Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-96-21

1996-01-01

Continuous Compilation for Software Development and Mobile Computing

Michael P. Plezbert

Software developers typically must choose between interpreted and compiled environments for their programming activities. However, the current trends toward mobile computing and platform independence suggest moving to a new continuous compilation paradigm that integrates the advantages of each environment. Movement in this direction can already be seen in the development of Sun Microsystems' Java environment. The resulting continuous compiler operates not as a prelude to, but rather in tandem with, program execution. In this thesis we present the results of experiments that compare the performance of the continuous compilation model with a more traditional model and show that a... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Plezbert, Michael P., "Continuous Compilation for Software Development and Mobile Computing" Report Number: WUCS-96-21 (1996). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/412

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/412

Continuous Compilation for Software Development and Mobile Computing

Michael P. Plezbert

Complete Abstract:

Software developers typically must choose between interpreted and compiled environments for their programming activities. However, the current trends toward mobile computing and platform independence suggest moving to a new continuous compilation paradigm that integrates the advantages of each environment. Movement in this direction can already be seen in the development of Sun Microsystems' Java environment. The resulting continuous compiler operates not as a prelude to, but rather in tandem with, program execution. In this thesis we present the results of experiments that compare the performance of the continuous compilation model with a more traditional model and show that a performance increase can be obtained by moving to continuous compilation. In addition, we examine the effects of various compilation strategies on the performance of the continuous sampler.

Continuous Compilation for Software Development and Mobile Computing

Michael P. Plezbert

WUCS-96-21

July 1996

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

A thesis presented to the Sever Institute of Washington University in partial fulfillment of the requirements for the degree Master of Science.

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

CONTINUOUS COMPILATION FOR SOFTWARE DEVELOPMENT AND MOBILE COMPUTING

by

Michael P. Plezbert

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 1996

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

CONTINUOUS COMPILATION FOR SOFTWARE DEVELOPMENT AND MOBILE COMPUTING

by Michael P. Plezbert

ADVISOR: Dr. Ron K. Cytron

May, 1996
Saint Louis, Missouri

Software developers typically must choose between interpreted and compiled environments for their programming activities. However, the current trends toward mobile computing and platform independence suggest moving to a new *continuous compilation* paradigm that integrates the advantages of each environment. Movement in this direction can already be seen in the development of Sun Microsystems' Java environment.

The resulting *continuous compiler* operates not as a prelude to, but rather in tandem with, program execution. In this thesis we present the results of experiments that compare the performance of the continuous compilation model with a more traditional model and show that a performance increase can be obtained by moving to continuous compilation. In addition, we examine the effects of various compilation strategies on the performance of the continuous compiler.

Contents

List of Tables iv

List of Figures vi

Acknowledgments vii

1 Introduction **1**

2 Related Work **7**

2.1 Interpreters and Compilers 7

2.2 Mobile Computing 9

2.3 Program Modeling 10

3 Continuous Compiler Model **12**

3.1 Comparison to Traditional Methods 13

3.2 Compilation Strategies 14

3.3 Replacement Strategies 16

4 The Simulator **19**

4.1 Simulator Input 20

4.1.1 Setup 20

4.1.2 Behavior 22

4.2 Simulator Implementation 22

4.3 Simulator Output 23

4.4 Simulation of Real Programs 25

5 ProGenitor **26**

5.1 Input Parameters 28

5.2 Probability Distributions 30

5.3 Program Generation 31

6 Experiments **36**

6.1 Source Programs 36

6.2 Continuous Compilation vs. Traditional Compilation 37

6.3 Continuous Compilation vs. Traditional Interpretation 42

6.4 Comparing Replacement Strategies 44

6.5 Comparing Compilation Selection Strategies 48

6.6 Effects of Library/User Ratio on Performance 50

6.7 Effects of Density of Functions per Source File 54

6.8 Various Interpretation Penalties 56

6.9 Summary 57

7 Conclusion **59**

Appendix A Sample Simulator Input and Output Files **62**

A.1 Sample Setup File 62

A.2 Sample Behavior File 66

A.3 Sample Order File 67

A.4 Sample Output File 68

Appendix B Sample ProGenitor Input File **72**

References **73**

Vita **75**

List of Tables

1.1 Compilation Times With and Without Optimization 3

5.1 Example ProGenitor Input 35

6.1 Real Programs Studied 37
6.2 Cold-start with Traditional Compilation System 38
6.3 Comparison between Traditional and Continuous Replace-at-Call . . 38
6.4 Comparison between Traditional and Continuous Replace-Preemptive 39
6.5 Single-processor performance of render 40
6.6 Single-processor performance of pico 41
6.7 Single-processor performance of ghostview 42
6.8 Continuous Compilation vs. Interpretation 43
6.9 Comparison of Replacement Strategies with Ghostview 44
6.10 Pico (Replace-at-Call) 46
6.11 Pico (Replace-Preemptive) 46
6.12 Render (Replace-at-Call) 47
6.13 Render (Replace-Preemptive) 48
6.14 Ghostview (Replace-Preemptive) 48
6.15 Ghostview (Replace-at-Call) 50
6.16 Common User File Parameters of Programs Generated to Examine the Effects of the Library/User Ratio on Performance 51
6.17 Library File Parameters of Programs Generated to Examine the Effects of the Library/User Ratio on Performance 52
6.18 Chance of Calling Another User Function 52
6.19 Variation of Library/User Ratio 53
6.20 User File Parameters of the Many-Functions-per-File Program 54
6.21 Library File Parameters of the Many-Functions-per-File Program . . 55

6.22 User File Parameters of the One-Function-per-File Program 55
6.23 Variation of Functions per Source File 56
6.24 Performance with Various Interpretation Penalties 57

List of Figures

1.1 The Traditional Development Cycle 2
1.2 The Continuous Compilation Cycle 4

3.1 The Continuous Compiler 12

4.1 Overview of the Simulator 19
4.2 Sample Setup File 20
4.3 Sample Behavior File 21
4.4 Simulator Pseudocode 22

Acknowledgments

I would like to extend my sincerest thanks to Dr. Ron K. Cytron, without whose tremendous help, dedication, and patience, this would never have been written.

I would also like to thank Dr. Takayuki Dan Kimura and Dr. Mark A. Franklin for agreeing to be on my defense committee at such short notice and at such a busy time of the year.

I would like to thank Noah Treuhaft for his work on the eer program and his other invaluable contributions to this project.

And I would like to thank Steve Scott for answering my interminable LaTeX questions and for putting up with me during this writing.

This research was supported in part by the National Science Foundation under grant CCR-94-02883.

Michael P. Plezbert

Washington University in Saint Louis
May 1996

Chapter 1

Introduction

Traditionally, the domains of interpretation and compilation have remained fairly separate. In part this is because languages often lend themselves to one paradigm or the other. For example, languages like LISP and Smalltalk have traditionally been interpreted while those like C and Pascal are usually compiled. However, neither paradigm is inviolate. Compilers exist for many LISP variants [8] and interpreters exist for both C and Pascal [4].

Interpretation is often used for fields such as rapid prototyping, due to the interactive nature of development and the quick response to change. Compilation is generally used for production-level applications due to the performance advantage of executing a program in native-code form. In industry, projects are often started in an interpreted environment to take advantage of the relative ease of development and then later moved to a compiled environment, often at high cost to the company.

In this thesis, we examine the feasibility and benefits of a continuous compilation system by examining the extent to which program interpretation and compilation can be overlapped and intermixed in a system that continuously translates programs into native-code form throughout program execution. We will present experimental data that indicates a significant performance gain by the continuous compilation model over more traditional approaches.

In a typical software development environment, the sequence of steps Edit $\rightarrow$ Make $\rightarrow$ Execute, as depicted in Figure 1.1, is repeated throughout program development, where the "Make" step involves

1. compilation of those files that have been changed,

2. compilation of those files that (ultimately) depend on files that have changed, and

3. linking of separately compiled objects into an executable image.

Execution then proceeds until the next "bug" is found, at which point execution is stopped and the cycle repeats from the "Edit" step.

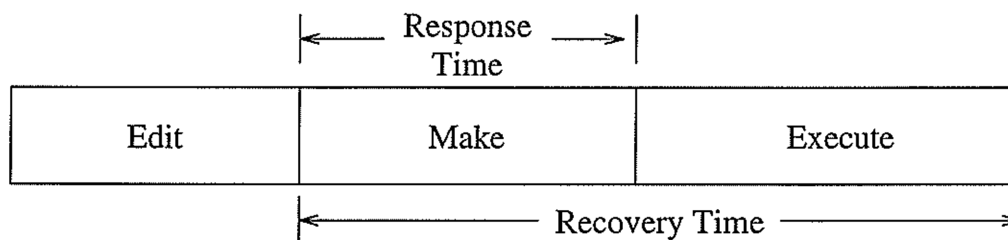

Figure 1.1: The Traditional Development Cycle

We are interested in the *recovery time* of the program development cycle, defined as the time interval occupied by the "Make" and "Execute" steps just up to the point of arresting program execution. During this time interval, the user has essentially abdicated control of the development cycle to the computer. In particular, shortening this interval improves life for the program developer, because the effects of a program edit can be seen more quickly. For example, if recovery time takes a minute or more, then there is a good possibility that the program developer has switched to a different task, such as reading mail or solving other problems. While recovery time can be shortened by purchase of a faster computer, we investigate an alternative in this thesis, namely adopting a better paradigm for program compilation.

Until recently, the practice of well-accepted software design principles (such as modularity and encapsulation) generally resulted in source files that took relatively little time to compile: the "Make" step occupied a relatively insignificant portion of the recovery time. However, developments in language and compiler design have conspired to make compilation times grow:

- The use of features such as templates in C++ has significantly increased the amount of work performed at compile-time on behalf of a single compilation unit.

- Program optimizations can significantly increase compilation times. Tests using Sun's cc compiler show quadratic growth in compile-time as a function of

Table 1.1: Compilation Times With and Without Optimization

Lines of Code	CPU Time (seconds)	
	No Opt.	Opt.
20	0.5	0.6
108	0.5	1.3
108	0.6	1.4
247	0.6	3.6
251	0.8	3.3
493	1.0	17.8
537	0.9	12.0
2336	2.0	12.5
2667	2.9	62.0

program size, with programs of 2400 lines taking 21 times longer to compile with optimization enabled; compile times of up to a minute on such programs are not uncommon. This trend is likely to become worse as interprocedural information is incorporated into program optimization.

While efforts at improving compiler performance for processing new language constructs and for aggressive program optimization can only be applauded, we propose in this thesis a complementary approach:

> *The compiler should effectively* continuously *transform a program from an interpreted to a fully optimized form.*

Consider the scenario depicted in Figure 1.2. Program execution begins with interpretation, with compilation proceeding concurrently with execution. As execution continues, the interpreted source code is gradually replaced with natively-executable code. Performance increases until, eventually, the entire program has been translated to a fully native form.

In this thesis we concentrate on the exploring the first step: the transition from interpreted to native code. But it does not have to stop there. The continuous compiler can proceed with improving the code, starting with interprocedural optimization and moving on to intraprocedural optimizations, until a fully optimized form has been produced.

This approach provides an immediate response time, as with interpretation. However, it also gives performance that approaches and, with more time available to spent on optimization, possibly surpasses that of executing native code.

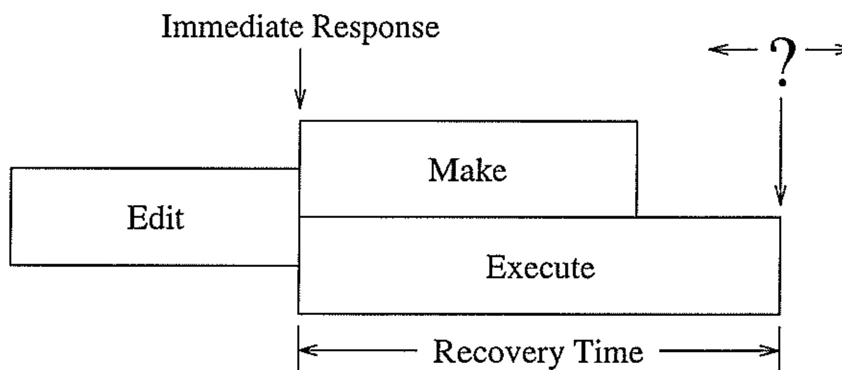

Figure 1.2: The Continuous Compilation Cycle

The reduction of response time in software development environments is becoming more important as the development environments become more interactive. Visual programming, in particular, is one area in which response time is of vital importance. Typically, visual programming environments use interpretation during the development stage with compilation occurring only when requested by the developer, usually when the code is considered "complete." Such an environment could benefit greatly by the introduction of a continuous compilation model.

The possible applications of this continuous compilation paradigm is by no mean limited to software development. We see an even greater impetus for migrating toward the continuous compilation model coming from the expanding field of *moving-target* computing. By the term "moving-target computing" we are referring to two disparate but parallel trends in computing, namely, *mobile computing* and *platform independence*.

Mobile computing simply refers to the use of workstations or terminals that are physically mobile. Until recently, these mobile computers were mostly independent units that connected to other computers or networks only at fixed places where wired communication media was available, and only to exchange relatively small amounts of data. However, in spite of the lower bandwidth provided by current wireless technology, wireless data communication is becoming more prevalent. At the same time, we are seeing a shift to a more connected, networked model of computing.

With this shift in mobile computing comes the need for rapid transmission of programs from a remote server to the mobile platform. Interpretable forms, such

as bytecodes, are generally more compact than directly executable forms and can therefore be transmitted much more quickly. For example, a desk calculator implemented by recursive-descent parsing takes approximately 20 Kbytes in native Sparc executable form (after stripping) while a bytecode representation of this same program takes only 3 Kbytes. Regardless of the transmission media used, it takes over six times as long to transmit the executable form. With current transmission rates this can mean a difference of over 15 seconds in transmission time.

The drawback to using interpretation is again execution performance. We generally assume that, interpreting a program incurs a penalty factor of 10 over executing the program in native-code form [19]. Reducing the time required to receive a program to one-sixth of its previous value may not provide any benefit if it takes ten times longer for the program to execute. The use of the continuous compilation model can provide the benefits of the short transmission time without sacrificing performance.

The second component of the moving-target paradigm is platform independence. Historically, making an application available on multiple platforms often required maintaining a separate source for each platform. At the very least, separate binaries are needed. However, it is becoming increasingly easy to deliver applications in a target-independent manner. It is becoming feasible for vendors to maintain a single source for an application, distribute the application in machine-independent form (such as bytecodes), and rely on the remote user's platform to generate the native executable code. The explosive popularity of Sun Microsystems' Java [18] language gives testimony to the desire that exists for this computing model.

For machine-independent application distribution, there is no choice but to deliver the application in platform-independent form. The continuous compilation model examined in this thesis allows interpretation of this form to begin immediately, thereby reducing perceived response time, while also reducing the performance overhead imposed by interpretation.

This thesis presents the results of experiments performed using a simulator of a continuous compilation system to determine the possible performance benefits or detriment of such a system. We also examine various system design issues in an attempt to discover their effect on the performance of the system. In particular, we examine the following questions:

1. How does the recovery time of the continuous compilation system compare with that of a tradition compilation system?

2. What effect does the order of compilation of files have on the performance of the continuous compilation system?

3. How great an effect does various methods of replacing source code with native code have on the continuous compilation system?

4. To what extent do various program characteristics affect the performance of the program under the continuous compilation system?

Chapter 2 presents some background information and related work. Chapter 3 describes the continuous compilation model in more detail. Chapter 4 describes the simulator we built to perform the timing experiments. Chapter 5 describes a utility we built to generate test cases. Chapter 6 describes the various experiments we carried out to explore the performance of the continuous compilation model. In Chapter 7 we give some concluding remarks and discuss future work.

Chapter 2

Related Work

2.1 Interpreters and Compilers

The idea of combining interpreted and native-code text could be traced back to early work by Ershov and others [2], and continues to this day, currently in the active field of partial evaluation.

For languages such as C, which have long been implemented primarily by compilation to native code (an exception is the si system [4]), interpreters are experiencing somewhat of a comeback. For one explanation, consider that Proebsting and others are making interpreters much more efficient [19], by combining compiled with interpreted text. However, even optimized interpretation cannot compete with the speed of execution of native code. Through the use of superoperators, Proebsting has achieved interpretation speeds only 3–9 times slower than executing unoptimized native code. Without superoperators that value jumps up to 8–16 [19].

Interpretation also has several other features to recommend it [16]. Foremost is the interactive nature of interpretation. Interpreted environments can respond immediately to change, without the impediment of lengthy compile times. In interactive applications this can be extremely important.

A couple of other advantages of interpretation that are sometimes overlooked are the possibilities of platform independence and smaller distribution file size. Native executables are by nature bound to the specific platform for which they were compiled. In contrast, interpretable code, whether a high-level source language or an intermediate bytecode representation, can be designed to be independent of any particular machine. In addition, an efficient bytecode representation can be much smaller than the equivalent native code.

Support for interactive, source-level debugging is also more innate with interpretation. Furnishing such support in a compiled environment is a much more arduous task, required the preservation of a mapping from the generated native code to the original source code.

Finally, when comparing the performance of an interpreted environment to that of a compiled environment, the comparison is usually between the speed of interpretation and the speed of execution of native code, without taking into account the time spent producing the native code. This may be an acceptable comparison in some circumstances, but in other situations, such as with mobile computing and software development as described in Chapter 1, the time spent in translation is of vital importance.

In other related work, adaptive optimization has been considered for the Self language, with the idea of using a fast compiler to generate initial code while an optimizing compiler recompiles heavily used parts [9]. Kastens has considered how to generate interpreters automatically from compiler specifications [12].

In comparing our work with the above, note that we are not as interested in maintaining two forms of language processors (a compiler and interpreter) as we are in examining the role of a common, machine-independent representation for applications. While speeding up interpretation is always a win, there is inevitably a performance gap between interpretation and heavily optimized target-machine code.

Perhaps closest in spirit to the paradigm we propose are the new "just-in-time" (JIT) compilers that are being developed for the Java language [11]. However, these compilers differ from the continuous compilation paradigm that we are proposing in several important aspects.

First of all, the JIT compiler does not work in tandem with the interpreter; rather it is meant to *replace* the interpreter. As classes are loaded into the runtime virtual machine (Java is an object-oriented language), the method pointers in the virtual method table, which in the interpreted version of Java would point to the bytecode of the corresponding methods, are replaced with pointers to the JIT compiler. Then, the first time each method is called, the JIT compiler is invoked to compile the method. The pointer in the virtual method table is then patched to point to the native-code version of the method so that future calls to the method will jump to the native-code. With the JIT compiler in place, no interpretation of the methods ever takes place.

While this is a valid, and valuable, technique, it differs significantly from the continuous compilation model. With the current implementations of the JIT compilers, compilation is only done on demand (*i.e.*, upon the first call to each method) and execution of the method must wait until compilation is complete. As long as the average size of an individual method tends to be rather small, the wait is not likely to be noticeable. However, it means that compilation speed is of great importance. Consequently, time cannot be spent on optimizing the generated code.

In addition, using a compile-on-demand model in an interactive environment (which is where Java seems to be aimed) leads to inefficiency. In environments characterized by user interaction, the CPU often remains idle, or runs well below 100% capacity, for much of the time while waiting for user input. If some sort of "precompiling" (analogous to prefetching in memory management or disk caching) model is used in the compile-on-demand system, then this idle time could be put to use compiling methods that are likely to be called in the near future. However, the current JIT compiler implementations do not seem to support precompiling (although they almost certainly will in the future).

The JIT compilers are closely tied to the Java language and run-time system. They do *not* produce independently executable object files. They are meant to only speed up execution of class methods called by the Java run-time virtual machine.

While the continuous compilation paradigm can be applied to Java, it is a considerably more generic approach. It is meant to be applicable to any situation in which traditional compilation or interpretation is used.

2.2 Mobile Computing

Even though wireless data transmission rates are improving, and the trend is expected to continue, it is still instructive to look at the various options currently available for wireless data transmission and the data throughput provided by each.

There are currently five options commonly available for wireless data transmission. These five options are described below [1, 10, 13, 17].

Spread-spectrum packet radio Wireless local area networks implemented using spread-spectrum packet radio are available. They have a raw data rate of up to 5.7 Mbps, but have a limited range, with a typical radius outside the LAN of 200 to 1000 feet. Bridges using microwave radios are available, but with data rates of only about 1.6 Mbps.

The Ricochet Wireless Network Service [17] also uses spread-spectrum packet radio, but uses a different network topology to provide greater area coverage. However, the raw data rate for this network is 100 Kbps, with throughput in the 9.8–28.8 Kbps range.

Private packet radio This technology is currently used by two companies (Ardis and RAM Mobile Data) to implement wide area coverage of most of the United States. Transmission rates are in the range of 4.8–19.2 Kbps.

Circuit-switched cellular networks This is the option offered by most cellular telephone companies. Transmission rates are in the 2.0–20.0 Kbps range.

Cellular digital packet data This alternative is an attempt to lower the cost of cellular data transmission by sending the data during the idle time between cellular voice calls. Throughput is around 9.8 Kbps with a 1 to 5 second response time.

Satellite All current system use geosynchronous satellites. This option is rather expensive and currently only supplies a two-way digital transmission rate of 2400 bps per connection.

Other systems using networks of low-earth-orbit satellites have been proposed by various companies (including Motorola and Qualcomm), but these have yet to materialize.

2.3 Program Modeling

In Chapter 5, we describe a system we built to generate profile information for synthesized programs. Such an endeavor would undoubtedly be helped by the examination of the properties of real programs. Unfortunately, such information is in short supply.

The most famous study of the characteristics of real programs is the one performed by Donald Knuth on FORTRAN programs back in 1971 [14]. While this study was an invaluable asset to the understanding of the characteristics of actual programs, it is sorely out of date. Blindly assuming that the profiles of programs written in modern modular or object-oriented programming languages would match those of programs written in early FORTRAN seems unconscionable. The very nature of modern programming languages differs vastly from that of early FORTRAN. The advent of modular languages, object-oriented languages, functional languages,

etc., has moved us ever farther afield from those beginnings. Surprisingly, though, there does not seem to be a modern equivalent to Knuth's study.

The characterization of real programs, while an important and desperately needed pursuit, is outside the scope of this work. Lacking experimental data for modeling programs, we opted instead to create a tool to study properties of interest. Perhaps a deeper study could be conducted at a future date.

Chapter 3

Continuous Compiler Model

The continuous compilation model proposed is shown in Figure 3.1. The Interpreter module is responsible for executing the program while the Compiler module is responsible for translating the source code into a natively executable format. The Interpreter and Compiler modules run concurrently, either physically or logically. Ideally, a dual-processor system would be used, with the Interpreter and Compiler modules running on separate processors; however, this is not a requirement. In Section 6.2 we discuss some of the issues involved in using a continuous compiler model on a dual-processor system as opposed to a traditional compilation model on a faster single processor.

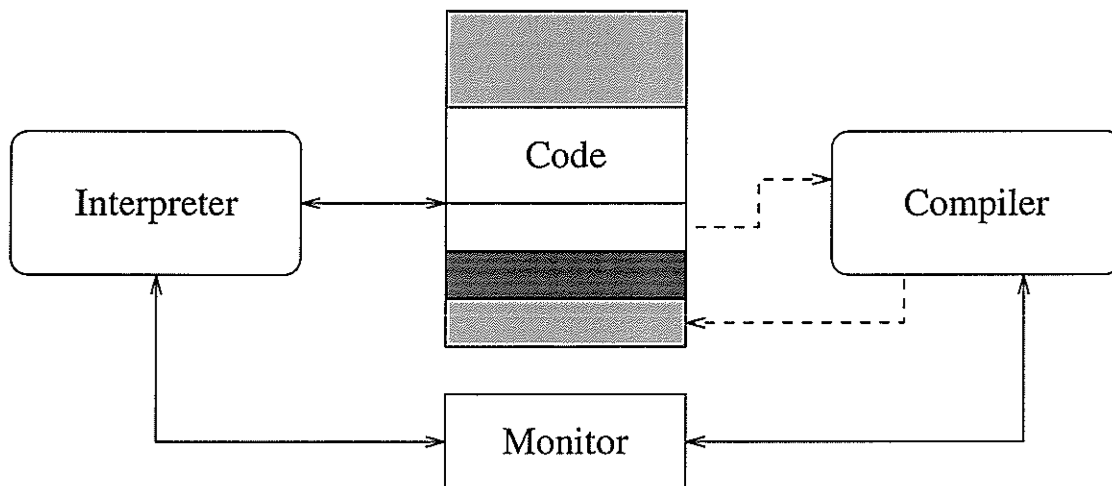

Figure 3.1: The Continuous Compiler

The pieces of this model function as follows:

Code The Code contains a mixture of interpreted (source) and native-code versions of the procedures of a program. Upon the initial start-up, all of the "user"

code is assumed to be in source form; library routines are assumed always to be available in native-code format. As the program is executed, the Compiler module generates native-code translations of the interpreted procedures. Not until a procedure has been fully translated to native code does the native-code version become available to the Interpreter.

Compiler The Compiler module translates the program's procedures from interpreted (source) form to native executable code. As the translation of each procedure is completed, the native-code version is made available to the Interpreter.

Interpreter The Interpreter module is responsible for the actual execution of the program. It starts by interpreting the source code, making jumps to the native-code version of procedures as they become available. We examined two replacement strategies: replace-at-call will jump to the native code version of a function only at the time that function is called; replace-preemptive will jump to the native code as soon as it becomes available, even while the function is active. Section 3.3 discusses the replacement strategies in more detail.

Monitor The two modules (the Compiler and the Interpreter) communicate through the Monitor structure. This is a shared data structure that contains information needed by both the Compiler and Interpreter. For example, some of the compilation strategies described in the next section require that the Compiler module have access to profile data about the program being executed. The Interpreter can gather this data while executing the program and then make it available to the Compiler by storing it in the Monitor structure.

While this thesis focuses on the initial transition from source code to native code, it is important to remember that compilation need not end when all of the source code has been translated to native form. The Compiler can continue to perform optimization while the program is executing.

3.1 Comparison to Traditional Methods

The continuous compilation model is an attempt to combine the benefits of the traditional interpretation and compilation models. With the continuous compiler, we get

all of the benefits of interpretation without sacrificing performance. These benefits include

1. immediate response to change in interactive environments,
2. platform independent format, and
3. smaller distribution file size (when comparing bytecodes to native executables).

In addition, with more time to spend on optimization, we have the possibility of ending up with a highly optimized, fully native-code version of a program that performs better than the binary that could be produced by a traditional compiler in the amount of time that we are willing to wait.

However, the question remains: how well does the continuous compiler perform in comparison to the traditional models? This thesis is an attempt to answer this question before paying the expense of developing such a system.

In conducting our study, we also examine various aspects of the continuous compilation model to determine their effect on the performance of the system. The next two sections describe the properties of the continuous compilation model that were thought most likely to affect performance.

3.2 Compilation Strategies

As we discovered in our experiments, the execution time of a program using the continuous compiler is strongly affected by the order in which the routines are translated into native code. Seven different compilation strategies were examined. Each is described below.

Three of the strategies, shortest-first, largest-first, and random, are static and use fixed information about the routines to determine the translation order before any translation begins. Two of the strategies, longest-so-far and most-frequently-executed-so-far, are dynamic and use information gleaned from monitoring the execution of the program to determine the order of translation; the next module to translate is picked at the time of translation.

The remaining two strategies, longest-overall and most-frequently-executed-overall, are hybrid. The order of translation is determined upon start-up, before any translation begins; however, they require knowledge about the execution of the program, gained from a previous run.

The three static strategies have the advantage of low overhead; the order of translation is fully determined before translation begins and is based only on easily determined static properties of the source modules. The two dynamic strategies have the disadvantage of a higher overhead; more processing is required at each step to determine the next module to translate. The order of translation cannot be fully determined at start-up, and in fact changes as the program is executed. The final two strategies have low overhead, but they have the disadvantage of requiring information from a previous run of the program, which is not always available.

Shortest-first The shortest-first strategy is based on the size of the source modules. The modules are simply sorted from smallest to largest and then translated in that order.

Largest-first The largest-first strategy is the opposite of the shortest-first strategy: the modules are translated in order from largest to smallest.

Random In the random strategy, the source modules are simply translated in random order. The main purpose of this strategy is to emulate the nondeterminism that might be caused by the transmission of source files over a network, when the files might not arrive in the order sent.

Most-frequently-executed-so-far In this strategy, the Monitor module keeps track of how often each routine is called. When the next source module to translate needs to be picked, the Compiler picks the untranslated routine that has accumulated the most calls up to that point in the execution of the program.

Longest-so-far This strategy is similar to most-frequently-executed-so-far, except that the next module for translation is chosen based on the current accumulated time spent in executing the routines in each module rather than a call count. The accumulated time for a function does not include time spent in other functions on behalf of this function.

Most-frequently-executed-overall This strategy uses information gained during a previous execution of the program to determine which routines were called the most over the entire execution. The untranslated source module which received the most calls during the previous execution is the one that is chosen for translation.

Longest-overall This strategy is the same as the most-frequently-executed-overall, except that the next module chosen for translation is the one that accumulated the most run-time during the previous execution instead of the one with the most calls.

An important point to note is that compilation takes place on a file-based level of granularity, not on the routine level. If each source file contains only one routine then these are of course identical, but that is not often the case. Section 6.7 describes some experiments run to determine how strong an effect the density of routines per source file has on the performance of the continuous compiler.

When choosing which file to translate next, our system looks at the statistics for the file as a whole, not at the individual routines. For example, the most-frequently-executed-so-far strategy picks the file with the largest number of calls made to *all* of the routines defined in each file, which is not necessarily the file which contains the single most called routine. All of our compilation strategies work in this manner. Picking the files based on individual routines is, of course, also a valid method, but was not examined in these experiments.

The shortest-first strategy deserves closer attention. If the assumption can be made that the time required for compilation of a file is a monotonically nondecreasing function of file size, then the shortest-first strategy maximizes the throughput (files translated per minute) of the Compiler module. It is analogous to the shortest processing time scheduling algorithm [6]. However, it is important to note that maximizing the throughput of the Compiler module is not the same as maximizing the performance of the continuous compilation system as a whole. In fact, we found through our experiments (see Chapter 6) that, while the shortest-first strategy usually gave fairly good results, it did *not* usually give the best results.

3.3 Replacement Strategies

The strategy used by the Interpreter in replacing interpreted source code with translated native code can also have a profound effect on execution time. In this study, two strategies were considered: replace-at-call and replace-preemptive.

Replace-at-call In this strategy, the switch from interpreting source code to running native code occurs only at function calls. When a function call occurs, a jump is made to the native code, if it is available; otherwise, the function is interpreted.

And when a function returns, execution of the calling function continues in whatever mode it was in prior to when the call occurred. That is, if routine A, which is being interpreted, calls routine B, which is available in native-code form, then when routine B exits and control is returned to routine A, A continues to be interpreted, even if the native code for A became available during execution of B. In other words, function invocations active on the stack are not replaced. However, if a recursive call is made to a routine that is now available in native form, then the native-code version of the routine is used for the new call, regardless of whether or not an interpreted instantiation of the function is on the call stack.

Replace-preemptive In this strategy, execution of a routine is switched from interpreting source code to running native code as soon as the native code is available, even if it is while the routine is being executed. Also, if the native-code version of a function is available when that function is popped off the activation stack to resume execution, then a jump is made to the native code, even if the function was originally being interpreted when it was pushed on the stack.

The replace-at-call strategy has the advantage of low overhead and relative ease of implementation. The only processing required by the Interpreter to implement replace-at-call occurs at function calls. When the Interpreter reaches a call instruction, it simply checks the program state information to see if the function is available in native-code form. If it is available, it simply jumps to the compiled code; if not, it continues interpreting. The processing required by the Compiler module is also fairly minimal. When the Compiler translates a function that contains calls to functions that have not yet been translated, the functions calls are implemented as jumps to the Interpreter.

The replace-preemptive strategy is more difficult to implement and has a higher overhead, but it also has the possibility of greatly improving performance. In addition to the processing required by replace-at-call, the replace-preemptive strategy requires that the Interpreter check for newly available native code whenever it removes an activation record from the stack to continue execution. The Interpreter would also need to know where to jump in the native-code version of the function to continue execution where it left off interpreting. In a fully replace-preemptive system, the Compiler would need to be able to interrupt the Interpreter if it happens to finish translating

the function that the Interpreter is currently processing so that the Interpreter can jump to the compiled code. However, the cost of implementation would most likely be prohibitive. A "mostly" replace-preemptive system, which did not include the interruption of the Interpreter, could be implemented instead. Jumps to compiled code would only occur at function calls and returns, but as function calls occur with a fairly large frequency in most well-structured programs, performance is not likely to be impacted much. The main advantage over the replace-at-call method still exists; we do not have to wait for a particular function to complete before replacing it with native code. The replacement can occur when execution of the function resumes after returning from a function call. Nevertheless, for the purposes of these experiments our simulator emulates the fully replace-preemptive method.

Chapter 4

The Simulator

For the purpose of obtaining empirical data on the possible benefits of our continuous compilation model, we constructed a simulator. This simulator accepts as input information about a particular program and a trace of its execution. It then *simulates* execution of the program using the continuous compilation model described in Chapter 3 and returns performance data. The simulator could also be used to do just-in-time compilation studies, although they were not done as part of this thesis.

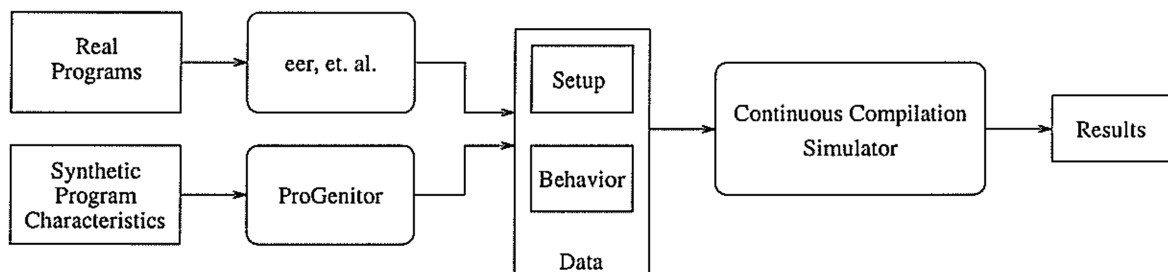

Figure 4.1: Overview of the Simulator

As shown in Figure 4.1, the simulator is designed to accept information about either real or synthetic programs. Information about the program structure is contained in the *setup* files while the *behavior* file contains a trace of the program execution. Each of these is described in detail in the next section.

A description of the ProGenitor program can be found in Chapter 5. We discuss how the requisite information about real programs was obtained later in this chapter.

4.1 Simulator Input

4.1.1 Setup

The setup file contains static data about the program. It consists of three sections:

1. Source and object file dependence information.
2. Function→File mapping.
3. File sizes and compilation times.

The source and object file dependence information is similar to that found in the "makefiles" used by the utility `make`. It is used by the Compiler module of the continuous compiler in the same way such information is used by a traditional compiler: to determine if some files must be compiled before others and to determine which files need recompilation after another file has been edited.

The function→file mapping allows the continuous compiler to know what new functions are available in native code after a file has been compiled. It is also used to help determine which file to compile next in some of the compilation strategies defined in Section 3.2. The information is identical to that produced by the `ctags` utility.

```
MakeStuff
main:  main.o smpl.o parsecmd.o
main.o:  main.c complex.h cmdstr.h
smpl.o:  smpl.c complex.h
parsecmd.o:  cmdstr.h

CtagStuff
smpl      smpl.c      / optional comment /
readcmd   parsecmd.c  / optional comment /
getopt    parsecmd.c  / optional comment /

CompileStuff
main.c      57     900
parsecmd.c  28     700
smpl.c      14     400
```

Figure 4.2: Sample Setup File

The file sizes and compilation times are used by our simulator both to simulate compilation and to implement the smallest-first and largest-first compilation strategies.

Our simulator implements the following five compilation strategies intrinsically: smallest-first, largest-first, random, most-frequently-executed-so-far, and longest-so-far. However, it is also possible to specify any desired compilation order by providing an ordered list of source files to the simulator as part of the setup. The most-frequently-executed-overall and longest-overall strategies were implemented in this manner.

Figure 4.2 shows a small sample setup file. The section under the "MakeStuff" heading specifies the file dependencies in exactly the same format as that used by the `make` utility. The "CtagStuff" section lists all of the functions defined in the source files along with the file in which each function is defined. The "CompileStuff" section lists all of the source files along with the number of lines in each file and the time, in milliseconds, required to compile the file. [1]

```
I  473   _start
I  670   _init
O  672
I  674   __do_global_ctors_aux
O  678
I  681   main
I  3649  readcmd
O  3652
I  3654  readcmd
I  3657  getopt
O  3664
O  3667
I  3703  smpl
O  3752
O  3769
I  3774  exit
I  3861  __do_global_dtors
O  3864
I  3867  _fini
O  3869
O  4025
```

Figure 4.3: Sample Behavior File

[1]Actual compilation times were measured in seconds. Units of milliseconds are used for compatibility with the setup file.

4.1.2 Behavior

Figure 4.3 shows a (very) small sample behavior file. Every line that begins with an "I" corresponds to a function entry and every line that begins with an "O" corresponds to a function exit. The number on each line is the time, in *microseconds*, from the beginning of program execution that the corresponding entry or exit occurred. For function entries, the function name is given. The names are not needed for function exits because a stack based implementation is assumed.

This program trace gives enough information for us to simulate executing the program with our continuous compilation model. How this simulation is accomplished is described in the next section.

4.2 Simulator Implementation

The simulator does not actually execute the program, nor translate code, but rather proceeds through the behavior trace file updating state information. Standard discrete, event-based simulation techniques are used.

Figure 4.4 describes the basic operation of the simulator in pseudocode. The

```
While Interpreter event-list is not empty do
    Let I←Head(Interpreter.event-list)
    Let C←Head(Compiler.event-list)
    If Time(I)<Time(C) then
        Do event I.
        Remove(Head(Interpreter.event-list))
    Else
        Do event C.
        If replacement is preemptive then
            Update Interpreter state.
        End-if
        Remove(Head(Interpreter.event-list))
    End-if
End-do
```

Figure 4.4: Simulator Pseudocode

Interpreter and Compiler modules are modeled as separate object, each with its own event-list. The event-list for the Interpreter object is derived from the behavior trace file. Each function entry and exit is an Interpreter event.

A Compiler event is the completion of compilation of a file. For two of the compilation strategies—most-frequently-executed-so-far and longest-so-far—the entire Compiler event-list is not known at any given time. However, the event at the head of the list is always known.

Similarly, the entire Interpreter event-list is not know at any given time. It is too dependent on when functions are compiled for it to be determined in advance. However, the event at the head of the list can again be calculated at any given time.

A shared data structure (the "Monitor" in Figure 3.1) contains the compilation status of each function, *i.e.*, whether the function is available in native code or not. Performing a Compiler event (the `Do event C` line in Figure 4.4) simply consists of updating this data structure by changing the status of the appropriate functions to "compiled." For the most-frequently-executed-so-far and longest-so-far compilation strategies, the next file to compile is chosen at this time and the event is placed in the Compiler event-list.

Performing an Interpreter event (the `Do event I` line Figure 4.4) consists of updating the statistics being kept on each function. These statistics consist of the number of calls made to each function, the amount of time spent interpreting each function, and the amount of time spent executing the native-code version of each function.

In order to charge each function properly, the Interpreter must keep a stack of function activation records. Whenever a function entry event occurs, the function and its current compilation status are pushed on the stack. When a function exit event occurs, the corresponding function activation record is popped from the stack.

The replace-preemptive strategy complicates our story. In order to keep a proper accounting of time spent in interpretation and time spent in execution of native code, the Interpreter object must be informed whenever a Compiler event occurs.

4.3 Simulator Output

Statistics on each function are collected during the simulation. These statistics include:

Number of calls The number of times the function has been called.

Accumulated interpreted time The total amount of time, over all calls, spent interpreting the function.

Accumulated native time The total amount of time, over all calls, spent running the native code for the function.

In addition, statistics on the entire program are gathered. This includes:

- total execution time,
- time spent interpreting source code,
- time spent executing translated (non-library) code,
- time spent executing library routines, and
- time spent translating source code to native code.

As mentioned earlier, we are assuming that the Interpreter and Compiler modules are *physically* running in parallel, so the total execution time reported does not include the time spent translating. However, the total time for a single-processor system can be calculated by simply summing the total execution time and the translation time, and adding in some additional amount to account for the overhead of context switching.

In addition to the above, the following information about source files can be output:

- sorted list (in descending order) based on the total number of calls to all functions defined in the file,
- sorted list (in descending order) based on the total execution time of all functions defined in the file, where the total execution time for a function is calculated without the interpretation penalty, and
- the point in time that the Compiler module *completes* translation of each file.

The sorted lists are useful for implementing the most-frequently-executed-overall and longest-overall strategies. The lists can be used to specify the compilation order for subsequent runs of the simulator with the same setup and behavior files.

4.4 Simulation of Real Programs

To use this simulator to simulate the performance of real programs it is necessary to somehow generate the required setup and behavior files.

The setup information for the simulator was not too difficult to obtain. The `make` utility was used to generate file dependencies [3]. The `ctags` utility determined which functions were contained in each source file. File sizes were obtained using `wc`. We gathered data on compilation times by compiling the programs with the `gcc` compiler and using the `time` utility to report the system and user time needed to compile each file.

Obtaining the behavior traces proved to be more difficult. Neither `Gprof` nor `purify` could supply us with the needed data.

We were fortunate to have Jim Larus' excellent tool `EEL` [15] for instrumenting the executables to generate trace information. Noah Treuhaft[2] used `EEL` to create a utility (called `eer`) to modify binary executables of our test programs. When these instrumented binaries are subsequently executed, each procedure entry and exit causes a time-stamped record to be generated for the behavior file. While the behavior traces produced in this manner are voluminous, they provide a detailed account of where time is spent in the program's functions.

[2]Noah Treuhaft was a visiting student from Oberlin College who was here as part of the Summer Undergraduate Research Assistantship program.

Chapter 5

ProGenitor

In the course of analyzing the performance of real programs with our simulator, we decided that it would be advantageous to be able to modify various characteristics of the programs to determine the extent to which these characteristics influence the performance. Making extensive modifications to the actual programs or writing the test programs from scratch both seemed infeasible. Instead, we developed the *ProGenitor* program.

The ProGenitor program is a *pseudo-program generator*. It takes as input the desired characteristics of a program and generates the setup and behavior files required by our continuous compilation simulator. Using ProGenitor we were able to examine the performance of programs fitting the characteristics we desired without the undue expense of creating each program or searching for existing programs exhibiting some particular trait.

The design of ProGenitor was necessarily driven by the use for which it was created. In particular, the defining characteristics of the generated programs are those which we thought might deserve the most consideration when examining their effect upon the efficiency of the continuous compiler.

The most obvious program trait to consider in terms of its effect on program performance is the relative amount of time spent in library routines as opposed to user routines. Library routines, such as those found in system libraries, are precompiled and available to the continuous compiler in native-code form at all times while user routines are defined in the program source files, which start out in interpreted form. As the percentage of time spent in library routines increases, the performance of the program using the continuous compilation model should approach that of executing a native-code version of the program [7].

ProGenitor thus focuses on the distinction between library routines and user routines since it was thought that this would be the factor that most influenced the performance of the continuous compiler. ProGenitor allows the defining traits of the library and user routines to be specified independent of each other, as well as providing the ability to specify the interaction between the two disjoint sets.

The relative amount of time spent in library and user routines is dictated in part by the relative number of routines of each type. However, the library/user time ratio is also dictated by a number of other factors, such as the average length of time spent in each function and the overall call structure.

In addition to its contribution to the library/user time ratio, the call structure is another factor likely to affect the performance of the continuous compiler. By the term "call structure" we are referring to the overall structure of the program as delineated by the component functions. For example, the call structure of a program would be considered wide and shallow if the program consisted of a large number of functions with a short amount of time spent in each function. At the other extreme, a program would be said to have a narrow and deep call structure if it spent a large amount of time in relatively few functions. While most real programs are not likely to fit neatly into such ordered categories, it is still a helpful generalization. The ProGenitor program was designed with this distinction in mind.

ProGenitor allows most of the input parameters to be specified as probability distributions rather than just exact values. The purpose of this is two-fold. First, some parameters are inherently multivalued and therefore fit probability distributions better than single, fixed values. An example would be the number of functions defined in each source file. This number is not likely to be the same for each source file of a program. However, rather than specify it separately for each source file, we took the more convenient approach of specifying the number of functions per source file as fitting a probability distribution.

The second reason for using probability distributions to specify various parameters is to allow us to generate closely related programs from the same specification by the simple expedient of varying the random seed. For example, by specifying the number of source files as a probability distribution rather than a single value, we can easily generate programs that are similar in structure, but with a varying amount of source files.

The individual parameters that are supplied to ProGenitor are described in detail in Section 5.1 and the allowed probability distributions are described in Section 5.2.

5.1 Input Parameters

Each of the following parameters is specified separately for user and library routines. Several of the parameters for the library routines are not relevant to our continuous compilation model, but they are still specified separately.

Number of source/header files The number of source files and the number of headers files are specified separately for both user and library routines. [1] For library routines these values are not too important, but for user routines they can have a significant effect since the time required for compilation is directly related to the number of files needing translation. However, the influence of this parameter is overshadowed by the effect of the next parameter: the number of functions per source file.

Number of functions per source file Since the Compiler module of the continuous compiler operates on the granularity level of the file (*i.e.*, it translates an entire file at a time, and the native-code version of the routines contained in the file is not available until the entire file is translated), the number of user routines per source file could profoundly affect performance. In addition, the *total* number of routines is *not* specified. Instead, the number of functions per source file is multiplied by the number of source files to determine the total number of functions. Again, the values for user and library files are given separately.

"Popularity" of each header file This parameter is used to indicate how many source files, on average, include a given header file.

Number of lines per source file This parameter is used to determine the time required to translate a given file to native code. Other parameters (described below) give the coefficients of the equation used to calculate the translation time based on the number of lines in the source file. Again, although the value

[1] For those readers not familiar with the C programming language, header files are files which are "included" in (*i.e.*, referenced by) one or more source files. They usually contain things such as declarations of constants and function prototypes.

of this parameter for library files has no affect on our continuous compilation simulation, it is specified separately for user and library files for the sake of generality and possible future applications.

Coefficients of the translation-time equation The equation used to calculate the translation time for a file is a standard quadratic equation

$$t = ax^2 + bx + c$$

where x is the number of lines in the file. The coefficients of this equation are given as parameters to ProGenitor.

Execution time between calls This parameter gives the time, in milliseconds, that a given function executes between making calls to other functions. Combined with the number of call sites per function, this parameter also specifies the total execution time of the function. Like all of the parameters, this one is specified separately for user and library functions. However, unlike the above parameters, the value of this parameter for library functions is as important as its value for user functions.

Number of call sites per function This parameter specifies the number of function calls made *by* a given function. It can arguably be said to have the most direct influence on the call structure of the generated program.

Maximum recursion depth When the ProGenitor program was first developed, this parameter was not included. However, due to the nondeterministic way in which the call structure of the generated program is built, mutually recursive function calls were sometimes generated, causing the program to get stuck in an endless chain of calls. This parameter was added to alleviate that problem.

Chance of calling a user function This parameter specifies the chance of a given call site being to a user function as opposed to a library function. This parameter has the most direct influence on relationship between user and library routines. The value of this parameter for user functions indicates the percentage of time that a user function calls other user functions rather than library functions. Using a nonzero value of this parameter for library functions allows us to model callbacks, where a library routine makes a call to a user function.

Chance of a call site being fixed A value of less than 1 for this parameter allows us to simulate variable functions calls, emulating the use of function pointers in real programs.

Chance of just returning This parameter specifies the chance of a function returning immediately when called.

The cutoff time for the program execution trace is also specified as a parameter to ProGenitor. If the generated program does not finish execution within the specified time, the trace is cut off to simulate the program being interrupted by the user.

Two other parameters, not directly related to the generated program, are required by the ProGenitor program. The first is the seed for the random number generator. Providing this as input allows us to duplicate previous executions exactly, or vary the execution nondeterministically, as needed.

The second parameter is a flag indicating if information about the library routines and files should be included in the output setup file. Our continuous compilation simulator treats any routine that is referenced in the behavior trace, but not in the setup file, as a library routine. Therefore, when generating programs to be used with our simulator, it is necessary that the library routines not be included in the setup file. However, the ability to include that information, perhaps for use with future applications, was retained.

5.2 Probability Distributions

While it is possible to specify the defining characteristics of the desired program to ProGenitor as exact values, ProGenitor also has the ability to accept the specification of certain probability distributions as values for most of the parameters. In particular, most of the input parameters are best specified as conforming to either a uniform or normal distribution.

Consider for example the parameter specifying the number of functions per source file. If this value is given as a single discrete number—say, for example, 3—then each source file will define the exact same number of functions: 3. While this behavior may sometimes be what is desired, it is certainly not very flexible. As an alternative, the number of functions per source file can be specified as following a uniform distribution with lower and upper bounds, say 1 and 5. The exact number of functions defined by each source file will then be generated randomly by ProGenitor,

but is guaranteed to be between 1 and 5 (inclusive); and, furthermore, the values chosen will follow a uniform distribution.

In addition to a uniform distribution, it is also possible to specify a (truncated) normal distribution. A normal distribution requires the specification of a mean and standard deviation, as well as minimum and maximum boundaries.

The ProGenitor program will accept a probability distribution as input for all parameters except the execution cutoff time and the random seed. For some parameters, such as the number of functions per source file or the number of call sites per function, multiple values can be used within a single generated program. For other parameters, such as the number of source files, only one value is needed. However, using distributions for these parameters is still sometimes valuable. By using distributions for these parameters, and by specifying various seed values for the random generator, it is possible to generate a *family* of closely related programs which share certain characteristics, but are not exactly the same.

Choosing truncated normal and uniform as the allowable distributions was rather arbitrary. Ideally, the distributions would be modeled after the characteristics of real programs, but as discussed in Section 2.3, no such study of the characteristics of modern modular or object-oriented programs has been done (at least as far as we know). If such information becomes available in the future, it should not be much trouble to expand ProGenitor take advantage of it.

The method used to generate values following uniform and normal distributions are based on those presented by Knuth [].

5.3 Program Generation

Explaining how ProGenitor works is probably best done by example. We will describe a sample execution of ProGenitor using the input parameters listed in Table 5.1.

In addition to fixed values, some of the parameters have been specified as following either a uniform or normal distribution, as described earlier in this chapter. Uniform distributions are specified with a "u" followed by the minimum and maximum allowed values. Normal distributions are specified with an "n" followed by the mean, the standard deviation, and the minimum and maximum allowed values, in that order.

The first thing ProGenitor does is generate a set of user source "files" and a set of user header "files." It is important to note that the "files" generated are not

actual disk files. Rather, they are objects internal to ProGenitor that correspond to source and header files of a program.

The number of source and header files generated is dictated by the input parameters. For example, the value given for the parameter **Number of Source Files** in the **User File Parameters** section of Table 5.1 is "u 2 5." This means that the number of user source files should be between 2 and 5 (inclusive), and furthermore, if the same inputs were supplied to ProGenitor multiple times, then the distribution of the number of user source files across the multiple runs would be uniform.

So, the first thing ProGenitor will do is pick the number of user source files to generate using standard pseudo-random number generation techniques. Similarly, ProGenitor will pick a value (based on a uniform distribution) between 4 and 6 for the number of user header files.

ProGenitor will then pick values for the number of *library* source files and header files in the same way, using the parameters in the **Library File Parameters** section of Table 5.1.

Once the source and header file objects have been generated, ProGenitor will decide how many source files should include each header file and will link the files appropriately. In Table 5.1, the **"Popularity" of Header Files** is given as "n 2 5 1 5." This indicates that the distribution of values chosen for this parameter should follow a truncated normal distribution with a mean of 2, a standard deviation of 5, and bounded by the values 1 and 5. ProGenitor will generate a separate (but not necessarily unique) value following this distribution for *each* header file. It will then randomly pick a corresponding number of source files to include each header file. However, like in most real programs, User source files are allowed to include library header files, but library source files are not allowed to include user header files.

We now have our source and header files linked with appropriate dependencies. This is enough to generate the first section of the setup file.

Next, ProGenitor creates the "functions" contained in the source files. (As with the files, these are not real functions, but rather are internal objects representing functions.) For each source file, ProGenitor picks a value according to the parameter **Functions per File** and creates a correspondingly sized set of function objects for the file. We now have enough information to generate the second section of the setup file.

ProGenitor then uses the **Lines per File** parameter to generate the length of each file. The compilation time for each file is then calculated using the specified

coefficients for the quadratic equation. This gives us enough information to generate the third, and last, section of the setup file.

Now that the setup file is complete, ProGenitor prepares to create the behavior trace. First, however, the remaining static properties of the functions are generated. ProGenitor assigns each function a number of call sites and a maximum recursion depth based on the corresponding input parameters. The number of call sites is the number of other functions that the given function will call whenever it is executed. The call sites are modeled as an *ordered* set of objects so that they will be visited in the same order every time the function is executed.

We are now ready to simulate execution. A user function is picked to be the starting point, the clock counter is set to 0, and a function-entry event is output to the behavior file.

ProGenitor then proceeds to simulate execution of the function. First, it uses the **Execution Time between Calls** parameter to increment the clock counter. Then it visits the first call site of the function.

Since this is the first time this call site has been visited, ProGenitor has a little extra work to do. First of all, it needs to decide if this call site will remain fixed (*i.e.*, call the same function) for the entire execution of the program or if it will vary (modeling a function pointer). ProGenitor decides this based on the **Chance of Call Site Being Fixed** parameter. A value, d, is generated according to specified distribution. Another value, p, following a uniform distribution and ranging from 0 to 1 is generated. If $p < d$ then the call site will be fixed, else it will be variable.

ProGenitor then needs to decide if this call will be to a library or another user function. This is determined using the **Chance of Calling a User Function** parameter in a manner identical to that of deciding if the call site was fixed. Once this is decided, ProGenitor will randomly pick a function from the proper set. If the call site is to be fixed, then this function is permanently assigned to the site; otherwise, it is only used for this particular call.

ProGenitor now enters the chosen function and another function-entry event is output to the behavior file. From now on, the first thing ProGenitor does upon entering a function is check to make sure that it has not exceeded the maximum recursion depth assigned to that function. If it has, then it simply returns to the calling function (outputting the appropriate function-exit event to the behavior file) and continues execution from there.

The next thing ProGenitor does upon every function entry is generate a probability for the function just returning in a manner similar to that of deciding if a call site was fixed. If this value is affirmative, then ProGenitor immediately returns to the calling function (again outputting a function-exit event).

If it is decided that this function is to be executed, then ProGenitor proceeds just as with the first function: the clock counter is advanced and the first call site is visited.

ProGenitor continues simulating execution of the program in this manner until one of two things happens. First, if run long enough, ProGenitor will eventually return all the way up the call stack; that is, the starting function will return. If this happens, then execution of the simulated program is said to have terminated normally.

The second possibility is related to a small detail omitted in the above discussion. It is possible to specify a "maximum execution time" to ProGenitor. This is not the maximum amount of time that ProGenitor itself will run but rather the maximum amount of time for the *simulated* program to run. If a maximum execution time is given, then at every step ProGenitor checks the current value of the clock counter against this maximum time. If the simulated program has not terminated normally by the time this value is reached, then the behavior trace is cut off abruptly, emulating the abnormal termination of the program.

In either case, we now have a viable behavior file to use with our continuous compilation simulator.

Table 5.1: Example ProGenitor Input

User File Parameters	
Parameter	Value
Number of Source Files	u 2 5
Number of Header Files	u 4 6
"Popularity" of Header Files	n 2 5 1 5
Functions per File	n 3 1 1 5
Lines per File	n 300 100 20 1000
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 3
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.5 0.05 0 1
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

Library File Parameters	
Parameter	Value
Number of Source Files	u 2 5
Number of Header Files	u 4 6
"Popularity" of Header Files	n 2 5 1 5
Functions per File	n 5 1 1 10
Lines per File	n 500 200 20 1500
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 4
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.0005 0.05 0 0.002
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

Chapter 6

Experiments

In this chapter we explore the performance of the continuous compiler by presenting results obtains through experiments conducted with our simulator.

In Section 6.2 and Section 6.3 we compare the performance of the continuous compilation model with that of the traditional compilation and traditional interpretation models.

In Section 6.4 and Section 6.5 we examine how the implementation of various aspects of the continuous compiler affect performance. Section 6.4 compares the replace-at-call and replace-preemptive schemes while Section 6.5 compares the effects of compiling the files in different orders.

In Section 6.6 and Section 6.7 we examine how certain features of the executed programs can affect the performance of the continuous compiler.

6.1 Source Programs

As mentioned earlier, we used both *real* and *synthetic* programs in our experiments. This section describes the real programs that were used.

We studied the following programs: `rcc`, `gzip`, `gnuchess`, `ghostview`, `pico`, and `render`. (See Table 6.1.) `Lcc`, `gzip`, `gnuchess`, and `ghostview` are all under the GNU General Public License. `Pico` is copyrighted by the University of Washington. `Render` was developed as a class project at Washington University.

The performance of each program is, of course, dependent on its input. The input for each program is as follows.

Table 6.1: Real Programs Studied

Program	Description
rcc	Compiler portion of lcc [5]
gzip	GNU file compression utility
gnuchess	GNU chess playing program
ghostview	Postscript file previewer
pico	Simple text editor
render	Simple raytracing program

rcc The input to `rcc` was the source file x86.c, which is `lcc`'s code generation module for the Intel x86 architecture.

gzip The input provided to `gzip` was a 206 Kbyte directory.

gnuchess The chess program was set up to play against itself with a very shallow search depth.

ghostview `Ghostview` was run with the file "ghostview.ps" specified on the command line. This 92 Kbyte Postscript file is a documentation file included with `ghostview`. Each of the 14 pages was viewed quickly before exiting the program.

pico The editor was started without an input file. Text was entered for about a minute, the file was saved, and then the program was exited.

render The input to the `render` program was a batch file which instructed the program to raytrace a very simple scene.

6.2 Continuous Compilation vs. Traditional Compilation

With these experiments we hope to show that the continuous compilation model compares favorably against the traditional compilation model in terms of performance. To do so, we compare the cold-start recovery times of the two models.

The cold-start recovery time is defined as the time necessary to reach a particular point in the program's execution when starting from the initial, uncompiled source. For the traditional system, the recovery time includes the time necessary to

Table 6.2: Cold-start with Traditional Compilation System

Program	Compile Time	Link Time	Execution	Total
rcc	112	4.9	16.9	133.8
gzip	24	0.9	1.9	26.8
gnuchess	36	1.7	16.3	54.0
ghostview	73.4	1.5	38.0	112.9
pico	63.5	2.5	63.5	129.5
render	46.8	1.3	131.0	179.1

(Time given in seconds.)

compile the program and then execute the program up to the specified point of execution. The recovery time for the continuous compilation system is the time necessary to reach that same point in the program's execution using the continuous compiler. The traditional-system recovery time is calculated by simply summing up the times needed to compile, link, and execute the program. The continuous-system recovery time is obtained from our simulator. Table 6.2 shows the cold-start recovery time in seconds for a traditional compilation system for each of the real programs under consideration.

In Table 6.3 we compare the recovery time, in seconds, of the traditional and continuous compilation systems from a cold-start, using the random compilation selection strategy (see Section 3.2), the replace-at-call replacement strategy (see Section 3.3), and an interpretation penalty of 10. By *interpretation penalty* of 10 we

Table 6.3: Comparison between Traditional and Continuous Replace-at-Call

Program	Recovery Time		Speedup
	Traditional	Continuous Replace-at-Call	
rcc	133.8	88.8	1.5
gzip	26.8	13.7	2.0
gnuchess	54.0	35.8	1.5
ghostview	112.9	377.6	0.3
pico	129.5	63.9	2.0
render	179.1	194.9	0.9
Compilation Strategy: random			

mean that we are assuming that it takes 10 times long to interpret a given piece of code than it would to execute the equivalent native code.

In four out of the six cases, the continuous compiler shows a significant improvement in performance, from 1.5 up to 2 times faster than the recovery time of the traditional compiler. However, in the other two test cases, `ghostview` and `render`, we see a decrease in performance. With `render` the slowdown is fairly small, but with `ghostview` the continuous compilation recovery time is more than three times the recovery time of the traditional compiler.

We can uncover a clue about the possible cause for the poor performance of `ghostview` with the continuous compiler using replace-at-call if we examine the performance using replace-preemptive. Table 6.4 shows the performance of `ghostview`, `pico`, and `render` using replace-preemptive instead of replace-at-call. (We are still using the random compilation selection strategy and an interpretation penalty of 10.)

Table 6.4: Comparison between Traditional and Continuous Replace-Preemptive

Program	**Recovery Time**		**Speedup**
	Traditional	**Continuous Replace-Preemptive**	
ghostview	112.9	83.2	1.4
pico	129.5	63.9	2.0
render	179.1	157.3	1.1
Compilation Strategy: random			

As expected, we see an improvement in the performance of the continuous compiler when using the replace-preemptive method over the replace-at-call method. With `render` this improvement is small, but enough to push it past the performance of the traditional compiler. With `ghostview`, however, the improvement is tremendous. The recovery time dropped from 377.6 seconds to 83.2 seconds, a speedup of over 450%. Again, this is enough to beat the recovery time of the traditional system. `Pico` is an anomaly; the recovery time of the continuous compiler is identical when using either the replace-at-call or replace-preemptive strategy. In Section 6.4 we examine the effect of the replacement strategies in more detail. First, however, a few comments are needed about the times given in Table 6.3 and Table 6.4.

The times given for the continuous compiler are the times for the Interpreter module only. In other words, we are assuming that there are two processors, one running the Interpreter module and one running the Compiler module. However, the times given for the traditional compiler assume sequential, single-processor execution. To make the comparison more fair, we need to either give the traditional compiler two processors or restrict the continuous compiler to one processor.

Giving the traditional compiler two processors would not simply halve the recovery times. The performance depends very strongly on the implementation of both the compiler and the program being executed. Both would have to be redesigned to take advantage of the additional processor. This would be a very complex process, and we have no easy way to determine what the actual speedup would be.

On the other hand, the continuous compiler is already parallel in nature and can easily be designed to take advantage of the extra processor. However, in order to make the comparison fair, we will compare the performance of the traditional compiler with that of the continuous compiler running on a system with only one processor.

To calculate the recovery time for the continuous compilation system with only one processor, we will simply add the time spent in the Interpreter module to the time spent in the Compiler module. (To be more realistic we should add some extra time to account for the overhead of context switching between the Interpreter and Compiler modules, but we will ignore that for now.)

Table 6.5: Single-processor performance of render

Compilation Strategy	**Compiler Module**	**Interpreter Module**	**Total Time**	**Speedup over Traditional**
random	46.8	157.299	204.099	0.9
mfex-so-far	46.8	143.083	189.883	0.9
long-so-far	46.8	143.345	190.145	0.9
small	46.8	173.064	219.864	0.8
large	46.8	143.988	190.788	0.9
mfex-overall	46.8	154.183	200.983	0.9
long-overall	46.8	139.956	186.756	1.0

Table 6.5 shows the performance of the render program for each of the compilation strategies on a single-processor system using the replace-preemptive scheme and an interpretation penalty of 10. The values in the "Speedup over Traditional"

column were calculated by dividing the 179.1 seconds recovery time of the traditional compiler (see Table 6.2) by the time in the "Total Time" column.

As expected, the recovery time of the continuous compiler with a single processor is worse than that of the traditional compiler, due to the extra time spent by the continuous compiler interpreting code before the code gets compiled. However, the difference is fairly small. In fact, with the longest-overall compilation strategy, the difference is only 7.6 seconds, which is less than a 5% slowdown. Even in the worst case, produced by the smallest-first compilation strategy, the slowdown is only about 20%

Table 6.6: Single-processor performance of pico

Compilation Strategy	Compiler Module	Interpreter Module	Total Time	Speedup over Traditional
random	63.5	63.856	127.356	1.0
mfex-so-far	63.5	63.619	127.119	1.0
long-so-far	63.5	63.619	127.119	1.0
small	63.5	64.038	127.538	1.0
large	63.5	63.767	127.267	1.0
mfex-overall	63.5	63.611	127.111	1.0
long-overall	63.5	63.608	127.108	1.0

When we look at the values for `pico` (again using replace-preemptive and an interpretation penalty of 10), as shown in Table 6.6, we see a similar trend, except that with `pico`, all the values for the continuous compiler are slightly *better* than the recovery time of 129.5 seconds for the traditional compiler.

An important point to bear in mind is that with the traditional compiler, the user must wait for compilation to finish before any useful work can be done. With `pico` this means waiting over 63 seconds before the program even starts execution. In contrast, execution of the program begins immediately with the continuous compiler.

The immediate response time of the continuous compiler is an extremely important benefit. Imagine a situation where someone is trying to decide whether to purchase a dual-processor system to use with a continuous compiler or, for the same money, a single-processor system, with a processor that is twice as fast as the ones in the dual-processor system, for use with a traditional compiler. Even though compilation now occurs twice as fast as before, the traditional compiler would still take over 30 seconds to compile `pico`. With the dual-processor continuous compiler, program

execution begins immediately. In addition, even if we assume that the recovery time for `pico` with the twice-as-fast processor is half that of the 129.5 seconds given in Table 6.2, it is still greater than the recovery time of around 63.6 seconds for the dual-processor continuous compiler. In actuality, the situation for the twice-as-fast traditional compiler would be even worse. With programs like `pico`, which spend most of their time waiting for input from the user (see Section 6.6), doubling the processor speed is not going to halve the execution time.

When we look at the values for `ghostview`, shown in Table 6.7, we get an even greater surprise: the continuous compiler *outperformed* the traditional compiler's recovery time of 112.9 seconds by up to 7.2 seconds. (This is again with the replace-preemptive strategy and an interpretation penalty of 10). If we take a closer look

Table 6.7: Single-processor performance of ghostview

Compilation Strategy	Compiler Module	Interpreter Module	Total Time	Speedup over Traditional
random	73.4	83.175	156.575	0.7
mfex-so-far	52.3	54.177	106.477	1.1
long-so-far	52.3	53.437	105.737	1.1
small	73.4	94.838	168.238	0.7
large	56.2	56.597	112.797	1.0
mfex-overall	73.4	80.991	154.391	0.7
long-overall	51.4	52.848	104.248	1.1

at Table 6.7, we find that in four of the cases (mfex-so-far, long-so-far, largest-first, and long-overall) the continuous compiler did not have time to compile the entire program. In fact in these four cases, the continuous compiler running on a dual-processor system would finish execution of the program before a traditional compiler would even finish compiling the program.

6.3 Continuous Compilation vs. Traditional Interpretation

Due to the nature of the continuous compilation model, we expect that it will always outperform interpretation if run on a dual-processor system. However, it is more instructive to compare the two when both are run on a single-processor system.

In order to calculate the interpretation time for each program, it was necessary to know how much time the program spent in "library" code and how much it spent in "user" code. The user code is interpreted, but the library code is precompiled and runs at native-code speed.

The amount of time spent in library code is determined by our continuous compilation simulator. (These values are given in various tables later in this chapter.) Knowing this, we can then subtract from the overall execution time (given in Table 6.2) to determine the amount of time spent in user code. The interpretation time is then calculated from the following equation

$$t_i = (t_e - t_l) * p + t_l$$

where t_e is the total native-execution time, t_l is the time spent in library code, and p is the interpretation penalty.

Table 6.8 compares the performance of `ghostview`, `pico`, and `render` on a single-processor continuous compiler with that of traditional interpretation. We used an interpretation penalty of 10 for both the continuous compiler and the interpreter. The continuous compiler used the replace-preemptive scheme and the largest-first compilation strategy. With `ghostview` and especially with `render` we see the expected improvement. `Render` in particular shows a huge performance increase—seven times faster—when using continuous compilation instead of traditional interpretation.

Table 6.8: Continuous Compilation vs. Interpretation

Program	**Continuous Compilation**	**Interpretation**	**Speedup**
ghostview	104.248	379.991	3.6
pico	127.108	64.004	0.5
render	186.756	1309.838	7.0

`Pico`, on the other hand, performs worse under single-processor continuous compilation. This is due to the fact that `pico` spends an overwhelmingly large percentage of time executing precompiled library code. `Pico` spends 63.444 seconds, out of 63.5 seconds, in library code—more than 99.9%. The continuous compiler loses because it spends half of its time compiling code that is executed for less than 0.1% of the total execution time. However, unlike traditional compilation, the program

is being executed concurrently during this time. In fact, the actual execution time of pico under the continuous compiler would probably be closer to the 63.5 seconds since the user would not need to wait for compilation to complete.

In contrast to pico, render and ghostview spend almost all of their time, over 99.9%, executing user code. Any program that spends a fair amount of time in user code will show a gain in performance when comparing continuous compilation to interpretation; and the longer the program runs, the greater the gain will be. This is why render shows such a tremendous gain in performance.

6.4 Comparing Replacement Strategies

Now that we know that the continuous compiler performs well compared to both traditional compilation and traditional interpretation, at least when using the replace-preemptive scheme, we take a closer look at the performance of the continuous compiler using replace-at-call to see if the added complexity of replace-preemptive is actually necessary.

Since we are now comparing the performance of the continuous compiler with itself, we are switching back to a dual-processor system. The times reported in this section, and in the rest of the chapter, are all based on a dual-processor system.

Table 6.9 compares the performance of ghostview using replace-at-call and replace-preemptive for all seven of the compilation strategies described in Section 3.2. Although the optimal compilation strategy differs for the two replacement schemes—largest-first is the best with replace-at-call while longest-overall is the best with

Table 6.9: Comparison of Replacement Strategies with Ghostview

Compilation Strategy	**Replace-at-Call**	**Replace-Preemptive**
random	377.647	83.175
mfex-so-far	377.547	54.177
long-so-far	377.606	53.437
small	377.553	94.838
large	373.412	56.597
mfex-overall	376.732	80.991
long-overall	377.579	52.848

replace-preemptive—the tremendous performance improvement of replace-preemptive over replace-at-call holds for all of the strategies.

When we look at the distribution of time over the functions in `ghostview`, we find that the vast majority of the time is spent in the function "main." In fact, when running `ghostview` with replace-at-call and most-frequently-executed-so-far, 369.100 seconds are spent in main. This is over 97% of the total execution time. And all of it is accumulated during interpretation.

The problem is that with replace-at-call, the continuous compiler will never get to run the native-code version of the function main, even if it gets compiled first. This is because with replace-at-call, once interpretation of a function has begun, that instantiation of the function will be interpreted until it exits.

With replace-preemptive on the other hand, execution of the function main can switch from interpretation to native execution as soon as the file is compiled. With the most-frequently-executed-so-far strategy, this file completes compilation at 16.9 seconds into program execution. Because of this, only 8.978 seconds are spent on interpretation of main. From then on, the native-code version of main is executed, accumulating another 36.013 seconds of execution time. So, with replace-preemptive, only 44.991 seconds are spent in the function main. This is still over 83% of the total program execution time, but because of the switch to native code, it is a vast improvement over the performance with replace-at-call.

In actuality, the situation with `ghostview` is not as bad as the numbers indicate. Due to a quirk in the way `eer`, our tool for collecting the behavior traces (see Section 4.4), works, the time spent in dynamically linked functions is charged to the calling statically-linked function. So the function main in `ghostview` is getting charged with a lot of time that was actually spent in various X-Window library routines, including XtAppMainLoop. However, the general principle remains true. With the replace-at-call strategy, the first few functions entered, including the main function, are going to be interpreted throughout the program's lifetime. If one of those functions happens to be large, or an event-loop, or any type of loop in which a lot of time is spent, then the program will exhibit the same performance degradation as exhibited by `ghostview` when using replace-at-call.

The behavior of `pico` with regard to replacement strategy appears to be just the opposite as that of `ghostview`. `Ghostview` shows a vast improvement when using replace-preemptive instead of replace-at-call. `Pico` shows no improvement at all.

Table 6.10: Pico (Replace-at-Call)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in			Total Time
			Interp. Code	Comp. Code	Lib. Code	
random	63.5	•	0.378	0.034	63.444	63.856
mfex-so-far	63.5	•	0.116	0.060	63.444	63.619
long-so-far	63.5	•	0.116	0.060	63.444	63.619
small	63.5	•	0.581	0.013	63.444	64.038
large	63.5	•	0.280	0.043	63.444	63.767
mfex-overall	63.5	•	0.106	0.061	63.444	63.611
long-overall	63.5	•	0.104	0.061	63.444	63.608

Table 6.10 shows the performance of `pico` using replace-at-call for all of the compilation strategies, broken down to show where the time is spent. The column "Time Comp." gives the time spent by the Compilation module compiling the program. A mark in the "Completed Comp." column means that the Compiler module completed compiling the entire program during execution. (This was true for every run of `pico`.) The "Time Spent in Interp. Code," "Comp. Code," and "Lib. Code" columns list the total time accumulated interpreting source code, executing compiled native code, and executing precompiled library code, respectively. The "Total Time" column gives the total execution time of the program.

Table 6.11: Pico (Replace-Preemptive)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in			Total Time
			Interp. Code	Comp. Code	Lib. Code	
random	63.5	•	0.378	0.034	63.444	63.856
mfex-so-far	63.5	•	0.116	0.060	63.444	63.619
long-so-far	63.5	•	0.116	0.060	63.444	63.619
small	63.5	•	0.581	0.013	63.444	64.038
large	63.5	•	0.280	0.043	63.444	63.767
mfex-overall	63.5	•	0.106	0.061	63.444	63.611
long-overall	63.5	•	0.104	0.061	63.444	63.608

Table 6.11 shows the performance of `pico` using the replace-preemptive strategy. A close examination of Table 6.10 and Table 6.11 will reveal that they are practically identical.

The explanation for this invariance can be found by comparing the amount of time spent in precompiled library code to the time spent in both interpreted and compiled source code. Over 98% of the total execution time is spent in library code. Upon examining where this time was spent, it turns out that 63.294 seconds were spent reading input from the keyboard.

This phenomenon is not restricted to `pico`. Any program which exhibits this behavior will see similar performance benefits. The higher the percentage of time that a program spends in library functions, or waiting for I/O, the better its performance will be using the continuous compiler. We examine this subject again in Section 6.6.

Table 6.12 and Table 6.13 show the performance of the `render` program for the replace-at-call and replace-preemptive methods, respectively.

Table 6.12: Render (Replace-at-Call)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in			Total Time
			Interp. Code	Comp. Code	Lib. Code	
random	46.8	•	71.035	123.838	0.018	194.891
mfex-so-far	46.8	•	27.807	128.161	0.018	155.986
long-so-far	46.8	•	27.931	128.149	0.018	156.098
small	46.8	•	47.838	126.158	0.018	174.014
large	46.8	•	22.037	128.738	0.018	150.793
mfex-overall	46.8	•	91.196	121.822	0.018	213.036
long-overall	46.8	•	25.163	128.426	0.018	153.607

The `render` program could be considered the opposite of `pico` since it spends almost no time at all in precompiled library code, only 0.018 seconds. This is far less than 1% of the total execution time for even the best case performance.

The replace-preemptive strategy again performs better than replace-at-call, although the improvement is not nearly as drastic as with `ghostview`.

The values of 194.891 seconds for the random compilation strategy and 213.036 seconds for the most-frequently-executed-overall strategy with replace-at-call is something of an anomaly; they are the only cases for which the performance of the continuous compiler is worse than that of the traditional model.

Table 6.13: Render (Replace-Preemptive)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in Interp. Code	Time Spent in Comp. Code	Time Spent in Lib. Code	Total Time
random	46.8	•	29.265	128.015	0.018	157.299
mfex-so-far	46.8	•	13.470	129.595	0.018	143.083
long-so-far	46.8	•	13.761	129.566	0.018	143.345
small	46.8	•	46.782	126.264	0.018	173.064
large	46.8	•	14.476	129.494	0.018	143.988
mfex-overall	46.8	•	25.804	128.361	0.018	154.183
long-overall	46.8	•	9.995	129.942	0.018	139.956

One other thing to note in the scores for `render` is that the compilation strategy which gives the best performance differs for the two replacement strategies. This is looked at in more detail in Section 6.5.

6.5 Comparing Compilation Selection Strategies

In this section we examine the effect of the compilation selection strategy on the performance of the continuous compiler. Table 6.14 shows the performance of `ghostview` for each of the compilation strategies using replace-preemptive. The top two win-

Table 6.14: Ghostview (Replace-Preemptive)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in Interp. Code	Time Spent in Comp. Code	Time Spent in Lib. Code	Total Time
random	73.4	•	50.220	32.954	0.001	83.175
mfex-so-far	52.3		18.000	36.176	0.001	54.177
long-so-far	52.3		17.179	36.258	0.001	53.437
small	73.4	•	63.179	31.658	0.001	94.838
large	56.2		20.689	35.907	0.001	56.597
mfex-overall	73.4	•	47.793	33.196	0.001	80.991
long-overall	51.4		16.523	36.323	0.001	52.848

ners are longest-so-far and longest-overall, with a slight advantage going to longest-overall. This is not too surprising since these two methods select which files to compile based on usage of those files. However, it is interesting to note that, while the most-frequently-executed-so-far strategy performed well, the most-frequently-executed-overall strategy gave fairly poor performance.

The function which was called the most is the user function "readline" with 7018 calls. This is much more that the next highest total of 105 calls. However, a total of only about 1.22 seconds are spent in the readline function. So, even though the function is called many times, compiling it first does not help because only a short amount of time is spent in it. The poor performance of most-frequently-executed-overall is due to the compilation of the function main being put off until after the relatively useless compilation of the frequently called functions. On the other hand, the reason that most-frequently-executed-so-far does so much better is that when the Compiler module first picks a file to compile, main is one of the only functions that has been called, so it gets compiled earlier.

It is also important to note that for the four fastest cases—most-frequently-executed-so-far, longest-so-far, largest-first, and longest-overall—the Compiler module did not have enough time to complete compilation of the entire program. The compilation and link time of `ghostview` using a traditional compiler is 74.9 seconds (see Table 6.2). So, in these four cases, the continuous compiler completed execution of the program in less time than just compiling the program would take with the traditional model.

Table 6.15 shows the performance of `ghostview` for each of the compilation strategies using replace-at-call. Because of the overwhelming effect of the function main as described in Section 6.4, there is not much variance in the performance of the compilation strategies. However, the largest-first strategy does have a slight advantage. This same advantage can be seen in the performance of `render` with the replace-at-call method (see Table 6.12).

At first, the good performance of the largest-first strategy might seem rather counterintuitive since it minimizes the throughput of the Compiler module. Yet it sometimes gives the best performance for the continuous compiler as a whole when using replace-at-call.

The key is again the use of replace-at-call. The four compilation methods that are based on statistics collected while the program is running—longest-so-far, most-frequently-executed-so-far, longest-overall, and most-frequently-executed-overall—all

Table 6.15: Ghostview (Replace-at-Call)

Compilation Strategy	Time Comp.	Completed Comp.	Time Spent in Interp. Code	Time Spent in Comp. Code	Time Spent in Lib. Code	Total Time
random	73.4	•	377.411	0.235	0.001	377.647
mfex-so-far	73.4	•	377.300	0.246	0.001	377.547
long-so-far	73.4	•	377.366	0.239	0.001	377.606
small	73.4	•	377.306	0.245	0.001	377.553
large	73.4	•	372.706	0.705	0.001	373.412
mfex-overall	73.4	•	376.394	0.336	0.001	376.732
long-overall	73.4	•	377.335	0.242	0.001	377.579

give preference to those functions in which a lot of time has been spent. However, it is often likely that these functions are the very ones that we are stuck interpreting for the entire lifetime of the program because of the replace-at-call method. So, by compiling the files containing those functions first, we are actually wasting time since we can never use the compiled versions during the current program's execution.

So why does the largest-first strategy, which minimized the throughput of the Compiler module, perform better than the shortest-first strategy, which maximizes the throughput of the Compiler module, for `ghostview` and `render` when using replace-at-call? We will find when we look at the results for some generated programs in the next two sections that this is not true of all programs. Which compilation strategy gives the best performance seems to be related to the structure of the particular program being executed in a nontrivial manner.

In most cases, especially when using replace-preemptive, the longest-overall strategy does the best. However, this strategy unfortunately requires knowledge about the program that might not be available. It might be possible, though, to apply data saved from previous runs of the program to try to approximate longest-overall for the current run. As an alternative, longest-so-far, which could also be considered an approximation to longest-overall, seems to perform almost as well.

6.6 Effects of Library/User Ratio on Performance

In this section we examine the effects of the ratio of time spent in library code vs. user code in more detail. To this end we used our ProGenitor program to generate a

family of three programs which are identical in all respects except one: the chance of a user function calling another user function as opposed to a library function.

For easy reference, we have named the three programs based on the defining parameter of the chance of a user function calling another user function. In the "user" program, each function call made by a user function has a high probability (mean of 99%) of being to another user function. In the "library" program it is just the opposite; each function call made by a user function has a low probability (mean of 1%) of being to another user function. The "equal" program is in the middle; each function call made by a user function has approximately a 50% chance of being to another user function.

Table 6.16: Common User File Parameters of Programs Generated to Examine the Effects of the Library/User Ratio on Performance

User File Parameters	
Parameter	**Value**
Number of Source Files	20
Number of Header Files	10
"Popularity" of Header Files	n 5 5 1 10
Functions per File	n 5 1 1 10
Lines per File	n 500 200 20 1500
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 10
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	(See Table 6.18.)
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

Table 6.16 lists the parameters for the user files that are common between the three programs. Table 6.17 lists the parameters for the library files for the three programs, all of which are the same for all of the programs. The meaning of the parameters is explained in detail in Section 5.1. However, an explanation of the format of the values is in order.

In addition to fixed values, some of the parameters have been specified as following either a uniform or normal distribution, as described in Chapter 5. Uniform

Table 6.17: Library File Parameters of Programs Generated to Examine the Effects of the Library/User Ratio on Performance

Library File Parameters	
Parameter	Value
Number of Source Files	20
Number of Header Files	10
"Popularity" of Header Files	n 5 5 1 10
Functions per File	n 5 1 1 10
Lines per File	n 500 200 20 1500
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 10
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.0005 0.05 0 0.002
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

distributions are specified with a "u" followed by the minimum and maximum allowed values. Normal distributions are specified with an "n" followed by the mean, the standard deviation, and the minimum and maximum allowed values, in that order.

Table 6.18 summarizes the one parameter that varies between the three programs. Each value is specified as a normal distribution with a small standard deviation so that there is a little variance in the value generated for each function call, but not much.

Table 6.18: Chance of Calling Another User Function

Program	**Chance of Calling Another User Function**
User	n 0.99 0.05 0.9 1
Library	n 0.1 0.05 0 0.2
Equal	n 0.5 0.05 0 1

Table 6.19 summarizes the total execution times of the continuous compiler for all three of the test programs. Times are again in seconds.

Table 6.19: Variation of Library/User Ratio

Compilation Strategy	Replace-at-Call			Replace-Preemptive		
	User	Equal	Library	User	Equal	Library
random	102.279	78.407	42.174	102.193	78.275	42.174
mfex-so-far	101.962	69.321	40.501	101.856	69.200	40.501
long-so-far	100.873	67.571	40.525	100.832	67.362	40.516
small	97.747	65.888	41.278	97.642	65.744	41.278
large	108.305	81.824	42.596	108.192	81.632	42.160
mfex-overall	101.609	69.794	40.316	101.497	69.662	40.316
long-overall	101.045	67.112	40.315	100.971	66.937	40.298

(Time is in seconds.)

As expected, we see a steady increase in performance as the amount of time spent in library functions increases. The "library" program performs better than the "equal" program which in turn performs better than the "user" program. For comparison purposes, each program would need 96.140 seconds to compile and 40.000 seconds to execute using a traditional compilation model, for a total of 136.140 seconds. Even the worst case performance of the "user" program beats that by a fair margin.

When we examine the values to compare the performance of the two replacement strategies on the three programs, we see they are about equal. This suggests that the behavior file was generated in such a way as to not cause the problem seen with `ghostview` where we were stuck interpreting the first few functions entered for the entire lifetime of the program. This is, unfortunately, a drawback to the way the ProGenitor program is implemented. With the current version of ProGenitor, there is no way to give any function more importance over another. In other words, there is no way to emulate specific types of program structures, such as event-loops.

When we look at the values to compare the seven compilation strategies, we again see a difference from the programs studied earlier. This time it the smallest-first strategy which performs the best for both the "user" and "equal" programs, and it does so for both replace-at-call and replace-preemptive. For the "library" program, like the real programs studied in Section 6.5, it is the again the longest-overall strategy that performs the best, for both replacement strategies.

6.7 Effects of Density of Functions per Source File

As mentioned before, since the Compiler module operates on a file-based granularity level, it is possible that the density of functions per source file could have an effect on the performance of the continuous compiler. To test this, we generated two programs which are similar in most characteristics but differ in the number of functions defined in each source file.

Table 6.20: User File Parameters of the Many-Functions-per-File Program

User File Parameters	
Parameter	Value
Number of Source Files	20
Number of Header Files	10
"Popularity" of Header Files	n 5 5 1 10
Functions per File	n 20 5 10 30
Lines per File	n 500 200 20 1500
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 10
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.5 0.05 0 1
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

Table 6.20 and Table 6.21 list the parameters that were used to generate the "many-per-file" program.

Most of the parameters are the same as those used in the "equal" program in Section 6.6. However, the parameters to pay attention to here are the number of source files, the number of functions per file, and the number of lines per file.

The "many-per-file" program has 20 user source files with an average of 20 functions per file, for a total of around 400 functions. To duplicate this number of functions in the "one-per-file" program, we used 400 source files with one function per file. Table 6.22 lists the parameters for the user files in the "one-per-file" program. The parameters for the library files are exactly the same as those used by the "many-per-file" program.

Table 6.21: Library File Parameters of the Many-Functions-per-File Program

Library File Parameters	
Parameter	**Value**
Number of Source Files	20
Number of Header Files	10
"Popularity" of Header Files	n 5 5 1 10
Functions per File	n 5 1 1 10
Lines per File	n 500 200 20 1500
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 5
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.0005 0.05 0 0.002
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

Table 6.22: User File Parameters of the One-Function-per-File Program

User File Parameters	
Parameter	**Value**
Number of Source Files	400
Number of Header Files	10
"Popularity" of Header Files	n 5 5 1 10
Functions per File	u 1 1
Lines per File	n 25 10 10 50
Execution Time Between Calls	u 50 500
Call Sites per Function	u 1 10
Maximum Recursion Depth	n 4 2 2 6
Chance of Calling a User Function	n 0.5 0.05 0 1
Chance of Call Site Being Fixed	n 0.999 0.05 0.99 1
Chance of Function Just Returning	n 0.5 0.9 0 1
Coefficients of Compile Equation	u 0 0 n 10 3 5 15 u 100 100

The other parameter that differs between the "many-per-file" and "one-per-file" programs is the number of lines per file. This value has to be adjusted to keep the number of lines *per function* approximately equal for both programs. The "many-per-file" program has an average of 20 functions per source file with an average of 500 lines per file. This gives a value of 25 lines per function (on average). To duplicate this, the number of lines per file for the "one-per-file" program was modified to have an average of 25.

Table 6.23 summarizes the performance of the continuous compilation system on the "many-per-file" and "one-per-file" programs. When we compare the performance of replace-at-call and replace-preemptive, we again see little difference. Once again, as described in Section 6.6, this is most likely an artifact of the way ProGenitor generates the programs. The more interesting comparison is between the performance of the "many-per-file" and "one-per-file" programs themselves. As expected, the "one-per-file" program does perform better than the "many-per-file" program. When using the longest-overall compilation strategy, the speedup is greater than 20%.

Table 6.23: Variation of Functions per Source File

Compilation Strategy	**Replace-at-Call**		**Replace-Preemptive**	
	Many Func. per File	**One Func. per File**	**Many Func. per File**	**One Func. per File**
random	66.580	65.174	66.509	65.107
mfex-so-far	67.308	57.009	67.239	56.886
long-so-far	67.522	55.373	67.363	55.218
small	62.143	58.676	62.033	58.603
large	72.055	70.224	71.932	70.190
mfex-overall	66.503	53.188	66.295	53.127
long-overall	65.151	51.655	65.106	51.616

6.8 Various Interpretation Penalties

Throughout this chapter we have been using an interpretation penalty of 10 for our continuous compiler. While this appears to be a reasonable value to use in light of recent work [19], a quick look at the performance of the continuous compiler with other interpretation penalties would be worthwhile.

Table 6.24: Performance with Various Interpretation Penalties

Program	3x	10x	100x
ghostview	44.841	52.848	76.043
pico	63.536	63.608	64.517
render	137.620	139.956	144.610

Table 6.24 shows the performance of the continuous compiler for `ghostview`, `pico`, and `render` with interpretation penalties of 3, 10, and 100. The longest-overall compilation strategy and the replace-preemptive model were used.

As expected, the continuous compiler performs better with a smaller interpretation penalty and does worse with a larger penalty. The surprise is that the variation in performance is so small. Even when the interpretation penalty is increased by a factor of 10, from 10 times to 100 times, the slowdown of `render` is only about 3%. With `ghostview` the slowdown jumps to about 44%, but that is still remarkable considering that it corresponds to an increase of 900% for the interpretation penalty.

6.9 Summary

So where does all of this leave us? We found, as discussed in Section 6.2, that the continuous compilation model performs well when compared to the traditional compilation model, especially when we take into account the additional benefits provided by the continuous compilation model, such as immediate response time.

We also found, in Section 6.3, that continuous compilation can outperform interpretation by tremendous margins, which is what was expected. However, we also found that as the time spent in library code increases, the gain in performance of continuous compilation over interpretation decreases.

In comparing replacement strategies we found, as expected, that the replace-preemptive method gives better performance than the replace-at-call method. In addition, while the gain in performance might be small for some programs, other programs show a tremendous increase in performance. In Section 6.4 we discuss the significant factors of the program structure that cause the replacement strategy to have such a strong effect.

Choosing a compilation strategy proved more difficult. As discussed in Section 6.5, which strategy performs the best seems to be tightly bound to the structure

of the particular program being executed. The longest-overall strategy gave the best performance most often, but it required previous knowledge about the execution of the program. However, the longest-so-far strategy seems to work fairly well as an estimate to longest-overall.

We then examined the effects of particular program characteristics on performance. In Section 6.6 we reexamined the effects of the library/user code ratio on the continuous compiler. While a high percentage of library code might not give the continuous compiler an advantage over interpretation, it can give the continuous compiler a large advantage over traditional compilation.

Lastly, in Section 6.7 we found that the density of functions per source file does affect performance slightly, but not significantly. If functions could be compiled on an individual basis rather than a conglomerate file basis performance might be improved, but the overhead of dealing with the larger number of files might cancel out any gains.

Chapter 7

Conclusion

The world of personal computing is becoming more connected. The advent of the World Wide Web has caused an explosive growth in the population of Internet users. Everyday new members join the network community. It is changing the way we work, the way we think. Everyday one reads another story about the "global network" and the way it is changing the world. Major computer companies have built entire advertising campaigns around this very idea, and the future promise of bringing it about. Providing on-line services to the home user has become a multi-million dollar business.

The world of computing is becoming more interactive. The days of batch computing are all but gone. We have become used to, and have come to expect, quick response times. Even as we move to a more decentralized, remote model of computing our expectations stay the same. In fact, the spread of graphical interfaces and multimedia has only served to increase our expectations. The effect on the computing world of the graphical, multimedia based nature of the Web stands as testimony to this.

The world of computing is becoming more mobile. Mobile computers are becoming smaller and more powerful everyday. However, people do not want to give up connectivity for mobility.

With the more connected, more interactive, more mobile paradigm comes the need for the rapid transmission of programs from a remote server. The recent and continuing advances in network technology will help alleviate this problem by providing greater bandwidth, but network usage seems to always increase to fill the amount of bandwidth available. In addition, while network technology is improving rapidly, advances in wireless computing still lag far behind that of wire-based media.

We believe that this shift to a more connected, more interactive model of computing is impelling a move to a new paradigm of program translation, namely that of continuous compilation as described in this thesis.

Moving-target computing is not the only possible application of continuous compilation. As software development becomes more interactive in nature, the benefits of a continuous compilation paradigm will become more evident. Visual programming is just one example of a field which could profit from exploring such a paradigm.

The continuous compilation paradigm provides many of the benefits of an interpreted environment while enabling tremendous performance increases over interpretation. In addition, we showed that continuous compilation outperforms a traditional compilation model in many circumstances, even when using a single processor. With a dual-processor system, the benefits of using the continuous compiler are even greater.

We also showed that the performance of the continuous compiler is strongly dependent on the design of various aspects of the compiler. Specifically, the strategy used to replace source code with native code and the order in which the translation from source code to native code is carried out both have a strong affect on performance.

As expected, the replace-preemptive strategy proved to be much more effective than the replace-at-call strategy in most cases, so much so that it would appear to be worthwhile to implement the replace-preemptive strategy in an actual system, even with the extra difficulty involved. One question that still needs to be resolved is exactly how much overhead the replace-preemptive strategy would incur.

The question of order of compilation has proven more difficult. While the longest-overall strategy often produced the best performance, it requires previous knowledge about the execution of the program that is difficult to collect and might not always be available. However, the longest-so-far strategy appears to be a good approximation of longest-overall. It might also be possible to use the history of previous executions to approximate the longest-overall strategy.

Although our pseudo-program generator provided valuable data in these experiments, it turned out to be less effective than was hoped. The ProGenitor program could be made more useful if it was modified so that the program structure generated more closely resembled that of real programs. Allowing the user to specify certain common structures, like an event-loop based program, would also be beneficial.

To summarize, we believe that the continuous compilation paradigm exhibits the following advantages:

1. immediate response time, equivalent to that of interpretation,
2. shorter recovery time than traditional compilation,
3. execution performance which approaches that of traditional compilation over time,
4. smaller distribution file size, and
5. the ability to deliver applications in a target-independent format.

In addition, since the user is no longer waiting for compilation to finish, more time can be spent on optimizing the compiled code.

The next step is, of course, the construction of an actual continuous compilation system. We believe that it is only a matter of time until such systems become commonplace.

Appendix A

Sample Simulator Input and Output Files

The sections below show examples of the input and output files used and produced by our continuous compilation simulator. The meaning and format of these files are described in Chapter 4.

A.1 Sample Setup File

Below is the setup file used for the `render` program. The "main: ..." line has been wrapped for illustrative purposes.

```
MakeStuff

main: render.o parsecmd.o name.o color.o scale.o rotate.o trans.o ambient.o \
diffuse.o spectral.o rdata.o read_geom.o state.o light.o dumpobj.o uedges.o \
planes.o shade.o texture.o fog.o reflective.o Xgraf.o readgif.o showgif.o \
dedges.o dlines.o shades.o dpolys.o vrp.o lookat.o vup.o zplane.o eye.o \
bview.o field.o paint.o hlist.o screen.o background.o area.o scanconv.o \
zbuffer.o trace.o rayshade.o raytrace.o m4dm4d.o cm4d.o pm4d.o p4dm4d.o \
p3dm4d.o rotvec.o rotpln.o m4dinv.o
parsecmd.o:     parsecmd.c
name.o:         name.c
color.o:        color.c
scale.o:        scale.c
rotate.o:       rotate.c
trans.o:        trans.c
ambient.o:      ambient.c
diffuse.o:      diffuse.c
spectral.o:     spectral.c
rdata.o:        rdata.c
read_geom.o:    read_geom.c
```

```
state.o:        state.c
light.o:        light.c
dumpobj.o:      dumpobj.c
uedges.o:       uedges.c
planes.o:       planes.c
shade.o:        shade.c
texture.o:      texture.c
fog.o:          fog.c
reflective.o:   reflective.c
Xgraf.o:        Xgraf.c
readgif.o:      readgif.c
showgif.o:      showgif.c
dedges.o:       dedges.c
dlines.o:       dlines.c
shades.o:       shades.c
dpolys.o:       dpolys.c
vrp.o:          vrp.c
lookat.o:       lookat.c
vup.o:          vup.c
zplane.o:       zplane.c
eye.o:          eye.c
bview.o:        bview.c
field.o:        field.c
paint.o:        paint.c
hlist.o:        hlist.c
screen.o:       screen.c
background.o:   background.c
area.o:         area.c
scanconv.o:     scanconv.c
zbuffer.o:      zbuffer.c
trace.o:        trace.c
rayshade.o:     rayshade.c
raytrace.o:     raytrace.c
m4dm4d.o:       m4dm4d.c
cm4d.o:         cm4d.c
pm4d.o:         pm4d.c
p4dm4d.o:       p4dm4d.c
p3dm4d.o:       p3dm4d.c
rotvec.o:       rotvec.c
rotpln.o:       rotpln.c
m4dinv.o:       m4dinv.c
render.o:       render.c
CtagStuff
AddToPixel   readgif.c    /^AddToPixel(unsigned char Index, struct objects *ob/
ReadCode     readgif.c    /^ReadCode()/
disclear     Xgraf.c      /^void disclear()/
disclose     Xgraf.c      /^void disclose()/
disline      Xgraf.c      /^void disline(struct pnts2d *s, struct pnts2d *e, s/
disopen      Xgraf.c      /^void disopen(struct worlds *world)/
dispixel     Xgraf.c      /^dispixel(int x, int y, struct colors *col)/
dispoly      Xgraf.c      /^void dispoly(int ncnt, struct pnts2d *nodes, struc/
disrawline   Xgraf.c      /^void disrawline(struct pnts2d *s, struct pnts2d *e/
```

```
disrect      Xgraf.c      /^disrect(int xs, int ys, int xw, int yw, struct col/
readgif      readgif.c    /^readgif(char *fname, struct objects *obj)/
setcolor     Xgraf.c      /^void setcolor(struct colors *col)/
showgif      showgif.c    /^void showgif(int cmdc, char **cmdv, struct worlds /
cm4d         cm4d.c       /^void cm4d(float m[4][4])/
m3ddet       m4dinv.c     /^float m3ddet(float m[3][3])/
m4ddet       m4dinv.c     /^float m4ddet(float m[4][4])/
m4dinv       m4dinv.c     /^void m4dinv(float m[4][4], float n[4][4])/
m4dm4d       m4dm4d.c     /^void m4dm4d(float a[4][4], float b[4][4], float m[/
p3dm4d       p3dm4d.c     /^void p3dm4d(struct pnts3d *a, float m[4][4], struc/
p4dm4d       p4dm4d.c     /^void p4dm4d(struct pnts4d *a,float m[4][4], struct/
pm4d         pm4d.c       /^void pm4d(float m[4][4])/
rotpln       rotpln.c     /^void rotpln(struct plane *a, struct pnts4d *p, /
rotvec       rotvec.c     /^void rotvec(struct pnts3d *a, float m[4][4], struc/
ambient      ambient.c    /^void ambient(int cmdc, char **cmdv, struct objects/
color        color.c      /^void color(int cmdc,char **cmdv, struct objects *o/
diffuse      diffuse.c    /^void diffuse(int cmdc, char **cmdv, struct objects/
dumpobj      dumpobj.c    /^void dumpobj(struct objects *top, char *str)/
ecmp         uedges.c     /^int ecmp(struct lines *a, struct lines *b)/
fog          fog.c        /^void fog(int cmdc,char **cmdv, struct worlds *worl/
light        light.c      /^void light(int cmdc, char **cmdv, struct lights **/
name         name.c       /^void name(int cmdc, char **cmdv, struct objects *o/
planes       planes.c     /^void planes(struct objects *obj)/
plights      light.c      /^void plights(struct lights *obj)/
rdata        rdata.c      /^void rdata(int cmdc, char **cmdv, struct objects */
read_geom    read_geom.c  /^void read_geom(FILE *fd, struct objects **top, int/
reflective   reflective.c /^reflective (int cmdc, char **cmdv, struct objects /
rg_err       read_geom.c  /^rg_err(str)/
rotate       rotate.c     /^void rotate(int cmdc, char** cmdv, struct objects /
scale        scale.c      /^void scale(int cmdc, char **cmdv, struct objects */
shade        shade.c      /^void shade(int cmdc, char **cmdv, struct worlds *w/
spectral     spectral.c   /^void spectral(int cmdc, char **cmdv, struct object/
state        state.c      /^void state(int cmdc, char **cmdv, struct objects */
texture      texture.c    /^void texture(int cmdc,char **cmdv, struct objects /
trans        trans.c      /^void trans(int cmdc, char **cmdv, struct objects */
uedges       uedges.c     /^void uedges(struct objects *obj)/
area         area.c       /^void area(struct worlds *world)/
areasub      area.c       /^void areasub(struct hlists **list, int cnt, int xl/
asplit       area.c       /^void asplit(struct hlists **list, int cnt, int xl,/
background   background.c /^void background(int cmdc, char *cmdv[], struct wor/
bview        bview.c      /^void bview(struct worlds *world)/
closest      area.c       /^int closest(float x, float y, struct hlists **list/
cmp          paint.c      /^int cmp(struct hlists *a, struct hlists *b) /
dedges       dedges.c     /^void dedges(struct worlds *world)/
dlines       dlines.c     /^void dlines(struct worlds *world)/
dpolys       dpolys.c     /^void dpolys(struct worlds *world)/
drawseg      scanconv.c   /^void drawseg(int y, struct edges *s, struct edges /
esort        scanconv.c   /^void esort(struct edges **pnt)/
eye          eye.c        /^void eye(int cmdc, char *cmdv[], struct worlds *wo/
field        field.c      /^void field(int cmdc, char *cmdv[], struct worlds */
freehlist    hlist.c      /^freehlist(struct hlists *tmp, int cnt)/
hlist        hlist.c      /^void hlist(struct worlds *world, struct hlists **t/
```

```
lookat       lookat.c     /^void lookat(int cmdc, char *cmdv[], struct worlds /
paint        paint.c      /^void paint(struct worlds *world)/
rayshade     rayshade.c   /^void rayshade(struct rays *ray, struct worlds *wor/
raytrace     raytrace.c   /^void raytrace(struct rays *ray, struct worlds *wor/
scanconv     scanconv.c   /^scanconv(struct hlists *poly, unsigned long *zbuf,/
screen       screen.c     /^void screen(int cmdc, char *cmdv[], struct worlds /
shades       shades.c     /^void shades(struct worlds *world, int type, struct/
trace        trace.c      /^void trace(struct worlds *world)/
vrp          vrp.c        /^void vrp(int cmdc, char *cmdv[], struct worlds *wo/
vup          vup.c        /^void vup(int cmdc, char *cmdv[], struct worlds *wo/
zbuffer      zbuffer.c    /^zbuffer(struct worlds *world)/
zcmp         zbuffer.c    /^int zcmp(struct hlists *a, struct hlists *b) /
zplanes      zplane.c     /^void zplanes(int cmdc, char *cmdv[], struct worlds/
Mrender      render.c     /^void main(int argc, char *argv[])/
help         render.c     /^void help()/
parse        render.c     /^void parse(FILE *fd, struct worlds *world)/
parsecmd     parsecmd.c   /^void parsecmd(char *str, int *cmdc, char ***cmdv)/
CompileStuff
parsecmd.c     28   700
name.c         46   600
color.c        57   700
scale.c        33   600
rotate.c       53   800
trans.c        33   600
ambient.c      58   700
diffuse.c      58   700
spectral.c     59   700
rdata.c        26   500
read_geom.c   152  1500
state.c        35   600
light.c       118   900
dumpobj.c      45   700
uedges.c       91   800
planes.c       77  1200
shade.c        25   600
texture.c      50   700
fog.c          31   600
reflective.c   51   700
Xgraf.c       202  2200
readgif.c     295  3000
showgif.c      50   800
dedges.c       62   600
dlines.c       43   500
shades.c       59   800
dpolys.c       64   600
vrp.c          18   600
lookat.c       18   500
vup.c          18   600
zplane.c       17   600
eye.c          18   600
bview.c       105   900
field.c        16   600
```

paint.c	41	500
hlist.c	131	1500
screen.c	17	500
background.c	18	600
area.c	256	2100
scanconv.c	187	1600
zbuffer.c	60	800
trace.c	67	800
rayshade.c	355	2200
raytrace.c	81	1000
m4dm4d.c	31	1000
cm4d.c	13	400
pm4d.c	12	400
p4dm4d.c	20	500
p3dm4d.c	18	500
rotvec.c	29	700
rotpln.c	30	600
m4dinv.c	135	1700
render.c	250	1600

A.2 Sample Behavior File

Below are the first 10 and last 10 lines of the render behavior file. The actual file is 2,076,693 lines long.

```
I 473 _start
I 670 _init
O 672
I 674 __do_global_ctors_aux
O 678
I 681 main
I 3649 cm4d
O 3652
I 17962 parse
I 45914 parsecmd
.
.
.
I 130959856 parsecmd
O 130959880
O 130959889
O 130959894
I 130959899 exit
I 130959986 __do_global_dtors
O 130959989
I 130959992 _fini
O 130959994
O 130960150
```

A.3 Sample Order File

Below is the optional order file used to simulate the longest-overall compilation strategy with the `render` program.

```
raytrace.o
Xgraf.o
render.o
rayshade.o
readgif.o
read_geom.o
bview.o
planes.o
uedges.o
trace.o
hlist.o
color.o
parsecmd.o
shades.o
m4dinv.o
scale.o
vrp.o
eye.o
field.o
screen.o
p4dm4d.o
rdata.o
light.o
paint.o
reflective.o
diffuse.o
texture.o
dumpobj.o
p3dm4d.o
lookat.o
zplane.o
name.o
dedges.o
background.o
cm4d.o
trans.o
state.o
shade.o
vup.o
rotpln.o
fog.o
dpolys.o
m4dm4d.o
area.o
rotate.o
showgif.o
pm4d.o
```

```
rotvec.o
ambient.o
spectral.o
dlines.o
scanconv.o
zbuffer.o
```

A.4 Sample Output File

Below is a somple output file for the render program. It was produced using the largest-first compilation strategy, the replace-at-call replacement strategy, and an interpretation penalty of 10.

```
Compilation strategy: largest first
Interpretation penalty: 10   Discreet replacement
Execution time: 150793432
Time spent in
     Interpreted code: 22036980
     Non-library compiled code: 128738162
     Library code (compiled): 18290
     Code of unknown status: 0

Compiler finished!
Time spent compiling: 46800000
(All times are in microseconds)

Library functions:
Name                            Entry count     Accumulated time
__do_global_ctors_aux           1               4
_start                          1               207
_init                           1               2
_fini                           1               2
main                            1               17353
__do_global_dtors               1               3
exit                            1               246

Non-library functions:
Name            Entry count     Comp. time      Int. time       Final status
reflective      1               0               790             Compiled
freehlist       0               0               0               Compiled
screen          1               0               890             Compiled
background      1               0               1020            Compiled
disopen         1               146884          0               Compiled
spectral        0               0               0               Compiled
eye             1               0               730             Compiled
fog             0               0               0               Compiled
showgif         0               0               0               Compiled
disclose        0               0               0               Compiled
m4dm4d          11              22              30              Compiled
```

p3dm4d	5	0	50	Compiled
p4dm4d	74	0	1040	Compiled
parsecmd	27	24	14230	Compiled
rotate	0	0	0	Compiled
drawseg	0	0	0	Compiled
rayshade	176773	5987955	0	Compiled
trace	1	3881509	0	Compiled
parse	2	6916596	1240810	Compiled
areasub	0	0	0	Compiled
read_geom	1	0	2173150	Compiled
cmp	0	0	0	Compiled
vrp	1	0	1890	Compiled
vup	1	0	770	Compiled
disrawline	0	0	0	Compiled
ambient	0	0	0	Compiled
closest	0	0	0	Compiled
hlist	1	1505	0	Compiled
lookat	1	0	650	Compiled
m4dinv	1	99	0	Compiled
bview	1	23726	0	Compiled
setcolor	160000	50878606	0	Compiled
rg_err	0	0	0	Compiled
zplanes	1	0	750	Compiled
plights	0	0	0	Compiled
uedges	4	0	13260	Compiled
disclear	0	0	0	Compiled
color	4	0	4070	Compiled
dedges	0	0	0	Compiled
dispixel	160000	2601703	0	Compiled
rotpln	13	0	2280	Compiled
area	0	0	0	Compiled
AddToPixel	256000	1831590	0	Compiled
cm4d	14	0	360	Compiled
pm4d	0	0	0	Compiled
name	4	0	620	Compiled
dumpobj	0	0	0	Compiled
readgif	1	0	18283540	Compiled
field	1	0	560	Compiled
disline	0	0	0	Compiled
diffuse	0	0	0	Compiled
scale	1	0	1770	Compiled
shade	1	0	70	Compiled
state	0	0	0	Compiled
texture	3	0	2780	Compiled
raytrace	233148	55699851	0	Compiled
rotvec	0	0	0	Compiled
ReadCode	51982	767737	0	Compiled
rdata	1	0	104960	Compiled
ecmp	218	0	4330	Compiled
zcmp	0	0	0	Compiled
help	0	0	0	Compiled
dispoly	0	0	0	Compiled

disrect	0	0	0	Compiled
asplit	0	0	0	Compiled
esort	0	0	0	Compiled
scanconv	0	0	0	Compiled
m3ddet	20	87	0	Compiled
m4ddet	1	23	0	Compiled
light	1	0	1650	Compiled
paint	0	0	0	Compiled
Mrender	0	0	0	Compiled
planes	4	0	179930	Compiled
trans	0	0	0	Compiled
shades	13	245	0	Compiled
zbuffer	0	0	0	Compiled
dlines	0	0	0	Compiled
dpolys	0	0	0	Compiled

Files not compiled:

Order of compilation:

Name	Compilation completed at
readgif.o	3000000
Xgraf.o	5200000
rayshade.o	7400000
area.o	9500000
m4dinv.o	11200000
render.o	12800000
scanconv.o	14400000
hlist.o	15900000
read_geom.o	17400000
planes.o	18600000
m4dm4d.o	19600000
raytrace.o	20600000
bview.o	21500000
light.o	22400000
rotate.o	23200000
uedges.o	24000000
trace.o	24800000
zbuffer.o	25600000
shades.o	26400000
showgif.o	27200000
color.o	27900000
reflective.o	28600000
rotvec.o	29300000
parsecmd.o	30000000
ambient.o	30700000
spectral.o	31400000
diffuse.o	32100000
texture.o	32800000
dumpobj.o	33500000
zplane.o	34100000
name.o	34700000
dedges.o	35300000

background.o	35900000
scale.o	36500000
trans.o	37100000
state.o	37700000
shade.o	38300000
vrp.o	38900000
vup.o	39500000
rotpln.o	40100000
fog.o	40700000
dpolys.o	41300000
eye.o	41900000
field.o	42500000
screen.o	43000000
p4dm4d.o	43500000
p3dm4d.o	44000000
lookat.o	44500000
rdata.o	45000000
paint.o	45500000
dlines.o	46000000
pm4d.o	46400000
cm4d.o	46800000

Appendix B

Sample ProGenitor Input File

Below is the input file used to generate the "equal" program. Note that in the actual input file, the comments on the right are not allowed. They were added for illustrative purposes only.

```
73984                   # Random seed
40000000                # Cutoff time
u 20 20                 # Number of source files  *** User File Parameters ***
u 10 10                 # Number of header files
n 5 5 1 10              # ''Popularity'' of header files
n 5 1 1 10              # Number of functions per source file
n 500 200 20 1500       # Number of lines per source file
u 50 500                # Spin time between function calls
u 1 10                  # Number of call sites per function
n 4 2 2 6               # Maximum nest depth
n 0.5 0.05 0 1          # Chance of calling another user function
n 0.999 0.05 0.99 1     # Chance of call site being fixed
n 0.5 0.9 0 1           # Chance of function just returning
u 0 0                   # Compile time coefficient a
n 10 3 5 15             # Compile time coefficient b
u 100 100               # Compile time coefficient c
u 20 20                 # Number of source files *** Library File Parameters ***
u 10 10                 # Number of header files
n 5 5 1 10              # ''Popularity'' of header files
n 5 1 1 10              # Number of functions per source file
n 500 200 20 1500       # Number of lines per source file
u 50 500                # Spin time between function calls
u 1 5                   # Number of call sites per function
n 4 2 2 6               # Maximum nest depth
n 0.0005 0.05 0 0.002   # Chance of calling a user function
n 0.999 0.05 0.99 1     # Chance of call site being fixed
n 0.5 0.9 0 1           # Chance of function just returning
u 0 0                   # Compile time coefficient a
n 10 3 5 15             # Compile time coefficient b
u 100 100               # Compile time coefficient c
0                       # Do not output library information
```

References

[1] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communication*, 2(6), December 1995.

[2] A.P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[3] S. Feldman. Make – A program for maintaining computer programs. *Software Practice and Experience*, 9:255–265, 1979.

[4] Alan R. Feuer. si—an interpreter for the C language. *Proceedings of the 1995 Usenix Summer Conference*, 1985. Portland, OR.

[5] Christopher Fraser and David Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings, 1995.

[6] S. French. *Sequencing and Scheduling.* Ellis Horwood Limited, 1982.

[7] K.J. Gough, C. Cifuentes, D. Corney, J. Hynd, and P. Kolb. An experiment in mixed compilation/interpretation. *Australian Computer Science*, 14(1), 1992.

[8] Martin L. Griss, Eric Benson, and Anthony C. Hearn. Current status of a portable Lisp compiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 276–283, June 1982.

[9] Urs Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, 1994. Report STAN-CS-TR-94-1520.

[10] Tomasz Imielinski and B.R. Badrinath. Mobile wireless computing. *Communications of the ACM*, 37(10), October 1994.

[11] Borland International. Interview with régis crelier. http://www.borland.com. 100 Borland Way, Scotts Valley, CA.

[12] Uwe Kastens. Generating interpreters automatically from compiler specifications. Technical report, Paderborn University, August 1994. Report tr-ri-94-151.

[13] Randy H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1), 1994.

[14] D.E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1:105–133, 1971.

[15] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. *Proceedings of the SIGPLAN '95 Conference on Programming Langauge Design and Implementation (Sigplan NOTICES)*, 30(6), 1995.

[16] Henry Ledgard and Michael Marcotty. *The Programming Language Landscape.* Science Research Associates, 1981.

[17] Metricom. A brief guide to wireless data solutions. http://www.metricom.com. 980 University Avenue, Los Gatos, CA.

[18] Sun Microsystems. Sun offers a cupful of... hotjava. *Network world*, 12(21), May 1995.

[19] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, January 1995.

Vita

Michael P. Plezbert

Date of Birth	December 5, 1969
Place of Birth	Springfield, Missouri
Degrees	B.S. Magna Cum Laude, Physics, May 1993, from Southwest Missouri State University.

May 1996

Short Title: Continuous Compilation Plezbert, M.Sc. 1996