithm below is the most radical example of this idea. It touches each node in the flow graph exactly once; if it ever examines a node before all of its predecessors, it makes conservative, worst-case assumptions about the unexamined predecessors. The order of traversal is breadth-first because it provides good chances of visiting a node after all of its predecessors. An interesting variation might be the simple linear layout order, which would enable traversal of the IR without building a control flow graph.

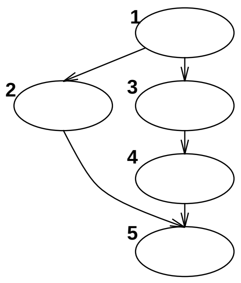

Figure 6-1: A small control flow graph. Nodes are labeled with their depth-first numbers.

```
BFS($G$)
    foreach $n \in G$ do
        $out(n) \leftarrow in(n) \leftarrow \{\}$
        $visited \leftarrow \{\}$
    $worklist \leftarrow \{n_0\}$
    while ($worklist \neq \{\}$) do
        $n \leftarrow$ pop($worklist$)
        BFS_NODE(n)
        foreach $s \in succ(n) - visited$ do
            append $s$ to $worklist$
```

```
BFS_NODE(n)
    $visited \leftarrow visited \cup \{n\}$
    $in(n) \leftarrow \bigcap_{p \in preds(n)} out(p)$
    $out(n) \leftarrow f_n(in(n))$
```

In its emphasis on touching each node exactly once, BFS makes some extremely conservative assumptions. For example, consider the control flow graph in Figure 6-1. One possible order in which BFS touches the nodes is $[1, 2, 3, 5, 4]$. When it reaches node 5, the algorithm has not yet visited node 4, which is a predecessor of 5, so it must make the most conservative possible assumption about the data flow information flowing into 5.

We can refine BFS and potentially make it more effective by traversing the flow graph in topological sort order. Then, at least for acyclic code, no basic block is visited until all of its predecessors are visited. In the case of loops, we simply make conservative assumptions about the information available at loop headers, and can therefore visit loop headers before their predecessors. Unfortunately, this means we must first perform a depth-first search to identify loops—in other words, each node is touched twice, rather than once as in the BFS algorithm. However, this is still less than full relaxation algorithms such as TRAD, and it may provide better results than BFS.

The pseudocode for this algorithm based on topological order appears below:

```
TOP($G$)
    foreach $n \in G$ do
        $out(n) \leftarrow in(n) \leftarrow \{\}$
        $npreds(n) \leftarrow$ number of predecessors of $n$
    Perform DFS(G) to identify loop headers
    $worklist \leftarrow \{n_0\}$
    while ($worklist \neq \{\}$) do
        $n \leftarrow$ pop($worklist$)
        TOP_NODE(n)
        foreach $s \in succ(n) - visited$ do
            $npreds(s) \leftarrow npreds(s) - 1$
            if $npreds(s) = 0$ or $s$ is loop header then
                append $s$ to $worklist$
```

```
TOP_NODE(n)
    $visited \leftarrow visited \cup \{n\}$
    if $n$ is loop header then
        $in(n) \leftarrow \{\}$
    else
        $in(n) \leftarrow \bigcap_{p \in preds(n)} out(p)$
    $out(n) \leftarrow f_n(in(n))$
```

6.2 Evaluation

One-pass data flow analyses have not been implemented in a dynamic compiler. The overhead required to create the necessary data structures and to perform optimizations would most probably reduce the speed of a system like ICODE by a factor of two or more. The relatively low performance benefits of any one data flow optimization (common subexpression elimination, copy propagation, etc.) make such increases in compilation overhead potentially unattractive. Despite these problems, one-pass analyses are interesting curiosities. Initial results indicate that they perform surprisingly well.

Each of the data flow analysis methods described above was implemented as a SUIF [2] pass and served as a base for three optimizations:

- **Redundant null pointer check removal for Java.** Java requires that all object dereferences be checked, and that the run-time system throw an exception in case of null pointers. Often, null pointer checks are redundant because checks on a given pointer are available on all paths to a particular use of that pointer. The check removal optimization basically does an "available check" analysis, and deletes redundant checks.

- **Copy propagation.** Again, this is an availability analysis, but not limited to the Java pointer check case: a copy may be propagated to a use if it is available on all paths to that use.

- **Common subexpression elimination.** Another availability analysis: an expression is replaced by a reference to a temporary if it is available on all paths to its use.

For the Java benchmarks, the Toba 1.0 compiler from the University of Arizona [60] translated the Java to C code that was compiled by SUIF. After analysis, the SUIF representation was converted to C and compiled by gcc. Unfortunately, this infrastructure made it difficult to obtain run-time measurements. As a result, this chapter reports only compile time and static measurements of the effectiveness of the optimizations. Dynamic measurements of the effectiveness of common subexpression elimination on a couple of SPEC 95 benchmarks (li, compress) and some simple Unix utilities (sort, wc) suggest that—at least for those benchmarks—static measurements are quite similar to dynamic ones. (Dynamic measurements had to be obtained by inserting counters into the code, confirming the relative ineffectiveness of single data flow optimizations like CSE on most code.)

The Java benchmarks used for null pointer check removal include an animation library, a simple drawing program, a set of matrix routines for 3D graphics, a spreadsheet, two compression programs (ICE and DES), a regular expression package, a random number generator, and an audio file format converter (.wav to .au). Copy propagation was evaluated on all these benchmarks except for DES, and also on files from two SPEC 95 benchmarks, lisp (130.li) and perl (134.perl).

For each benchmark and type of analysis, we report two metrics: compile-time overhead and effectiveness of optimizations. Due to the very high overhead of SUIF, compile-time cost was measured by counting the number of times that basic blocks had to be "touched" to recompute data flow information. However, for some of the bigger benchmarks (such as DES), the approximate algorithms were visibly faster, in one case cutting runtime of the SUIF pass down from about 30 minutes to about 5 minutes. The effectiveness of the analyses was measured statically. In the case of null pointer check removal, it is the static count of deleted null pointer checks; in the case of copy propagation, it is the static count of copies found available.

Figures 6-2 and 6-3 show the overhead and benefit, respectively, of null pointer check removal. Figures 6-4 and 6-5 show the same information for the copy propagation analysis. These results can be summarized in three points:

- The approximate algorithms do a lot less work (frequently five times less) than the traditional ones. In particular, for the null pointer check analysis, they touch fewer basic blocks than there are in the program, because they delete a check (and its associated "error" basic block) as part of the analysis, as soon as they detect that it is redundant.

- For most benchmarks, the approximate algorithms provide benefits comparable to those of the traditional ones. In the case of null pointer check removal, they are often within 90% of the effectiveness of the traditional algorithms.

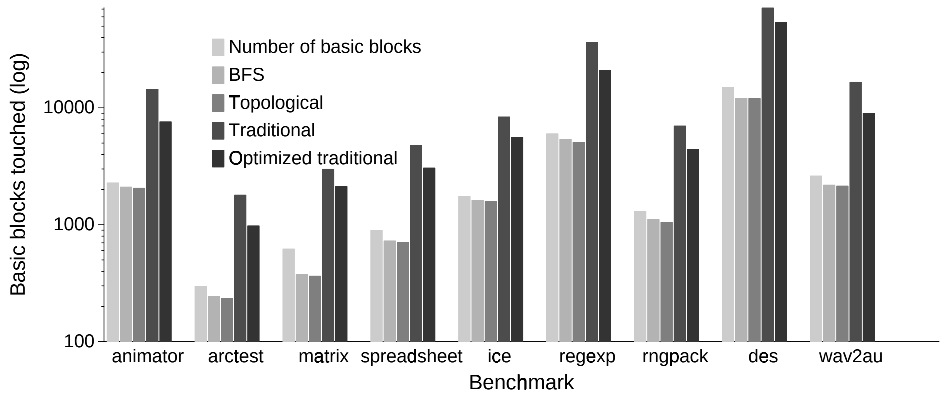

Figure 6-2: Work done (basic blocks touched) during null pointer check removal.

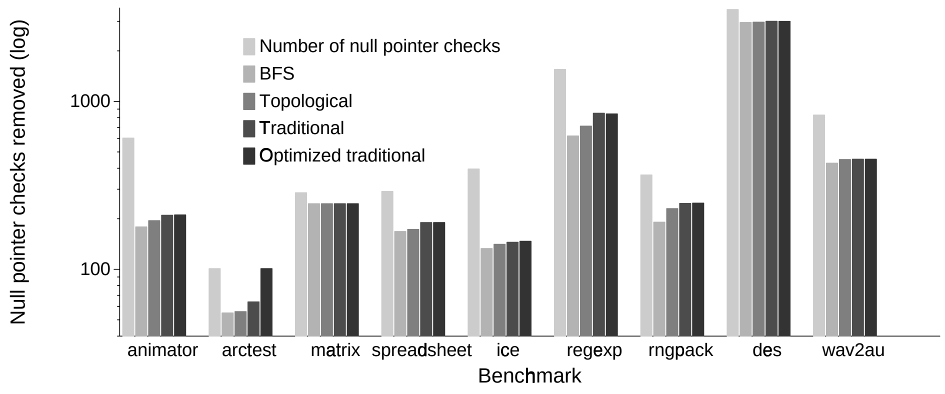

Figure 6-3: Results from null pointer check removal: static count of null pointer checks removed.

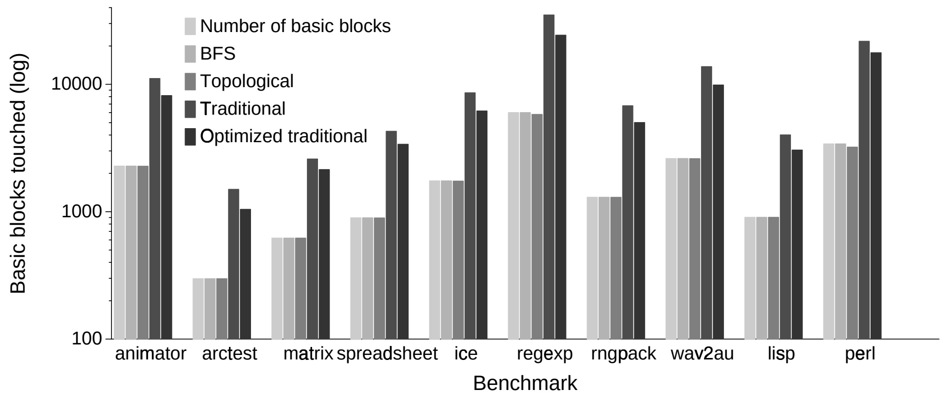

Figure 6-4: Work done (basic blocks touched) during availability analysis.

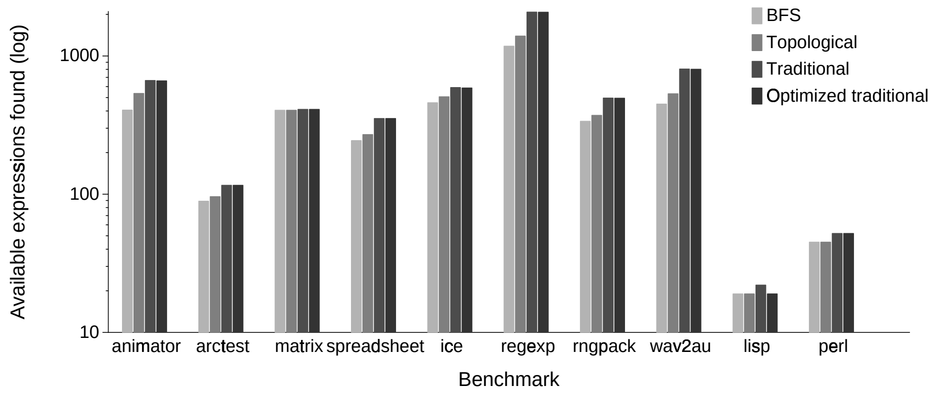

Figure 6-5: Results from availability analysis: static count of expressions found available.

- The effectiveness of the approximate algorithms relative to the traditional algorithms is slightly lower for copy propagation than for null pointer check removal. This suggests that the success in null pointer check removal may be due in part to the particular stylized way in which Toba translates Java and the associated pointer checks to C.

These results are far from complete—the lack of run-time performance data is the biggest problem. However, they already show that approximate analyses can be almost as effective as traditional ones, at one third to one fifth of the compile-time overhead.

Future work should measure run-time performance more extensively, use more benchmarks, and extend the one-pass approach to other types of data flow analysis. Such a study would provide a more accurate picture of the effectiveness of one-pass analysis. The algorithms also need to be compared to local analysis: for some optimizations, local analysis might do almost as well as this approximate global analysis. In addition, it will be interesting to explore how the order in which the control flow graph is traversed affects the effectiveness of approximate algorithms, and whether coarsening data flow information—for instance, at the level of strongly connected components, as in linear scan allocation—provides any benefits.

6.3 Summary

This chapter has discussed one-pass data flow analyses. These analyses traverse the flow graph in one sweep and make conservative assumptions about unavailable data flow information. They are surprisingly effective at improving analysis performance without excessively degrading accuracy.

However, despite their effectiveness, one-pass analyses are probably still unsuitable for lightweight dynamic compilation like that in ‘C. Data flow analysis is only one part of the optimization process: other parts, such as setting up the data structures necessary for the analysis and actually performing the optimization, are also time-consuming. Furthermore, even if the cost of data flow analysis is reduced by a factor of three to five, it remains relatively high due to bit vector operations and the manipulation of the data structures that store the bit indices of variables and expressions. As mentioned earlier, the improvement in code quality is probably not worth the increase in dynamic compile time.

Nonetheless, one-pass analyses could be useful for long-running dynamic code applications in which the performance of generated code is crucial. For instance, a Java just-in-time compiler for long-running server-like applets (“servlets”) might benefit from this sort of optimization. One-pass analyses could also serve to improve the performance of static optimizing compilers without greatly decreasing the resulting code quality. Certainly, static compilers that perform many optimizations may be better served by an incremental representation such as static single assignment form [16], rather than by fast non-incremental analyses. However, building the SSA representation requires considerable overhead, so in some situations one-pass analyses may be preferable.

Chapter 7

Fast Peephole Optimizations

The one-pass analyses presented in Chapter 6 are effective, but still incur considerable overhead. A technique that can substantially improve dynamic code with lower run-time overhead is peephole optimization, which tries to replace short instruction sequences within a program with equivalent but smaller or faster instruction sequences. The term "peephole" [52] refers to the local nature of this optimization: the compiler only looks at a small sliding window (or peephole) of code at one time.

Peephole optimizations are simple and useful. They are usually performed late in the compilation process, and can improve code quality without greatly increasing compilation cost or compiler complexity. This makes them attractive for optimizing hastily generated dynamic code, such as that produced with 'C or by a just-in-time compiler.

Past work on peephole optimizations, however, has been primarily in the context of assembly code or compiler intermediate representations. This chapter describes the issues involved in direct peephole optimization of binary code and proposes a solution to the problem. It presents bpo ("binary peephole optimizer"), a generic binary peephole optimization tool that accepts a set of rewrite rules and a buffer of binary data, and produces a new buffer in which the data has been transformed according to the rewrite rules. bpo is simple, portable, and fast, and has been used successfully to improve the quality of dynamic code written in 'C.

7.1 Related Work

Peephole optimizations are in widespread use today. The GNU C compiler [69], for example, performs peephole optimization on its RTL intermediate representation. Apparently, however, no system other than bpo performs direct optimization on binary code.

Tanenbaum et. al. [71] successfully used table-driven peephole optimization to improve intermediate code of a portable compiler. Davidson and Fraser described PO, a peephole optimizer for a compiler intermediate representation (register transfer language) that generated good code and allowed the rest of the compiler to be made simpler and easier to retarget [17, 18]. They also extended the system to automatically infer new peephole optimization rules by tracking the behavior of an optimizer on a code testbed [19]. Similarly, Fraser and Wendt tightly integrated a peephole optimizer with a code generator to speed up code generation and improve code quality [33, 34, 76]. They also explored the creation of code generators and peephole optimizers from compact specifications of the intermediate code and target machine. Fraser also implemented copt [29], a very simple rule-based peephole optimizer for rewriting ASCII text. Text rewriting is useful for optimizing assembly code, and as described in Chapter 4, copt is employed in tcc.

Executable editing tools, such as Morph [78], Etch [63], EEL [49], and Atom [68], rewrite binary code, but they are not quite comparable to bpo. They provide generic frameworks for binary editing, rather than addressing the specific problem of peephole optimization. They also do not have the performance constraints imposed by dynamic code generation. However, bpo can be—and has been—used effectively as part of an executable editing tool.

rules	::=	*rule rules* | *empty*
rule	::=	*input output*
input	::=	*data* =
output	::=	*data* +
data	::=	{ *const* | *variable* | *newline* | *comment* }*
const	::=	{ 1 | 0 }+
variable	::=	{ [a-z] | [A-Z] }-*
comment	::=	# ⟨ any character other than *newline* ⟩ *newline*

Figure 7-1: Grammar for binary peephole optimization rewrite rules.

Generating code that efficiently decides whether one of a set of binary rewrite rules is applicable to some data is similar to generating good code for a C switch statement. bpo's efficient pattern matching is based on multiway radix search trees (MRSTs) [26], which have been used to implement fast switch statements.

7.2 Design

A binary peephole optimizer has somewhat different requirements than a peephole optimizer for textual assembly code or compiler intermediate representation. The most obvious difference is that a binary optimizer works on arbitrary bit patterns, below the level of a byte or machine word, whereas the other optimizers manipulate pointers or strings of characters. The second difference is that a binary rewrite rule has a natural notion of length, as well as position, whereas text or IR rules generally only have a notion of position. In other words, one would like to write, "match bits 4–7 of word 2," whereas in a traditional rule one would write, "match operand 2," or "match the string between , and [."

Because it can be used at run time, a binary peephole optimizer must have excellent performance. Several other important details must also be addressed when optimizing binary code: they are described in Section 7.2.3.

7.2.1 Rewrite Rules

bpo rules are written in a simple language for describing bit patterns whose BNF grammar appears in Figure 7-1. Each rule consists of an input pattern that is matched against binary input data, and an output pattern that determines how the input is rewritten if the input pattern matches. The language is centered around bit addressing: each character in a pattern corresponds to one bit of binary data. As a result, there are only two constants, 0 and 1. Variables (wildcards) are denoted by a letter of the alphabet, optionally followed by a sequence of dashes which denote the number of bits assigned to that variable. For example, a— matches a sequence of four bits, bound to the variable a. In each rule, the input pattern is separated from the output pattern by the = character, and rules are separated from each other by the + character.

For instance, the rule a-1001a-=a-a-1111+ matches any byte in which the two most significant bits are the same as the two least significant bits, and the middle four bits have value 1001. It replaces any matching byte with a new byte in which the four most significant bits are copies of the original byte's two most significant bits, and the bottom bits are 1111. Similarly, the rule 11110000=+ removes any bytes that have value 0xf0.

Input and output patterns need not have the same length, but their length must be a multiple of the size (in bits) of a machine data type. This unit, which we call the *rewrite element*, is the level of granularity at which matches occur; it is usually a byte or word.

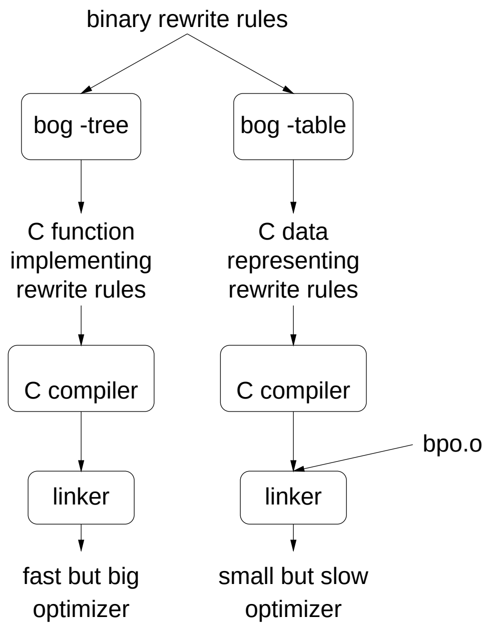

Figure 7-2: Generating a binary peephole optimizer in two different ways.

7.2.2 Optimizer Architecture

The need to have both excellent performance and support for easily changeable rewrite rules led to a two-part design similar to that of yacc and other compiler compilers. Figure 7-2 illustrates this structure. The use of an optimizer generator does not affect the usability of the system, since rule sets change infrequently and recompiling them takes a negligible amount of time.

A "binary optimizer generator," bog, accepts a set of binary rewrite rules written in the language described above. bog can then generate either C initializers for data structures that encode the rules (basically a big rule table), or a C function that directly implements the rules. The C initializers must be linked with a driver program, labeled bpo.o in the figure, which is responsible for applying the rules encoded in the C data structures to a provided buffer. The C function created by bog, on the other hand, provides the same interface as the driver program but implements the rules directly, using a variant of multiway radix search. These two alternatives provide a tradeoff: the data structures and bpo driver implement a slow but compact optimizer; the hard-coded algorithm takes up more space (the code can be tens of times bigger, though this is offset in part by the lack of rule tables) but is much faster (approximately an order of magnitude).

7.2.3 The bpo Interface

The functioning of bpo is outlined in Figure 7-3. The most important parts of the bpo interface are two buffers, *input* and *output*. Initially, *input* contains all the data to be rewritten, and *output* is empty. When bpo terminates, *output* contains the result of applying the rewrite rules to *input*, and *input* contains garbage. When a rule matches, its output is added to *output* one rewrite element at a time, and as each element is added all rules are tried against the end of *output*.

For example, consider Figure 7-4, which illustrates a simple rewriting. At each step, one rewrite element is copied from the head of the input buffer to the tail of the output buffer. bpo tries to apply all its rules to the tail of the output buffer. If a match occurs, the input of the rewrite rule is removed (with a simple pointer change) from the output buffer, and the output of the rewrite rule is prepended to the head of the input buffer. It is then moved onto the output buffer one rewrite element at a time, just like any other data in the input buffer. This two-buffer scheme allows the

```
while input is not empty do
    remove rewrite unit at head of input, and append it to tail of output
    foreach r in rules do
        try to match input pattern of r on tail of output
        if r matches then
            remove matched rewrite units from tail of output
            push output pattern of r (with values assigned to variables) onto head of input
            break
```

Figure 7-3: An outline of the bpo rewriting process.

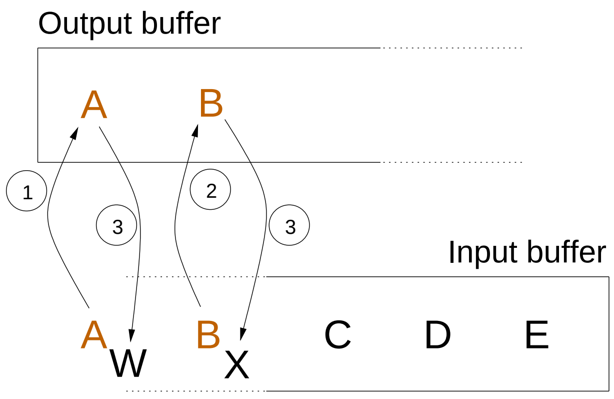

Figure 7-4: Example of bpo operation. The binary data ABCDE (where each letter is a rewrite element) is being rewritten according to the rule AB=WX+. First, A is copied from the input buffer to the output buffer; bpo finds no matches. Then B is copied, the rule matches, and its output is added to the head of the input buffer.

optimizer to transparently consider fragments of previous rewrites, possibly preceded or followed by additional inputs, as the input to new rewrite rules. It also ensures that data does not have to be copied and "pushed down" in memory if the output of a rule is longer than its input.

On architectures with variable instruction width, notably the Intel x86, some suffixes of instructions may themselves be legal instructions. As a result, a rewrite rule may fire when it actually should not, because it matched an instruction suffix rather than the instruction it was really intended to match. To solve this problem, bpo can be parameterized with a function—such as a disassembler—that identifies the boundaries of instructions in the input buffer. bpo stores the instruction lengths provided by this function and uses them to ensure that matches in the output buffer happen only on instruction boundaries.

Another difficulty is that some information in the input buffer may not be rewritable. For instance, bpo should not mangle jump tables, even if some data in them may look like instructions and match some peephole optimization rules. For this purpose, bpo accepts a vector that maps onto the input buffer and indicates which sections of the input should not be touched.

Lastly, it may be necessary to track the permutations of rewrite elements performed by bpo. For example, one may need to adjust the destination of a jump after bpo has changed the order of some instructions. bpo can optionally create a buffer that maps onto the output buffer and denotes the position that each rewrite element in the output buffer had in the input buffer. It only tracks permutations of variables spanning entire rewrite elements—bit permutations within a rewrite element and new constant values cannot be tracked easily, and they do not need to be.

7.3 Efficient Pattern-Matching Code

As described above, bog can generate code which directly implements the specified rewrite rules, which avoids the data structure interpretation performed by the bpo driver. Generating code to match a set of rules is somewhat similar to generating code for a C switch or Pascal and Ada case statement. In fact, if all rules have only constant values (i.e., there are no variables or wildcards) and all rules have input patterns of the same length, then each input pattern is analogous to a case label in a switch statement, and the process of matching an input is analogous to switching on that input. In this simple case, a technique for generating efficient switch statements, such as multiway radix search trees, would suffice for peephole optimization also. However, bpo rules are usually not so simple: input patterns may each have arbitrary lengths, and they may contain arbitrarily many variables. This section presents a generalization of multiway radix search that produces efficient pattern-matching code in bpo.

The original paper on MRSTs [26] defines the important concepts of a window and a critical window. We repeat those definitions here. Let Z be the set of all k-bit values, for some positive integer k. Denote the bits of k-bit values by $b_{k-1}, \ldots, b_0$. A *window* W is a sequence of consecutive bits $b_l, \ldots, b_r$, where $k > l \geq r \geq 0$. Let $\text{val}(s, W)$ be the value of the bits of s visible in the window W. For example, if $W = b_5, b_4, b_3$, $\text{val}(101001_2, W) = 101_2 = 5$. A window $W = b_l, \ldots, b_r$ is *critical* on $S \subset Z$ if the cardinality of the set $V_W = \{\text{val}(s, W) \mid s \in S\}$ is greater than 2^{l-r}. A window W is *most critical* if the cardinality of $|V_W|$ is greater than or equal to the cardinality of $V_{W'}$ for all equally long windows W'.

In other words, since W can represent 2^{l-r+1} values, W is critical on S if the number of values in S that differ in W is greater than half the number of values that W can represent. The MRST algorithm finds the most critical window of a set of case labels so as to (1) identify the sequence of consecutive bits that maximally differentiates the input with respect to the case labels and (2) create jump tables [1] that are more than half full. Once it has partitioned the possible input based on a critical window, MRST applies itself recursively to each partition and creates a tree of jump tables, each indexed by the most critical window on the set of values that could reach it. This scheme produces a switch statement that performs the minimal number of tests while maintaining jump tables at least half full. (Of course, one could trivially perform just one test, looking at all bits of the input and using them to index into just one huge jump table. However, this method would waste huge amounts of memory and decrease performance due to paging.) The details of how to find critical windows, as well as the MRST algorithm for generating switch statements, are described by Erlingsson et al. [26].

The first issue that must be addressed to extend the MRST algorithm to binary rewriting is how to match input patterns that differ in length. Consider, for example, one rule, r_1, with input pattern abc, and another rule, r_2, with input pattern bc. When using the table-driven approach, the bpo driver applies all rules in the order in which they appear in the rule list. Thus, if r_2 appeared before r_1, r_1 would never be applied because r_2 would always match first. As a result, a programmer would never write a rule r_1 after a rule r_2 whose input pattern was a suffix of r_1's input pattern.

bog uses this fact to generate a tree-based pattern matcher. The generated matcher first tries to identify an input by matching it against the common suffix of all input patterns. If this suffix is not enough to identify the matching pattern (if any), the matcher considers bigger suffixes (i.e., moves towards the front of input patterns). If one pattern is the suffix of another, the algorithm attempts to match against the longer pattern before the shorter one, in keeping with the semantics described above.

The core of the algorithm is two mutually recursive procedures, META-MRST and MRST-AT-POS, shown in Figure 7-5. META-MRST searches for a rewrite element in a position that is common to all input patterns in *rules* and that has a critical window more critical than that of any other such rewrite element that has not been processed yet. It then invokes MRST-AT-POS to perform a variant of the MRST algorithm at that position.

Let us look in more detail at the pseudocode in Figure 7-5. Initially, META-MRST is called with all known rewrite rules. The rules may look like this example (where each letter is a rewrite element):

```
MRST-AT-POS(rules, pos, mil, status)
    if rules = {} then return
    mask ← MASK-AT-POSITION(rules, pos)
    w ← CRITICAL-WINDOW(rules, pos, mask)
    if (w = 0) then
        if done ∈ status[j] for all j in 1..mil then
            longrules ← {r ∈ rules | input-length[r] > pos}
            shortrules ← rules \ longrules
            META-MRST(longrules, status)
            EMIT-CHECKS(shortrules)
        else
            META-MRST(rules, status)
        return
    GENERATE-SWITCH(pos, w)
    foreach i in 0...$2^{\text{leftbit(w)}-\text{rightbit(w)}+1}$ do
        GENERATE-CASE(i)
        r2 ← {r ∈ rules | val(r, pos, w) = i}
        MRST-AT-POS(r2, pos, mil, copy(status))
        GENERATE-BREAK()

META-MRST(rules, status)
    if rules = {} then return
    mil ← MAX-COMMON-INPUT-LEN(rules)
    pos ← CRITICAL-WINDOW-POSITION(rules, mil, status)
    status[pos] ← status[pos] ∪ {done}
    MRST-AT-POS(rules, pos, mil, status)
```

Glossary

- *pos*: the position of a given rewrite element within an input pattern, relative to the end of the input pattern. For example, in the input part of the rule abcd=b+, where each letter is a rewrite element, d has position 1, c has position 2, b has position 3, and a has position 4.
- *status*: for each position, a set of flags pertaining to it. $done \in status[pos]$ indicates that position *pos* has been visited by META-MRST.
- MAX-COMMON-INPUT-LEN(*rules*): returns the length of the shortest input pattern in *rules*.
- CRITICAL-WINDOW-POSITION(*rules*, *mil*, *status*): returns the position p of the rewrite element with the most critical window among all those with position $p' \leq mil$ and such that $status[p'] \not\ni done$.
- MASK-AT-POSITION(*rules*, *pos*): returns a bit mask identifying those bits that can be used to differentiate rules at the given position. Some bits may not be usable because they correspond to wildcards.
- CRITICAL-WINDOW(*rules*, *pos*, *mask*): this is the critical window algorithm from Erlingsson et al. [26], adapted to ignore bits not specified in *mask*. This extension is required to handle wildcards correctly.
- EMIT-CHECKS(r): generate if statements to sequentially match all the rules in r.
- GENERATE-SWITCH(pos, w): generate a switch on window w of position *pos*.
- GENERATE-CASE(i): generate case label i.
- GENERATE-BREAK: generate a break statement.

Figure 7-5: Generating efficient binary pattern matching code with multiway radix search trees.

```
ABC = Q+        // rewrites ABC to Q
  C = D+        // rewrites C to D; fires if input ends in C but not ABC
 XY = WZ+       // rewrites XY to WZ
```

For clarity, we have formatted the rules so that input rewrite elements are lined up vertically according to their distance from the end of the input pattern: the last rewrite elements of each pattern (C, C, and Y) are in one column, and so forth. The distance from the end of an input pattern is the *position*. Thus, in the first rule, C has position one and B has position two. Initially, the *status* flag indicates that no position has been processed.

In accordance with our earlier insight about the "precedence" of small rules over large ones, META-MRST must find the rules' greatest common input length, or simply the shortest input length of any rule. In the case of the example, that length is one, since the second rule is short. It then iterates over all these common positions (only one in our case), and uses the critical window calculation algorithm from Erlingsson et al. [26] at each position to find the position at which the most critical window is more critical than that at any other position. Having thus picked a fixed-length rewrite element, we mark its position as "done" and call MRST-AT-POS to apply an extension of the original MRST algorithm at that input position.

MRST-AT-POS highlights the second difference between generation of switch statements and binary pattern matching: not only do inputs vary in length, but some bits—those corresponding to wildcards or variables—cannot be used to differentiate among rules. For instance, in the byte-rewriting rules below

```
a−−−1010=10101010+
a−−−0101=01010101+
```

the first four bits cannot be used to decide whether one rule or the other matches the input. Therefore, MRST-AT-POS first computes a mask of significant bits at the given input position, and then computes the critical window based on that mask.

If the critical window exists, then MRST-AT-POS behaves like the original MRST algorithm. Beginning with the call to GENERATE-SWITCH, it generates a switch statement in which each case handles a possible value of the critical window. For each case, it calls itself recursively, creating a tree of switch statements as necessary to differentiate all rules.

However, a critical window may not exist. For instance, all remaining bits may correspond to wildcards or variables. In that case, there are two alternatives. If any of the positions between position one and the maximum common input length have not been processed yet, MRST-AT-POS calls META-MRST, which will pick the position with the most critical window.

If, on the other hand, no positions up to the maximum common input length remain to be processed, then the algorithm has "bottomed out." It divides the rules into long rules—those longer than the maximum common input length—and short rules. As discussed earlier, if a short rule is the suffix of a long rule and is checked before the long rule, then the long rule will never be applied. As a result, MRST-AT-POS first generates code to match the long rules, by calling META-MRST on them. It then calls EMIT-CHECKS to generate if statements that sequentially match all the remaining short rules—in other words, those that fall into this case of the pattern matching switch statement. These if statements include assignments to and checks on the values of variables, and their bodies directly implement the rewrite actions specified by the rules.

Figure 7-6 provides an example of the generated C code. The code is compiled by an optimizing C compiler to provide a fast tree-based implementation of the given rewrite rules.

7.4 Applications

The binary peephole optimizer is used in the 'C compiler, tcc. At compiler compile time, bog generates a peephole optimizer based on rewrite rules for tcc's target architecture (either MIPS or SPARC). The peephole optimizer can be table-driven or based on MRSTs, depending on the desired run-time performance. Then, during the run time of code compiled by tcc, the peephole optimizer

```
if (ot−0 >= oh) { /* begin bounds check */
switch((ot[ − 0]&0x3)>>0) {
case 0:        switch((ot[ − 0]&0x1000)>>12) {
        case 0:
                if ((tnf || blengthmatch(lhl, lhp, 2))
                    && (ot−1 >= oh)) {
                if (((ot[ − 1]&0xffffff00) == 0xf070100)
                    && ((ot[ − 0]&0xfe03feff) == 0x4020c10)
                    && (((c = ((ot[ − 1]&0xff)>>0)))|1)
                    && (((a = ((ot[ − 0]&0x1fc0000)>>18)))|1)
                    && (((b = ((ot[ − 0]&0x100)>>8)))|1))
                {
                        /* Implements:
                           00001111000001110000000 1c——-
                           0000010a——100000110b00010000
                           =
                           000000010000110000000001100001100
                           0000011b0000011100b01111a——0
                           +
                        */
                        ih[ − 1] = 0x10c030c ;
                        ih[ − 0] = 0x6070f00
                                | ((b<<24)&0x1000000)
                                | ((b<<13)&0x2000)
                                | ((a<<1)&0xfe);
                        ot −= 1;
                        ih −= 1;
                        goto nextpass;
                }
                }
```

Figure 7-6: An example of code generated by the MRST-based algorithm.

is invoked as the last phase of dynamic code generation. The details of tcc's implementation appear in Chapter 4. Binary peephole optimization of dynamic code can be effective: it is separate from the rest of the compiler and therefore does not add complexity to it, and it sometimes improves dynamic code significantly. Section 7.5 below shows some performance results.

Fast binary peephole optimization, however, is not limited to improving dynamic code. For example, bpo was designed to be easy to plug into executable editing frameworks. Experience with a binary editing tool at Microsoft Research confirms that bpo can be useful in this sort of situation.

Furthermore, binary peephole optimizations are in some sense at the bottom of the "food chain" of peephole optimizations[1]. For instance, they can be used to implement other kinds of peephole optimizations. Consider a text-based peephole optimizer that takes rules such as this

```
loadi %reg1, %imm
add %reg2, %reg2, %reg1     # reg2 = reg2 + reg1
=
addi %reg2, %reg2, %imm
```

and applies them to assembly code. Such an optimizer probably represents literal strings, as well as wildcard variables, by their pointers. It is not difficult to implement this optimizer on top of bpo, so that text transformation happens as a result of bpo rewriting a buffer of string pointers and indices (i.e., replacing one pointer with another) rather than through actual change to the contents of the strings.

[1]This insight is due to Chris Fraser.

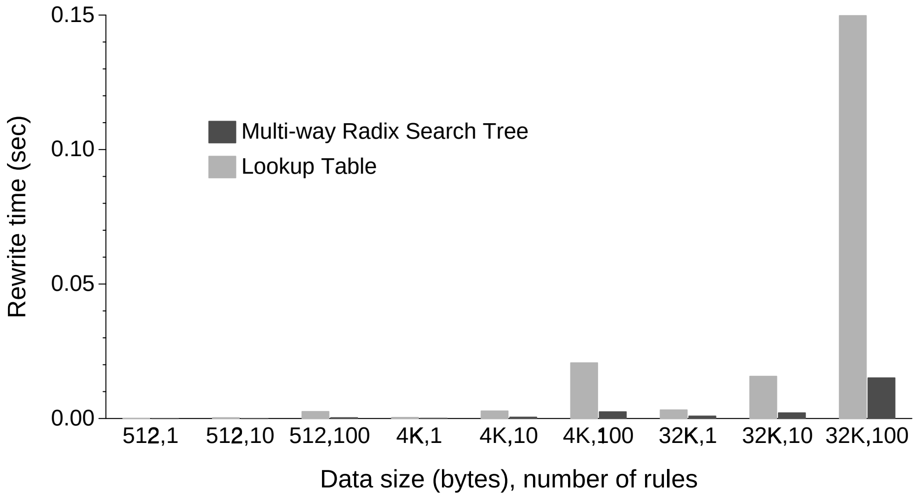

Figure 7-7: Comparative performance of lookup-table algorithm and multiway radix search algorithm. Points on the horizontal axis correspond to input buffers of 512, 4K, and 32K bytes, each processed with 1, 10, and 100 rules. The vertical axis reports processing time in seconds.

Given a text rewriter built on top of bpo, it is possible to go one step further: the bpo implementation uses the text rewriter to simplify the creationg of binary rewrite rules. Rather than writing the bpo binary rewrite rules as long strings of 1s, 0s, and dashes, which is tedious and error-prone, one writes the rewrite rules in an assembly-like syntax similar to the short example above. One then writes a second set of rewrite rules that implement a simple assembler: when applied by the text-based peephole optimizer, these "assembler" rules translate the bpo rules written in assembly-like syntax to real bpo rewrite rules. Writing the assembler rules is a little tricky, but simpler and faster than writing a real translator from assembly to bpo rule syntax, especially since one usually wants to translate only a subset of the machine instructions. Once the assembler rules are written, creating additional bpo rules is simple.

7.5 Evaluation

This evaluation of bpo tries to answer two questions. First, does the extra complexity of the MRST-based pattern matching algorithm relative to a simple table-driven matcher really translate into improved performance *Second, are peephole optimizations actually practical in a dynamic compilation context*

To answer the first question, we compare the performance of the MRST-based and table-driven algorithms on various inputs. Each algorithm was tested with three different rule sets—consisting of one, ten, and 100 rules, respectively—and four different input buffers—with sizes of 512 bytes, 4K bytes, 32K bytes, and 1M byte, respectively. One megabyte is larger than most conceivable dynamic code, but it nonetheless provides a useful data point. Both the rewrite rules and the input buffers were generated randomly. Depending on the number of rules and the size of the input buffers, the number of rules matched (and transformations performed) in one test run varied between two and 200. These measurements were run on an IBM Thinkpad 770 with a 200MHz Pentium MMX processor and 128MB of physical memory. The measurement methodology is the same as that described in Section 4.4.

The results for inputs up to 32K bytes appear in Figure 7-7. The overhead of the MRST-based and table-driven algorithms grows similarly with increasing buffer size: in both cases, the relationship is roughly linear. However, the MRST algorithm scales more effectively than the table-driven algorithm as the number of rules increases. For example, with 32K byte inputs, the MRST

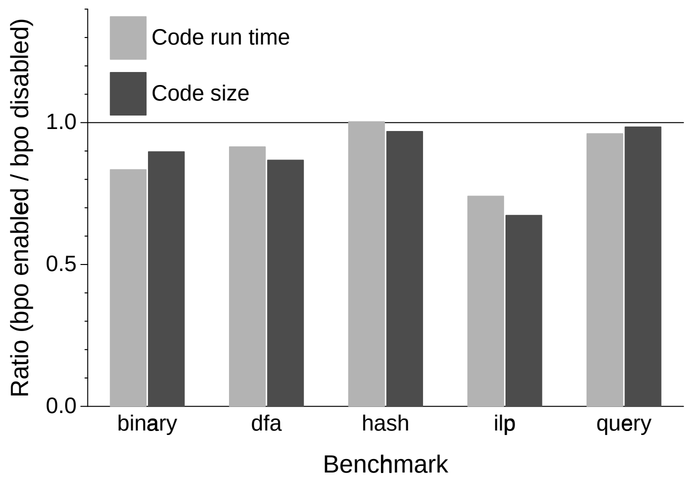

Figure 7-8: Run-time performance of code generated using VCODE and then optimized by bpo. For each benchmark, the left bar is the ratio of optimized to unoptimized run-time; the right bar is the ratio of optimized to unoptimized code length. Values below one mean that the optimized code is faster or smaller.

algorithm is roughly three times faster than the table-driven one when using one rewrite rule, but almost ten times as fast when using 100 rewrite rules. The numbers for 1MB are not plotted in the figure because they swamp the differences for smaller inputs, but the trend is the same: with 1M byte input and 100 rules, the MRST-based routine takes 0.48 seconds, and the table-driven one requires 5.1 seconds. These results coincide with intuition, since the key feature of the MRST-based algorithm is its ability to efficiently distinguish among many rewrite rules.

To answer the second question—whether binary peephole optimizations are actually useful in dynamic compilation—we evaluated bpo with the MRST-based algorithm in the tcc compiler. We measured the effect of bpo on the code generated by VCODE, tcc's fast but non-optimizing dynamic run-time system. Measurements were performed on a Sun UltraSPARC, as described in Section 4.4, using a subset of the benchmarks presented in that section. The peephole optimizations consisted of ten rules that eliminate redundant moves introduced by VCODE to "patch" cspec and vspec composition. The peephole optimizations were not performed in conjunction with ICODE because it does not generate those redundant moves.

Figure 7-8 reports the effect of peephole optimizations on the performance of dynamic code, and Figure 7-9 reports their effect on the dynamic compilation overhead. Figure 7-8 reports two values for each benchmark: the bar on the left is the ratio of dynamic code run time after peephole optimizations to run time before peephole optimizations; the bar on the right shows the ratio of code size, which gives an indication of how many instructions were removed by peephole optimization. Figure 7-9 shows the ratio of dynamic compile time including bpo to dynamic compile time without bpo.

Figure 7-8 indicates that binary peephole optimizations can decrease the run time of generated code by up to 40%, with an average improvement of about 15% across the tested benchmarks. At the same time, as shown in Figure 7-9, they only roughly double the overhead of VCODE, tcc's fast run-time system. In some cases, such as the ilp benchmark, VCODE and bpo together produce code that runs as quickly as that produced by ICODE: the peephole optimizations remove all redundant moves, and there are no other differences, such as extra spills, between VCODE and ICODE. Yet the overhead of VCODE plus bpo is a quarter to a third of ICODE overhead. bpo could be used with ICODE, too—though of course one would need to provide more useful rewrite rules, since ICODE does not usually generate redundant moves. Since the overhead of ICODE is about five times greater than that of VCODE (see Chapter 4), the impact of bpo on its compilation performance is

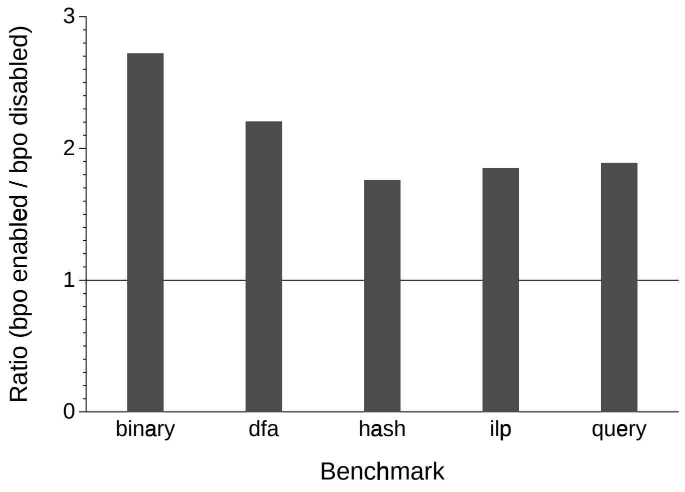

Figure 7-9: Compile-time performance of binary optimization. For each benchmark, the bar is the ratio of the overhead of VCODE and bpo together to the overhead of VCODE alone. For instance, a value of 1.8 means that bpo increases the cost of dynamic compilation by 80%.

lower; however, since the generated code is better to begin with, peephole optimizations are also less useful. In summary, it appears that efficient on-the-fly peephole optimization can be most useful in high-speed, one-pass dynamic compilation systems such as VCODE.

7.6 Summary

This chapter has discussed binary peephole optimizations, simple local optimizations performed directly on executable code. We introduced a fast technique for binary pattern matching based on multiway radix search trees, and described bpo, a binary peephole optimizer that uses this technique to provide fast peephole optimizations within the tcc compiler. Experimental results indicate that the fast matching technique is an order of magnitude faster than a simple table-based matcher. Furthermore, peephole optimizations used in tcc can improve code performance by as much as 40%, yet they only increase the overhead of the fastest code generator by about a factor of two.

Chapter 8

Conclusion

8.1 Summary and Contributions

This thesis has described ways of exposing dynamic code generation to the programmer, discussed its efficient implementation, and presented a number of applications of dynamic code.

8.1.1 Language

Dynamic compilation can be expressed via a library interface or via language extensions. A library can be easier to implement, but a language approach has many other advantages: better performance, better error checking, and greater expressiveness.

Language interfaces to dynamic compilation can be imperative, in which the user explicitly manipulates dynamic code objects, or declarative, in which the user writes annotations that specify run-time constants and specialization policies. The declarative approach, which has been adopted by systems such as Tempo and DyC, has two main benefits. First, it provides greater ease of use in the common case of simple specialization. Second, it enables the static compiler to deduce more of the structure of the dynamic code, and therefore to create better dynamic code at lower overhead.

This thesis has described 'C, currently the only high-level imperative language for dynamic compilation. Relative to the declarative style, the imperative style used by 'C complicates static analysis of dynamic code and therefore requires more work at dynamic compile time. However, it allows greater flexibility and control than most annotation-based systems. It can be used not only to improve performance, but also as a software development tool, to simplify the implementation of programs that involve compilation. In the end, however, the choice of imperative or declarative style is probably a matter of individual taste.

Programming effectively in 'C requires some experience because one must think, in part, at a "meta" level, writing code that specifies the construction of more code. This thesis described several potential extensions to make 'C more user-friendly. However, most of these extensions have some drawbacks: the simple and minimal 'C interface, which in many ways reflects the bare-bones feel of C itself, may well wear best.

8.1.2 Implementation

A dynamic compiler must strike a balance between the overhead of dynamic compilation and the quality of the resulting code. Dynamic compilation overhead can be reduced by moving as much work as possible to static compile time. However, because of its programming model based on dynamic code composition, 'C somewhat limits static analysis. As a result, this thesis presented several algorithms that improve code quality at a fraction of the overhead of traditional optimizations. Linear scan register allocation, for instance, is a global register allocation algorithm that usually produces good code at several times the speed of graph coloring on realistic benchmarks. Fast peephole optimizations and statically staged optimizations are other examples of ways to improve code quality without excessively slowing down compilation.

The thesis also described tcc, a portable implementation of ‘C. tcc supports efficient dynamic code generation, and provides two run-time systems in order to accommodate the tradeoff between dynamic compilation overhead and quality of generated code. The VCODE run-time system generates code in one pass, at an overhead of approximately 100 cycles per instruction; ICODE uses an intermediate representation to optimize code more carefully, and therefore generates one instruction in approximately 600 cycles. In most of the benchmarks in this thesis, the overhead of dynamic code generation is recovered in under 100 uses of the dynamic code; sometimes it can be recovered within one run.

The implementation of tcc has provided useful lessons. First, one must carefully weigh the benefit of an optimization against its overhead. Most "classical" compiler optimizations, such as common subexpression elimination, simply provide too little bang for the buck to be of use in a dynamic context. Allowing the user to choose the level of dynamic optimization—as done in tcc with the two different back ends, ICODE and VCODE—can improve the effectiveness of dynamic compilation across a wide range of applications. Second, "best effort" is usually good enough. Modern static compilers use elaborate graph coloring and register coalescing heuristics to make register allocation as good as possible. Section 5.2, by contrast, describes a simple algorithm that allocates registers in a single sweep of the intermediate form. It is not close to optimal, but it is fast and good enough for most uses. The same philosophy applies to the one-pass data flow analyses in Chapter 6. Lesson three is a reminder that, unfortunately, there is often more to an algorithm than its big-O properties. The fast data flow analyses are linear in the number of basic blocks and several times faster than their traditional analogs, but they are still impractical for lightweight dynamic compilation due to the high overhead of initializing data structures. Finally, lesson four is that ad-hoc solutions can work well. For instance, as described in Chapter 7, peephole optimizations can be used effectively to locally improve code.

8.1.3 Applications

A technique like dynamic code generation is only useful if it provides real performance or software development benefits for some classes of applications. Chapter 3 defined a space of dynamic code generation optimizations and populated it with several examples. We also explored the use of dynamic compilation in specific areas such as numerical methods and computer graphics. ‘C can be used to improve the performance of many programs, from networking routines to math libraries to sorting algorithms. ‘C improves the performance of some benchmarks in this thesis by almost an order of magnitude over traditional C code; speedups by a factor of two to four are not uncommon. In addition to performance, an explicit interface to dynamic compilation can also provide useful programming functionality. For instance, ‘C simplifies the creation of programs that involve compilation or interpretation, such as database query languages and compiling interpreters.

8.2 Future Work

This thesis only scratches the surface of dynamic compilation. Much remains to be done in the area.

8.2.1 Code Generation Issues

The most direct and incremental additions to this thesis are further mechanisms for expressing dynamic code and new algorithms for efficient code generation. Some of the ‘C extensions described in Chapter 2 could be implemented and evaluated in terms of utility and effectiveness. Fast data flow analyses could be studied further; it may be possible to simplify them even more and to decrease their setup cost by coarsening data flow information and by limiting analyses to only certain areas of code. The two-pass register allocation algorithm is quite promising: it should be implemented and evaluated. The same two-pass strategy might be employed to perform other fast analyses across cspecs, similarly to region- or structure-based analyses in traditional compilers. All these techniques are likely to further improve dynamic code quality and generation speed.

8.2.2 Architectural Interactions

Another area to explore is the interaction between dynamic code generation and computer architecture. For instance, dynamic compilation performance improved significantly in passing from an old MicroSPARC to the UltraSPARC used for the measurements in this thesis, primarily thanks to the newer processor's deeper write buffers. On the older processor, VCODE often generated code quickly enough to saturate the write buffers, slowing dynamic compilation down to memory speeds. As another example, unified instruction and data caches can improve dynamic compilation performance because they do not require flushing the instruction cache after dynamic code generation. Larger caches might also encourage more aggressive dynamic loop unrolling and inlining. A detailed study of the effect of various architectural features on dynamic compilation might provide insights that would improve overall system performance.

Similarly, it may be useful to explore the interaction of dynamic compilation and parallelism. Berlin and Surati [6] argue that specialization of high-level code such as Scheme can expose vast amounts of fine-grained parallelism. Such parallelism should become increasingly important as architectures evolve towards greater instruction-issue widths and deeper pipelines. Dynamic specialization should be more practical and easily applicable than off-line, static specialization. Dynamic code generation could therefore play an important role in extracting fine-grained parallelism from applications.

8.2.3 Automatic Dynamic Compilation

A third, and rather more ambitious, research direction is automatic dynamic code generation. All current systems employ annotations or imperative constructs to direct dynamic compilation. Most of these constructs simply give the compiler information about run-time invariants. It should be possible to detect run-time invariants automatically by profiling the traditional static program and keeping track of values.

Automatic dynamic compilation has important potential applications. For instance, it could be used to identify frequently used call paths in complex API layers and then trigger dynamic inlining. Such optimization is already possible with 'C or with a powerful declarative system like DyC, but it would require rewriting the libraries that implement the APIs. Ideally, an automatic profiling and optimization system would only require that the libraries be recompiled—no source code changes would be necessary.

Such an approach is not guaranteed to work: it may fail to discover many opportunities for dynamic compilation, and the overhead of value profiling may well be prohibitive. Furthermore, at best, automatic detection of invariants would provide only the performance benefits of dynamic compilation, without the useful programming functionality of an imperative dynamic code interface. However, if it worked even for a small set of application domains, it would certainly make dynamic code generation a more widespread compiler optimization than it is today.

8.3 Final Considerations

The most important question about dynamic code generation, and one that cannot be answered by research alone, is whether it is useful enough to be adopted in a widespread manner. As a compiler optimization, it is not as applicable as many traditional methods, but when it is applicable, it is quite effective. As a programming technique, its domain is limited to systems that involve compilation or interpretation, but again, in those areas it can provide substantial benefits. On balance, therefore, it deserves attention.

Unfortunately, it is not equally clear what makes a good implementation. Certainly, to break onto the wider programming scene, any implementation would have to be based on an already popular optimizing C compiler, such as GNU or Microsoft C. Most people are understandably reluctant to leave a full-featured and well-integrated compiler like gcc for a less effective experimental one, even if the latter provides a new feature such as dynamic code generation. In retrospect, the decision to use lcc simplified some aspects of development, but it made it more difficult to create efficient code and probably limited the potential user base.

The programming interface to dynamic compilation is the second major factor in its acceptance. A declarative interface would be simpler than ‘C's imperative interface, but it would lose the expressiveness that makes ‘C useful for writing compiler-like programs. Again, the choice is a matter of taste; a real implementation in a popular compiler might actually provide both. A fully automatic dynamic code generation system based on value profiling, like the one proposed in the previous section, would require no user intervention, and could therefore be treated as just another compiler optimization. If it worked—a big if—it would be the best way to make the performance benefits of dynamic code widely available. In the meantime, a combined imperative and declarative implementation based on a widely used compiler like gcc is probably the best way to make this technology available.

Appendix A

A Dynamic Compilation Cookbook

Dynamic compilation finds its most natural and straightforward application in the writing of compiling interpreters and other systems, such as database query languages, in which it is beneficial to remove a layer of interpretation. In these problem domains, a language like ‘C improves both performance and ease of programming relative to a completely static language such as C or Java. However, dynamic compilation can be useful in many other areas, including numerical methods, traditional algorithms, and computer graphics. Whereas Chapter 3 provided a broad overview of the applicability of dynamic compilation, this section contains a more in-depth “cookbook” description of how ‘C can be used to solve problems in each of these three areas.

A.1 Numerical Methods

The Newton’s method code in Section 3.1.3 illustrates the potential of dynamic compilation in numerical code. Here we consider a few other representative examples. In most cases, the examples are adapted from Press et al. [59].

A.1.1 Matrix Operations

As hinted in Chapter 3, matrix and vector code provides many opportunities for dynamic code generation. Such code often involves a large number of operations on values which change relatively infrequently. Furthermore, matrices may have run-time constant characteristics (for instance, size) that can be used by a dynamic compiler to improve performance.

Sparse matrix multiplication

Sparse matrices are generally represented with an indexing scheme that uses two arrays. One array stores the actual data values; the other stores integers that establish a mapping between positions in the first array and positions in the actual dense matrix. Operations on this representation require accessing both arrays, generally in tight loops. For example, Press et al. [59] present a routine for multiplying a sparse matrix by a vector to its right. This routine is one of several that use a similar programming style. The C code for it, below, multiplies a matrix in row-indexed sparse storage arrays sa (that contains values) and ija (that contains indices) by a vector x[1..n], giving a vector b[1..n].

```
void sprsax (float sa[], unsigned ija[], float x[], float b[], unsigned n) {
    unsigned i, k;
    if (ija[1] != n+2) error("Matrix and vector size mismatch.");
    for (i = 1; i <= n; i++) {
        b[i] = sa[i]*x[i];
        for (k = ija[i]; k <= ija[i+1]−1; k++) b[i] += sa[k] * x[ija[k]];
    }
}
```

Assuming that the sparse matrix is used several times, we can use ‘C to create more optimized code, in which the references to the two sparse storage arrays are eliminated, and the tight, doubly nested loop above is unrolled. The ‘C routine below dynamically generates code specialized to multiplying a run-time constant matrix (originally denoted by the row-index sparse storage arrays sa and ija) by an arbitrary vector px[1..n], giving a vector pb[1..n].

```
typedef void (*ftype)(float *, float *, int);
ftype sprsax(float sa[], unsigned long ija[], unsigned long n) {
    float *vspec px = param(float *, 0), *vspec pb = param(float *, 1);
    int vspec pn = param(int, 2); void cspec c; int i, k;
    if (ija[1] != n+2) error("Matrix and vector size mismatch.");
    c = ‘{ if (pn != $n) error("Bad vector size."); };
    for (i = 1; i <= n; i++) {
        c = ‘{ @c; pb[$i] = $sa[i]*px[$i]; };
        for (k = ija[i]; k <= ija[i+1]−1; k++) c = ‘{ @c; pb[$ija[k]] += $sa[k]*px[$i]; };
    }
    return compile(c, void);
}
```

The size of the code generated by this routine is proportional to the number of elements in the sparse matrix. If the matrix is sufficiently large or sufficiently dense, the dynamic code may actually be slower than equivalent static code due to poor cache behavior. As explained in Chapter 3, dynamic code size often depends on the data that is being processed. This fact should be taken into account when considering many of the examples below.

Matrix factorization

One important matrix factorization technique is Cholesky decomposition. This method can be used to solve linear equations about twice as quickly as alternative techniques [59]. Although it places some special requirements on its inputs, these requirements are satisfied frequently enough in practice to make it useful. Like in the sparse matrix routines above, solving linear equations by Cholesky decomposition involves tight loops and array accesses. The ‘C routine below takes advantage of the fact that often one wants to find a solution to the system of equations denoted by $A \cdot x = B$ for several different right-hand sides B. It produces optimized code specialized for A, in which loops are unrolled and array indices are run-time constants. In the code, a is the Cholesky factorization of the positive-definite symmetric matrix A and p contains its diagonal elements. The dynamic code can be called multiple times with different values of b and stores the result in the passed-in array x.

```
typedef void (*ft)(float *, float *);
ft mkcholsl (float **a, int n, float p[]) {
    float *vspec pb = param(float *, 0), *vspec px = param(float *, 1), vspec sum = local(float);
    void cspec c = ‘{}; int i, k;
    for (i = 0; i <= n; i++) {
        c = ‘{ @c; sum = pb[$i]; };
        for (k = i−1; k >= 1; k−−) c = ‘{ @c; sum −= a[$i][$k]*px[$k]; };
        c = ‘{ @c; px[$i] = sum/p[$i]; };
    }
    for (i = n; i >= 1; i−−) {
        c = ‘{ @c; sum = px[$i]; };
        for (k = i+1; k <= n; k++)  c = ‘{ @c; sum −= a[$k][$i]*px[$k]; };
        c = ‘{ @c; px[$i] = sum/p[$i]; };
    }
    return f = compile(c, void);
}
```

Note that the values of the run-time invariant a and p arrays are not actually treated as run-time constants because they contain floating point data. Since floating point values cannot be encoded as immediates, denoting them as run-time constants would provide no benefit. Nonetheless, even loop unrolling alone can be beneficial. For instance, the toy loop

```
for (i= 1; i <= k; i++) { t += d[i]; t /= d[i]; }
```

requires only three instructions per iteration when unrolled on an Intel x86. When not unrolled, each iteration requires six: the extra three instructions are the loop counter increment, termination test, and branch. Of course, excessive unrolling can cause cache problems, but the dynamically generated code is effective for relatively small matrices.

Another popular matrix factorization technique is QR decomposition. Like Cholesky decomposition, QR decomposition involves routines with tight loops that iterate over a matrix, so it also is amenable to the optimizations discussed above.

A.1.2 Numerical Integration

Numerical integration can benefit significantly from dynamic inlining. Consider, for instance, integration by the trapezoidal rule. The coarsest refinement approximates

$$\int_a^b f(x)dx \approx (b-a) * (f(a) + f(b))/2.$$

The second refinement adds the value of f at $(a+b)/2$ to the average, the next refinement adds the values at the 1/4 and 3/4 points, and so forth. At each stage, the approximation is more accurate. The following routine, adapted from Press et al. [59], computes the nth refinement of the trapezoidal rule for $\int_a^b f(x)dx$:

```
typedef float (*floatfunc)(float);
float trapzd (floatfunc f, float a, float b, int n) {
    float x,sum,del; static float s; static int it; int j;
    if (n == 1) { it = 1; return s = (b-a)/2 * (f(a) + f(b)); }
    else {
        del = (b-a) / it; x = a + del/2;
        for (sum = 0.0, j = 1; j <= it; j++, x += del) sum += f(x);
        s = (s + (b-a)*sum/it) / 2; it *= 2;
        return s;
    }
}
```

Each invocation of this code with sequentially increasing values of n ($n > 1$) improves the accuracy of the approximation by adding 2^{n-2} internal points [59]. The code is usually called repeatedly to achieve the desired level of accuracy. The function pointed to by f, which may be unknown at compile time, is therefore used several (2^{n-2}, for $n > 1$) times within the code, and the code itself is called many times. We can improve the code's performance in the same way as the Newton's method routine in Section 3.1.1, by replacing the function pointer with a cspec and creating a specialized routine in which the function is dynamically inlined:

```
typedef float (*trapzfunc)();
typedef float cspec (*dfloatfunc)(float vspec);
trapzfunc mktrapz (dfloatfunc f) {
    float vspec a = param(float, 0), b = param(float, 1);
    int vspec n = param(int, 2);
    void cspec c = `{
        float x,sum,del; static float s; static int it; int j;
        if (n == 1) { it = 1; return s = (b-a)/2 * (f(a) + f(b)); }
        else {
            del = (b-a) / it; x = a + del/2;
            for (sum = 0.0, j = 1; j <= it; j++, x += del) sum += f(x);
```

```
                s = (s + (b−a)*sum/it) / 2; it *= 2;
                return s;
            }
        };
        return compile(c, float);
    }
```

This use of ‘C does not degrade readability: the code inside the backquote expression is identical to the body of the original static function. Additional performance benefits could be obtained by dynamically unrolling the loop, but such a change would complicate the code. The dfloatfunc in the code is a ‘C function that returns a float cspec corresponding to the function we need to integrate. For instance, for $f(x) = x^2$:

```
    float cspec f(float vspec x) { return ‘(x*x); }
```

Similar benefits can be obtained with other integration techniques, such as Gaussian quadratures. Multidimensional integration provides further opportunities for optimization: since it uses one-dimensional integration routines like the one above as building blocks, it can potentially benefit from multiple levels of dynamic inlining.

A.1.3 Monte Carlo Integration

The Monte Carlo method is a well-known and widely used technique for numerically integrating functions over complicated multidimensional volumes that are difficult to sample randomly—for instance, when the limits of integration cannot easily be written in closed form. The idea is to enclose the complicated volume within a simpler volume that is easy to sample randomly. We then pick random points within this simpler volume, and estimate the complicated volume as the simple volume multiplied by the fraction of random points that fall within the complicated volume.

Consider the following code fragment, which uses the Monte Carlo method to compute $\int_a^b f(x)dx$ for some f such that $0 < y_1 < f(x) < y_2$ for $x \in [a, b]$:

```
    sum = 0;
    for (j = 0; j < N; j++) { /* Loop for some number of iterations */
        x = randinterval(a,b); /* Pick a random number in [a,b] */
        y = randinterval(y1,y2); /* Pick a random number in [y1,y2] */
        if (y < f(x)) sum++; /* If we hit inside f, increment the count */
    }
    a = (b−a)*(y2−y1);
    intf = a*sum/N; /* integral(f) is area of [a,b]x[y1,y2] times fraction of samples in f */
```

Thanks to the constraints that we chose for f, this code is straightforward. However, depending on the volume being integrated, the test that determines whether a random point falls within the region of interest (y < f(x), in this case) can be arbitrarily complicated. The whole code, in fact, depends on the function that is being integrated and its dimensionality.

In general, one writes such a Monte Carlo routine from scratch for every function of interest. With ‘C, one can imagine writing a generic “Monte Carlo builder” function that takes cspecs corresponding to the function being integrated and its bounds, as well as other information about the dimensionality of the problem and other parameters, and dynamically constructs a function optimized for that problem. This solution enables Monte Carlo integration that is both efficient and interactive (dynamically modifiable).

In addition to the simple Monte Carlo technique, there exist more advanced methods, such as vegas and miser [59], that converge more rapidly. These routines are generic—they take a function pointer that denotes the function of interest. Like some routines described earlier, they may also benefit from dynamic optimizations such as inlining and loop unrolling.

A.1.4 Evaluation of Functions

Certain mathematical functions can be evaluated more efficiently with dynamic compilation. We discuss just a few examples.

Polynomials

Consider a polynomial with integer coefficients. If we are given the task of repeatedly evaluating such a polynomial at different points, we may want to write a routine like this:

```
typedef float (*polyfunc)();
polyfunc mkpoly (int c[], int nc) { /* Coefficients; c[0] is constant term */
    float vspec x = param(float, 0); /* Evaluate the poly at point x */
    float cspec y = `(float)$c[nc];
    int i;
    for (i = nc−1; i >= 0; i−−) y = `(y*x+$c[i]);
    return compile(`{ return y; }, float);
}
```

This code dynamically builds up an expression of the form (((c[4]*x+c[3])*x+c[2])*x+c[1])*x+c[0]. It avoids the loop overhead normally associated with polynomial evaluation; furthermore, since the coefficients are integers, it may be able to reduce some of the multiplications in strength. If the polynomial is evaluated several times, the overhead of code generation is easily recouped.

Similar optimizations can be applied to more complex routines that simultaneously evaluate multiple derivatives of polynomials or that perform multiplication or division of polynomials. Rational functions—the quotients of two polynomials—can also be optimized in this way.

Numerical derivatives

Dynamic inlining can be useful for computing numerical derivatives. For instance, Ridder's method for computing derivatives of a function by polynomial extrapolation [59] requires the function to be invoked several times within a loop—inlining offers obvious advantages. The algorithm may also benefit from dynamic loop unrolling and constant folding.

Root finding

The Newton's method code in Section 3.1.1 is one example of root-finding code that benefits from dynamic compilation. Most other popular methods—bracketing, bisection, the secant method, and others—can be optimized analogously thanks to dynamic inlining and, in some cases, loop unrolling.

Ordinary differential equations

Likewise, the Runge-Kutta method and other routines for solving initial value problems for ODEs are amenable to specialization and inlining of user-supplied functions.

A.1.5 Fast Fourier Transform

The fast Fourier transform (FFT), ubiquitous in signal processing, can be made a little faster by dynamically specializing it to the size of its input data set. For instance, the first part of the algorithm is a relatively tight loop that performs bit reversal and whose behavior depends only on the size—not the values—of the input:

```
for (i = 1; i < n; i += 2) {
    if (j > i) { swap(data[j], data[i]); swap(data[j+1], data[i+1]); }
    m = n>>1;
    while (m >= 2 && j > m) { j −= m; m >>= 1; }
    j+= m;
}
```

This code can be rewritten in ‘C:

```
c = ‘{};
for (i = 1; i < n; i += 2) {
    if (j > i) c = ‘{ @c; swap(data[$j], data[$i]); swap(data[$j+1], data[$i+1]); };
    m = n>>1;
    while (m >= 2 && j > m) { j −= m; m >>= 1; }
    j+= m;
}
```

where c is a cspec that accumulates a straight-line sequence of swap routines that implement optimal bit reversal reordering for this input size. If the code will be used several times with inputs of the same size, the specialized code will be significantly faster than the general loop. These benefits may be even greater for multidimensional FFTs, in which the loop structure is more complicated.

The second part of the algorithm contains bigger nested loops that are less amenable to unrolling, but it may still benefit from dynamic optimizations for small inputs.

A.1.6 Statistics

One important code example from statistics is the Kolmogorov-Smirnov test, which can be used to determine whether two unbinned distributions, each a function of a single independent variable, are different. Like much of the integration and root-finding code, this routine uses a function pointer in a tight loop and makes frequent calls to math routines to compute maximum and absolute values. These calls could be inlined dynamically to increase performance. Again, similar optimizations apply to higher-dimensional versions of the algorithm.

Other statistics routines, such as linear least squares code, can also benefit from dynamic loop unrolling and other optimizations based on input size.

A.1.7 Arbitrary Precision Arithmetic

Dynamic compilation can also provide efficient support for arbitrary precision arithmetic. For instance, Press et al. [59] present a math library in which values are stored in unsigned character strings that are interpreted as base 256 numbers. Values are stored with most-significant digits first (earlier in the array). Carries are handled by using a variable of short type to store intermediate values. Here is a simple addition routine:

```
#define LO(x) ((unsigned char) ((x) & 0xff))
#define HI(x) ((unsigned char) ((x) >> 8 & 0xff))
void mpadd(unsigned char *w, unsigned char *u, unsigned char *v, int n) {
    /* w = u+v; u and v have length n; w has length n+1 */
    int j; unsigned short t = 0;
    for (j = n; j >= 1; j−−) { t = u[j] + v[j] + HI(t); w[j+1] = LO(t); }
    w[1] = HI(t);
}
```

Although code like this can handle inputs of arbitrary length, the values may frequently fall within a relatively small range. In this case, one can use ‘C to dynamically generate code specialized to a particular input length, removing loops and making array offsets constant. The code could be recompiled when the lengths of inputs change, or one could generate several functions, each specialized to a different input length. These optimizations should be at least as useful for more complicated operations, such as multiplication of two arbitrary-length multiple precision values.

Furthermore, if one of the operands is to be employed many times, it may be beneficial to dynamically specialize the relevant operations with respect to that operand. In the simple addition example, it might then be possible to optimize away some of the addition and carry operations. Such specialization effectively transforms the multiple precision vector into an executable data structure that knows how to apply itself to other data—a philosophy almost reminiscent of Smalltalk and other object-oriented systems.

A.2 Non-Numerical Algorithms

Dynamic code generation can improve the performance of many traditional algorithms. The specialized hash table and the executable binary search tree described in Section 3.1.1 are two examples. This section describes some others.

A.2.1 Sorting Algorithms

Mergesort

Mergesort is an important sorting algorithm. Like quicksort, it has "optimal" $O(n \log n)$ asymptotic performance. Unlike quicksort, however, it is stable, and it insensitive to the initial order of the input—by contrast, quicksort is quadratic in the worst case. Furthermore, if implemented cleverly, mergesort's inner loop is shorter than quicksort's, and it is executed on average 38% fewer times [64]. Consider the mergesort code below, which uses an auxiliary array b:

```
void mergesort (int a[], int l, int r) {
    int i, j, k, m;
    if (r > l) {
        m = (r+l)/2; mergesort(a, l, m); mergesort(a, m+1, r);
        for (i = m+1; i > l; i−−) b[i−1] = a[i−1];
        for (j = m; j < r; j++) b[r+m−j] = a[j+1];
        for (k = l; k <= r; k++) a[k] = b[i] < b[j] ? b[i++] : b[j−−];
    }
}
```

This code from Sedgewick [64] is quite efficient. It avoids the use of sentinels in the sorting loop (the third for loop) by copying the second half of the subarray (the part sorted by the second recursive call) back-to-back to the first, but in reverse order (in the second for loop). In this way, the largest elements of each array are adjacent, and each array acts as a sentinel for the other in the sorting loop. However, the code is still not as fast as it could be: it copies data from a to b every time (using the first two for loops), which amounts to $\log n$ copies of the entire array.

We can remove all but one of these copies by alternately merging from a into b and from b into a. Alternating the merge destination while avoiding loop sentinels unfortunately requires four mutually recursive routines, shown below. ainc and adec merge the sorted subarrays into a, just like the code above; binc and bdec merge into b. ainc and binc leave their output in increasing order; adec and bdec leave their output in decreasing order. For instance, ainc avoids the copy for loops in the code above by calling binc to sort the first half of the interval into ascending order and bdec to sort the second half into descending order. Producing output in descending order without using sentinels requires inverting the test in the sorting loop (b[i] > b[j] rather than b[i] < b[j] in adec) while sorting the first half of the interval in decreasing order and the second half in increasing order. The top-level mergesort routine simply copies a into the auxiliary array b once, so that lower levels of the recursion will access correct values, and then calls ainc, which sorts a into ascending order. Here is the optimized code:

```
void mergesort (int a[], int l, int r) {
     int i;
     for (i = l; i <= r; i++) b[i] = a[i];
     ainc(a, b, l, r);
}
void ainc (int a[], int b[], int l, int r) {
     int i, j, k, m;
     if (r <= l) return;
     m = (r+l)/2; binc(a, b, l, m); bdec(a, b, m+1, r);
     for (i = k = l, j = r; k <= r; k++) a[k] = b[i] < b[j] ? b[i++] : b[j--];
}
void adec (int a[], int b[], int l, int r) {
     int i, j, k, m;
     if (r <= l) return;
     m = (r+l)/2; bdec(a, b, l, m); binc(a, b, m+1, r);
     for (i = k = l, j = r; k <= r; k++) a[k] = b[i] > b[j] ? b[i++] : b[j--];
}
void binc (int a[], int b[], int l, int r) {
     int i, j, k, m;
     if (r <= l) return;
     m = (r+l)/2; ainc(a, b, l, m); adec(a, b, m+1, r);
     for (i = k = l, j = r; k <= r; k++) b[k] = a[i] < a[j] ? a[i++] : a[j--];
}
void bdec (int a[], int b[], int l, int r) {
     int i, j, k, m;
     if (r <= l) return;
     m = (r+l)/2; adec(a, b, l, m); ainc(a, b, m+1, r);
     for (i = k = l, j =r; k <= r; k++) b[k] = a[i] > a[j] ? a[i++] : a[j--];
}
```

Of course, this code is still not perfect. Although it performs no sentinel checks or redundant copies, it contains several mutually recursive calls and tight loops. One could avoid the recursive calls by implementing an iterative version of the algorithm, but that would still leave a good deal of loop overhead. Furthermore, the iterative implementation is far less natural and intuitive than the recursive one.

One frequently needs to sort several different arrays that all have the same length. In that case, ‘C can help us write a specialized mergesort routine that, in the limit, performs no calls and no loops. The insight necessary to write this code is that mergesort—unlike, for instance, quicksort—performs the same set of comparisons no matter what the input data. Only the size of the input array matters. Given this insight, we can write the code below. The code closely mirrors the optimized recursive implementation discussed above, but rather than sorting an array, it returns a specification for dynamic code that sorts an array of a given size. The top-level routine, mk_mergesort, dynamically compiles this cspec and returns the function pointer to the caller:

```
int *vspec a, *vspec b, vspec i, vspec j;
void (*mk_mergesort(int len))() {
     a = param(int *, 0); b = local(int[len]); i = local(int); j = local(int);
     return compile(‘{ int k; for (k = 0; k < $len; k++) b[i] = a[i]; @mainc(0, len); }, void);
}
void cspec mainc (int l, int r) {
     int m = (r+l)/2;
     if (r <= l) return ‘{};
     return ‘{ int k; @mbinc(l, m); @mbdec(m+1, r); i = $l; j = $r;
               for (k = $l; k <= $r; k++) { a[$k] = b[i] < b[j] ? b[i++] : b[j--]; } };
}
void cspec madec (int l, int r) {
     int m = (r+l)/2;
```

```
          if (r <= l) return `{};
          return `{ int k; @mbdec(a, b, l, m); @mbinc(m+1, r); i = $l; j = $r;
                   for (k = $l; k <= $r; k++) { a[$k] = b[i] > b[j] ? b[i++] : b[j--]; } };
     }
     void cspec mainc (int l, int r) {
          int m = (r+l)/2;
          if (r <= l) return `{};
          return `{ int k; @mainc(l, m); @madec(m+1, r); i = $l; j = $r;
                   for (k = $l; k <= $r; k++) { b[$k] = a[i] < a[j] ? a[i++] : a[j--]; } };
     }
     void cspec mbdec (int l, int r) {
          int m = (r+l)/2;
          if (r <= l) return `{};
          return `{ int k; @madec(l, m);  @mainc(m+1, r); i = $l; j = $r;
                   for (k = $l; k <= $r; k++) { b[$k] = a[i] > a[j] ? a[i++] : a[j--]; } };
     }
```

The generated code is excellent: it contains no calls or loops, and performs only the minimal set of pointer increments and element compare-and-swap operations needed to mergesort an array of the given length.

Counting sort

Counting sort is an algorithm that can be used to sort in linear time when the input is known to be N integers between 0 and $M-1$ (or, more generally, N records whose keys are such integers). Below is a standard C implementation of counting sort, once again taken from Sedgewick [64]:

```
for (j = 0; j < M; j++)  count[j] = 0;
for (i = 1; i <= N; i++) count[a[i]]++;
for (j = 1; j < M; j++)  count[j] += count[j-1];
for (i = N; i >= 1; i--) b[count[a[i]]--] = a[i];
for (i = 1; i <= N; i++) a[i] = b[i];
```

Since the loop bodies are very small, loop overhead is a large fraction of the total run-time cost. For instance, the first loop consists of a store, an increment, a comparison, and a branch: half of these instructions could be removed by complete unrolling. The overhead could also be reduced by partial unrolling at static compile time, but often we know at run-time the size of the data we will be processing, so it may be useful to completely optimize the code for that case. ‘C makes this optimization straightforward:

```
return `{
     int i, j;
     for (j = 0; j < $M; j++)  count[j] = 0;
     for (i = 1; i <= $N; i++) count[a[i]]++;
     for (j = 1; j < $M; j++)  count[j] += count[j-1];
     for (i = $N; i >= 1; i--) b[count[a[i]]--] = a[i];
     for (i = 1; i <= $N; i++) a[i] = b[i];
};
```

This code returns a cspec that corresponds to code specialized to perform a counting sort on an input of length N with integers between 0 and $M-1$. The contents of the backquote expression is identical to the original code, with the exception that the loop bounds are marked as run-time constants. At run-time the loops can be completely unrolled based on the values of the loop bounds. The dynamic code will have no loop overhead and no index increments; it will consist of the minimal sequence of loads and stores and use constant array offsets.

A.2.2 Graph Algorithms

Shortest Paths

Dynamic loop unrolling can also be useful for certain graph algorithms. Consider the Bellman-Ford single-source shortest path algorithm: its main loop repeatedly relaxes each edge in the graph, and successively finds all shortest paths of at most one hop, then at most two hops, and so forth, up to the maximum diameter of the graph [15]:

```
for (i = 0; i < num_vertices−1; i++)
    for (j = 0; j < num_edges; j++)
        if (dist[destination[j]] > (d0 = dist[source[j]] + wt[j]))
            dist[destination[j]] = d0;
```

If one wanted to perform multiple relaxations using the same graph topology but different edge weights, then dynamic code generation could be used to remove loop overhead and eliminate the reads from the destination and source arrays that specify edge endpoints.

Depth-First Search

'C can also decrease the data structure lookups required for other types of graph traversals. The specialized binary search code in Section 3.1.1 exemplifies specialization in the case of a simple and regular acyclic graph, a binary search tree. Another example is a generic "executable graph" specialized to performing depth-first search. Depth-first search is usually implemented recursively; at each node, the algorithm loops through all the node's neighbors that have not been visited yet, recursing on each one in turn. If we will be performing several searches on the same graph, we can replace the graph data structure by an array of function pointers, each pointing to a dynamically generated function that is specialized to one of the nodes in the original graph. Each function makes a hard-wired sequence of calls to functions that correspond to its node's neighbors, without having to look into any data structure or perform any loop. Implementing this code would be somewhat more difficult than standard depth-first search, but the potential performance benefit may justify it in some situations.

A.2.3 String and Pattern Matching

String searching and pattern matching easily lend themselves to dynamic optimization. In many such algorithms, the string or pattern to be matched can be viewed as a state machine that processes the input. Dynamic code generation allows us to "compile away" these state machines and create code that is specialized to a particular search pattern.

Consider for example the following implementation of the popular and efficient Knuth-Morris-Pratt string search algorithm, adapted from Sedgewick [64]:

```
void initnext (char *p) {
    int i, j, M = strlen(p);
    next[0] = −1;
    for (i = 0, j = −1; i < M; i++, j++, next[i] = (p[i] == p[j]) ? next[j] : j)
        while (j >= 0 && p[i] != p[j]) j = next[j];
}
int kmps (char *p, char *a) {
    int i, j, M = strlen(p);
    initnext(p);
    for (i = 0, j = 0; j < M; i++, j++)
        while (j >= 0 && a[i] != p[j]) j = next[j];
    return i−8;
}
```

The actual sort is performed by the kmps routine. To avoid a test for end of input at every loop iteration, the code assumes that a copy of the search pattern is stored as a sentinel at the end of the input text.

The initnext function is called by kmps to initialize the next array, which determines how far kmps should back up in the search string when it detects a mismatch. The next array allows kmps to move through the input monotonically from left to right, possibly backing up in the search string but never in the input text. For instance, consider searching for "10100" in the input "1010100": when the algorithm detects a mismatch at the fifth input character, it does not back up four characters in the input. Rather, it backs up to the third character in the search string, since the first two characters in the search string match the last two characters that were matched prior to the final mismatch. If we append the following code to initnext,

```
for (i = 0; i < M; i++)
   printf("s%d: if (a[i] != '%c') goto s%c; i++;"n", i, p[i], "m0123456789"[next[i]+1]);
```

and then add a little boilerplate, we obtain the following optimized function:

```
int kmps_10100 (char *a) { /* Match "10100" in string a */
     int i = −1;
 sm: i++;
 s0: if (a[i] != '1') goto sm; i++;
 s1: if (a[i] != '0') goto s0; i++;
 s2: if (a[i] != '1') goto sm; i++;
 s3: if (a[i] != '0') goto s0; i++;
 s4: if (a[i] != '0') goto s2; i++;
 s5: return i−5;
}
```

It should now be clear that next simply encodes the finite state automaton for matching the given input pattern! Although the specialized code is much more efficient than the general kmps routine, it is impractical to statically create all the specialized routines that may be needed. Fortunately, ‘C allows us to create such specialized code dynamically. The mk_kmps function below returns a cspec for a function that is patterned after the kmps_10100 example above but searches for the arbitrary run-time determined input string p. For clarity, the code expects the next array to have already been computed by initnext, but it is not difficult to merge initnext and mk_kmps into one routine.

```
void cspec mk_kmps (char *p, int *next) {
     int i, M = strlen(p);
     int vspec iv = local(int); char * vspec av = param(char *, 0);
     void cspec code, cspec *labels = (void cspec *)malloc((M+1) * sizeof(void cspec));
     for (i = 0; i < M+1; i++) labels[i] = label();
     code = ‘{ iv = −1; @labels[0]; iv++ };
     for (i = 0; i < M; i++)
          code = ‘{ @code; @labels[i+1]; if (av[iv] != $p[i]) jump labels[next[i]+1]; iv++; };
     return ‘{ @code; return iv−8; };
}
```

A.2.4 Circuit Simulation

Similarly, dynamic compilation might be used to remove interpretation overhead in some kinds of circuit simulation. Traditional static simulations must either rely on truth tables and other data structures that describe the circuit structure, or statically compile the circuit descriptions to optimized code. One can use ‘C to implement a simulator that has the flexibility and interactive quality of the first solution without sacrificing the performance of the second solution. For instance, a data structure that described a user-specified series of combinational elements would be dynamically compiled to straight-line code that directly used hardware instructions (and, or, and so forth) to implement the circuit without any data structure overhead.

A.2.5 Compression

Some compression algorithms can also be made more efficient by using dynamic code generation to strip away a layer of interpretation. Consider, for example, the following simplified Huffman encoding routine. The code is adapted from Press et al. [59]; for clarity, error and bounds checking have been omitted.

```
int hufenc (unsigned ch, unsigned char *out, unsigned nb, huff *code) {
    int l, n; unsigned nc;
    for (n = code—>ncod[ch]—1; n >= 0; n——, ++nb)
        if (code—>icod[ch] & (1<<n)) {
            nc = (nb >> 3); l = nb & 7; (*out)[nc] |= (1<<l);
        }
    return nb;
}
```

The routine encodes the single character ch according to the Huffman code in the data structure code, and writes the output into the out array starting at bit position nb. The function loops through the bits of the character's Huffman code (denoted by code->icod[ch]), and places them into the appropriate bit position (nb) in the output. Even without a detailed understanding of Huffman coding, it is easy to see that the sequence of actions performed by this routine is specified completely by the encoding data structure.

Since the function must be called repeatedly to encode consecutive characters in a message, any inefficiencies can have substantial impact. One solution is to use ‘C to dynamically generate several pieces of code, each specialized to encode one character; we can then stitch these pieces together to create an optimized encoding function that never needs to interpret the encoding data structure. Consider the code below:

```
void cspec mk_hufenc (unsigned ch, unsigned char *vspec out, unsigned vspec nb, huff *code,
                  int vspec l, unsigned vspec nc) {
    int n, nbi = 0;
    void cspec c = ‘{};
    for (n = hc—>ncod[ch]—1; n >= 0; n——)
        if (hc—>icod[ch] & (1<<n)) {
            if (nbi) c = ‘{ @c; nb += $nbi; };
            c = ‘{ @c; nc = nb >> 3; l = nb & 7; (*out)[nc] |= (1<<l); };
            nbi = 1;
        } else nbi++;
    }
    return c;
}
```

This ‘C function takes in a character (ch) and a Huffman code (code), and specifications (vspecs) for an output vector, a bit index, and a couple of temporaries. It specifies the optimal straight-line code necessary to store the bits corresponding to the Huffman encoding of ch at some run-time determined bit-offset in the output vector. The variable nbi accumulates consecutive bit counter increments, minimizing the number of such increments performed by the dynamic code. Similarly, conditionals and lookups in the code data structure are performed only at dynamic compile time. This routine can be invoked on every character in the input alphabet before doing any compression, and the returned cspecs can be combined into a large jump table indexed by input character. The resulting code will have only one large loop (over the characters that must be encoded) and no function calls—just an indirect jump to straight-line code to encode each character.

A.3 Computer Graphics

Computer graphics offers many applications for dynamic code generation. For instance, Draves [21] describes how to improve the performance of a lightweight interactive graphics system via dynamic compilation. The exponentiation example in Section 3.1.1 is derived from his code. Similarly, the customized copy code in Section 3.1.2 is inspired by the high performance graphics bitBlt work [55]. This section touches on just a few other examples.

A.3.1 Clipping

All line clipping algorithms, such as the Cohen-Sutherland algorithm and the Lian-Barsky algorithm [27] repeatedly use the endpoints of the clipping rectangle in their computation. If a particular clipping rectangle is used frequently, we can hardwire its endpoints into the algorithm and obtain advantages similar to those of the specialized hash function in Section 3.1.1. The same optimization can be performed when these algorithms are extended to 3 dimensions. Similarly, we can use dynamic compilation to optimize clipping polygons against polygons: if one polygon is repeatedly used as a clipping mask, the Weiler algorithm [27] can be specialized to eliminate the overhead of traversing the data structure representing that polygon. This trick is similar to the vector dot product example in Section 3.1.1.

A.3.2 Shading

An analogous optimization can be performed with shading algorithms. Consider the Phong model, where the amount of light specularly reflected is written as

$$I = I_a k_a O_d + f_{att} I_p [k_d O_d (\bar{N} \cdot \bar{L}) + k_s (\bar{R} \cdot \bar{V})^n].$$

O_d and k_d describe the object's diffuse color, I_a and k_a describe ambient light conditions, I_p describes point light intensity, and $\bar{N}$, $\bar{L}$, $\bar{R}$, $\bar{V}$ denote the orientation of the object, the direction of the light source, the direction of reflection, and the orientation of the viewer, respectively. This formula, as well as others like it, has many terms, but not all of them may change rapidly. A language like 'C can be used to dynamically partially evaluate the expression with respect to terms that change infrequently, and to do this repeatedly as those terms slowly change.

A.3.3 Scan Conversion of Conics

Scan-conversion of conic sections offers another opportunity for low-level optimization based on dynamic partial evaluation. A general conic can be described as the solution to an equation of the form

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0.$$

The algorithm to efficiently plot such a curve is complicated and uses the six coefficients repeatedly. Special cases for circles or ellipses whose axes are aligned with the axes of the plane are simpler, but still use several integer parameters repeatedly. If some of these parameters remain constant across several uses—for instance, if one is drawing many similar circles at different points in the plane—then dynamic specialization to those run-time constant parameters can be an effective optimization.

A.3.4 Geometric Modeling

Geometric models of an object consist of data structures that describe the object's shape and connectivity [27]. Models are complex and often hierarchical; for instance a robot may consist of a body and limbs, and limbs may themselves be built out of segments and joints. The three-dimensional "world" coordinates contained in the model are projected onto the two-dimensional screen coordinates to reflect the position and orientation of the object and the viewer. These matrix-based transformations can be expensive: a viewing engine takes in the geometric model and position information, and outputs a two-dimensional representation. It should be possible to improve on this process by expressing the geometric model as an executable data structure. The viewing engine could take a code specification that describes the model and compile it to code that takes position information and outputs a two-dimensional representation. This process is simply dynamic partial evaluation of the traditional viewing engine with respect to a geometric model. It removes the overhead of accessing complex data structures, and provides the opportunity for strength reduction, loop unrolling, and other optimizations based on the values in the model. Since most objects are likely to be long-lived (at least on the order of a few seconds), the overhead of dynamic compilation should be easily recouped.

A.3.5 Image Transformation

One important geometric transformation is scaling; a standard image expansion algorithm is the Weiman algorithm. Given a scaling factor p/q, where p and q are coprime integers, the Weiman algorithm uses a bit vector of length p called the Rothstein code to decide how to evenly distribute q pixels into p columns [27]. The Rothstein code does not change throughout a particular scaling (although it may be cyclically permuted to improve the destination image quality), and it is referenced for every column in the image. Dynamically generating a specialized scaling routine for a particular Rothstein code may therefore be quite profitable.

Dynamic compilation is also useful for improving the performance of image filtering algorithms—for instance, convolution-based algorithms such as blurring, edge-detection, and anti-aliasing. The values of the convolution masks are constant throughout the filtering operation. Furthermore, for all but an outer strip along the border of the image, boundary checking is unnecessary. A dynamically generated convolution routine in which the values of the convolution mask are hardwired into the code and boundary checks are eliminated is significantly faster than general-purpose code. Even for relatively small images, the cost of code generation can be recouped with just one filtering operation. Section 4.5.1 describes our experiences with filter specialization in the xv image manipulation package.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159. ACM, May 1996.

[5] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, June 1994. ACM.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995. ACM.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[9] D.S. Blickstein, P.W. Craig, C.S. Davidson, R.N. Faiman, K.D. Glossop, R.P. Grove, S.O. Hobbs, and W.B. Noyce. The GEM optimizing compiler system. *Digital Equipment Corporation Technical Journal*, 4(4):121–135, 1992.

[10] P. Briggs and T. Harvey. Multiplication by integer constants, 1994. `http://softlib.rice.edu/MSCP`.

[11] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6:47–57, 1981.

[12] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989. ACM.

[13] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[14] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, FL, January 1996. ACM.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989. ACM.

[17] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[18] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[19] J. W. Davidson and C. W. Fraser. Automatic inference and fast interpretation of peephole optimization rules. *Software Practice & Experience*, 17(11):801–812, November 1987.

[20] P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984. ACM.

[21] S. Draves. Lightweight languages for interactive graphics. Tech. Rep. CMU-CS-95-148, Carnegie Mellon University, 1995.

[22] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–170, Philadelphia, PA, May 1996. ACM.

[23] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, January 1995. ACM.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-specific resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995. ACM.

[25] D. R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272, San Jose, CA, October 1994. ACM.

[26] Ú. Erlingsson, M. Krishnamoorthy, and T. V. Raman. Efficient multiway radix search trees. *Information Processing Letters*, 60:115–120, 1996.

[27] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 2nd edition, 1990.

[28] G. E. Forsythe. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[29] C. W. Fraser. copt, 1980. `ftp://ftp.cs.princeton.edu/pub/lcc/contrib/copt.shar`.

[30] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Department of Computer Science, Princeton University, 1990.

[31] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1995.

[32] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[33] C. W. Fraser and A. L. Wendt. Integrating code generation and optimization. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 242–248, Palo Alto, CA, June 1986. ACM.

[34] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–84, Atlanta, GA, June 1988. ACM.

[35] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, November 1974.

[36] L. George and A. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[37] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[38] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM.

[39] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, The Netherlands, June 1997. ACM.

[40] S. P. Harbison and G. L. Steele, Jr. *C, A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, 4th edition, 1995.

[41] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–335, Orlando, FL, June 1994. ACM.

[42] W.-C. Hsu, C. N. Fischer, and J. R. Goodman. On the minimization of loads and stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[43] N. D. Jones, C. K. Gomarde, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, 1993.

[44] R. Kelsey, W. Clinger, J. Rees (editors), et al. *Revised^5 Report on the Algorithmic Language Scheme*. MIT Project on Mathematics and Computation, February 1998.

[45] D. Keppel. A portable interface for on-the-fly instruction space modification. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, Santa Clara, CA, April 1991. ACM.

[46] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington, 1991.

[47] D. Keppel, S. J. Eggers, and R. R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, 1993.

[48] T. B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, PA, May 1996. ACM.

[49] J. Larus and E. Schnarr. EEL: Machine independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, CA, June 1995. ACM.

[50] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Indiana University Computer Science Department, 1997.

[51] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, May 1996. ACM.

[52] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[53] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report 698, Courant Institute, New York University, July 1995.

[54] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[55] R. Pike, B. N. Locanthi, and J. F. Reiser. Hardware/software trade-offs for bitmap graphics on the Blit. *Software Practice & Experience*, 15(2):131–151, February 1985.

[56] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–121, Las Vegas, NV, June 1997. ACM.

[57] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 1999. (To appear).

[58] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 1999. (To appear).

[59] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[60] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications. Technical report, University of Arizona, 1997.

[61] C. Pu, T. Autry, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commerical operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995. ACM.

[62] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Comput. Syst.*, 1(1):11–32, 1988.

[63] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, B. Bershad, H. Levy, , and J. B. Chen. Etch, an instrumentation and optimization tool for win32 programs. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[64] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[65] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[66] M. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.

[67] SPARC International, Englewood Cliffs, NJ. *SPARC Architecture Manual Version 9*, 1994.

[68] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994. ACM.

[69] R. Stallman. Using and porting GCC, 1990.

[70] G. L. Steele, Jr. *Common LISP*. Digital Press, Burlington, MA, 2nd edition, 1990.

[71] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, January 1982.

[72] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[73] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998. ACM.

[74] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Durham, NC, 1994. ACM.

[75] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly, Sebastopol, CA, October 1996.

[76] A. Wendt. Fast code generation using automatically-generated decision trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 9–15, White Plains, NY, June 1990. ACM.

[77] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, Philadelphia, PA, May 1996. ACM.

[78] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles*, St. Malo, France, November 1997. ACM.