Does "Just in Time" = "Better Late than Never"? *

Michael P. Plezbert and Ron K. Cytron†
Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

Abstract

The World-Wide Web is emerging as a medium for distributing platform-independent, intermediate-form programs. Most Java vendors have recently announced plans to construct "just-in-time" systems, which translate the intermediate text into native code on demand. In this paper, we present experiments that show the benefits of just-in-time systems as compared with the traditional (compile prior to execution) systems. We introduce a new method—the *continuous compiler*—that can outperform just-in-time systems by overlapping compilation with program interpretation and native execution. Based on those results, we then present a *smart* just-in-time system that blends interpretation with native-code execution, thereby obtaining improved performance.

1 Introduction

With the advent of the World-Wide Web has come the desire to distribute programs in a machine-independent form, suitable for execution on a wide variety of platforms. The Java virtual machine (Java VM) has emerged as a vehicle for executing Web-distributed programs: most browsers currently offer a mechanism for interpreting Java VM code. While other such platforms are under development, it seems clear that program distribution on the Web requires initial transmission of an interpreted, intermediate-language format; the downside to this approach is that a performance penalty must be paid for interpretation, as compared with native-code execution.

*This work is supported by the National Science Foundation, under grant CCR 9402883.
†Contact author: cytron@cs.wustl.edu

Submitted to the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997.

To remedy performance problems, vendors such as Borland [7], Sun [14], Symantec[3], and others are implementing systems that transform interpreted Java VM code into native form "just-in-time": a class (compilation unit) is transmitted in Java VM form, but interpretation never takes place. Instead, the portable VM code is compiled to native code when first needed by the executing program. While the just-in-time approach avoids the penalty of interpretation, our experiments show that the cost of compilation can significantly interrupt the flow of execution; furthermore, in many cases, better performance could be obtained by interpreting the original form rather than compiling to native code.

In this paper, we study the performance of three kinds of systems that mix, to varying degrees, interpreted and native-code forms. The goal of this study is to identify an approach that allows distribution of platform-independent programs in a (perhaps mobile) Web-like setting, while minimizing the overhead experienced by the user.

Consider an example where a method (or procedure or routine) A is about to call another method B, where A and B arise from separate compilation units.[1] In this paper we study the following three systems:

Just in time (JIT): As described above, this system transmits code in interpretive format, but compiles code to native form on demand. This system uses a single thread of control that alternates between compilation into native-code form and native execution of the compiled object code. In our example, JIT requires that A is already available in native-code form. If native code for B has not already been generated, then the system appears to pause when B is first invoked so that code can be generated for B. Section 2 describes these kinds of systems in greater detail and presents the results of our experiments for JIT.

[1]We assume a compilation unit is also the basis for transmission on the Web; such is the case for Java classes.

Program	Lines of Code	Compile Time (elapsed, in ms)
ghostview	93029	75100
pico	53443	66200
render	53724	69000

Figure 1: Compile-time statistics for our study programs. "Lines of Code" was measured *after* preprocessing.

Continuous compilation (Contcom): In this system—of our own design—interpretation, compilation to native-code form, and native-code execution are intertwined. JIT begins translation to native-code form when B is first invoked; the goal with continuous compilation is to have the native-code form available *by the time* the call to B occurs. If object code for B is not yet available, then execution can proceed (albeit more slowly) with the interpreted form of B. This system essentially uses two threads of control: one thread compiles interpreted code into native-code form, while the other thread handles program execution of the interpreted and compiled forms.

Smart JIT: This system is effectively a compromise between the above two systems. While limited to only a single thread of control, we allow the smart JIT system to execute interpreted or native code. This system can also elect to pause execution at any moment to compile an interpreted compilation unit into native-code form. Thus, when A invokes B, execution of B can proceed in interpreted form, or the system could compile B to native code. The difficulty lies in deciding *when* a given compilation unit should be translated to native-code form, if ever. As we discuss in Section 4, profiling information can greatly improve the precision of such decisions, but the more interesting case is when the choices must be made on-the-fly. In Section 4 we present a model for making such choices, based on experiments we have conducted on actual programs.

Our paper is organized as follows: Sections 2, 3, and 4 describe the above systems in greater detail and present the results of our experiments on such systems. Section 5 explains how these studies were conducted. Section 6 examines previous work on related topics, and Section 7 presents our conclusions.

2 Just-in-time

In this section, we examine the properties of a just-in-time system in greater detail, and we present the results of experiments that quantify the effectiveness of such systems as compared with the traditional compile-link-execute cycle. Just-in-time systems operate by obtaining native code on demand, thus avoiding the overhead associated with compiling all units prior to execution. As such, they obtain two key benefits:

1. Large programs often contain substantial portions of code that are not executed in a typical run of the program. For example, a Web browser may contain routines that handle various formats of transmitted data, but some of these routines may not be used in a given run of the browser. Because the JIT systems translate only those compilation units that contain actually executed code, the overhead of compiling unused code is avoided.

2. Traditional systems performs all compilation prior to execution, which can present the user with a substantial time gap between the moment all compilation units have been transmitted and the moment execution can begin. The JIT system does not translate a compilation unit until it is first referenced by the running program. This approach tends to amortize compilation time throughout a program's execution. While the system would appear to lapse whenever compilation occurs, the effect would be similar to the pauses present in a nonconcurrent garbage collected system.

Our study was conducted on a set of programs with diverse characteristics yet representative of the kinds of programs one can expect to distribute across the Web. These programs were written in C: there is as yet no set of large Java applications available for study; moreover, it is likely that Web-based systems will support languages such as C in the near future [1]. Specifically, our study involved 5 program executions of three applications as detailed below:[2]

pico is a widely used text editor, similar to emacs but scaled down in scope. Our copy was obtained from the University of Washington and represents version 2.4 of pico. We ran pico in two settings:

[2]The two runs each of pico and ghostview demonstrate the variation among runs of the same application.

Program	Compile Time	Execution Time		Speedup	Pauses		
		User	Library		Min.	Max.	Average
ghostview1	51500	37975.7	1.1	1.26	600	10300	6437.5
ghostview2	73300	91607.8	1.5	1.01	600	10300	6108.3
pico1	30800	75.1	63439.9	1.24	700	8900	2566.7
pico2	33300	3436.1	30205.6	1.30	700	8900	2561.5
render	48400	130959.2	0.9	1.11	600	4500	1382.9

(times shown in milliseconds)

Figure 2: Just-in-time results. In comparison with Figure 1, the compile times shown here omit those compilation units *not* touched by the JIT system.

- **pico1** was a simple text-entering session lasting approximately one minute, with a human providing the input keystrokes at a rate typical for graduate students.
- **pico2** was a spell-checking session for a document of 30 words.

ghostview is a widely used postscript viewer available from the Gnu software archive. We used version 1.5 in the following two settings:

- **ghostview1:** Ghostview was provided a 92 Kbyte input file as a command-line argument.[3] Each of the document's 14 pages was viewed for a few seconds, and then ghostview was exited.
- **ghostview2:** Ghostview was invoked without an input file. The same input file used for ghostview1 was then chosen using the mouse and a menu. Each page was previewed and one section was enlarged by zooming.

render accepts an input file that directs the ray tracing of a very simple scene (house with texture and image mapping, single light source).

Summary compile-time statistics are shown in Figure 1. Throughout this paper, the times we report are *elapsed* times, not CPU times, because we are interested in the delays actually experienced by the user. The times were measured on a relatively quiescent SPARCstation 5.

We now examine the JIT properties of the 5 runs under study. Figure 2 summarizes the effects of JIT, showing the compile-time of those compilation units that required translation. Note that execution time is separated into "user" and "library" time. The "user" time accounts for time spent in the source files of the program's distribution; the "library" time is spent elsewhere, such as in the I/O library, math library, etc. We assume that native code is always available for library routines prior to program execution.

[3]In fact, we used `ghostview.ps`, the documentation file included with ghostview.

The speedup for the JIT system shown in Figure 2 is computed as

$$\frac{\text{compile time for all source files} + \text{execution time}}{\text{compile time for JIT-translated files} + \text{execution time}}$$

which reflects the performance gained by the JIT system over the traditional compile-link-execute cycle. Figure 2 also shows the minimum, maximum, and average pauses experienced by the user when the system suspended execution to translate a compilation unit to native code.

3 Continuous compiler model

In this section we outline the design of our continuously compiling system and compare its performance with the JIT compiler. The results we obtain then motivate the smart JIT system we present in Section 4.

Our continuous compilation model is shown in Figure 3. The Interpreter module is responsible for executing the program while the Compiler module is responsible for translating the interpreted 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; while this is not a requirement, we assume such a system for the purposes of this paper unless otherwise stated. The pieces of the compiler function as follows:

Code: contains a mixture of interpreted- and native-code versions of a program's procedures. At start-up, all of the "user" code is assumed to be in interpreted 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

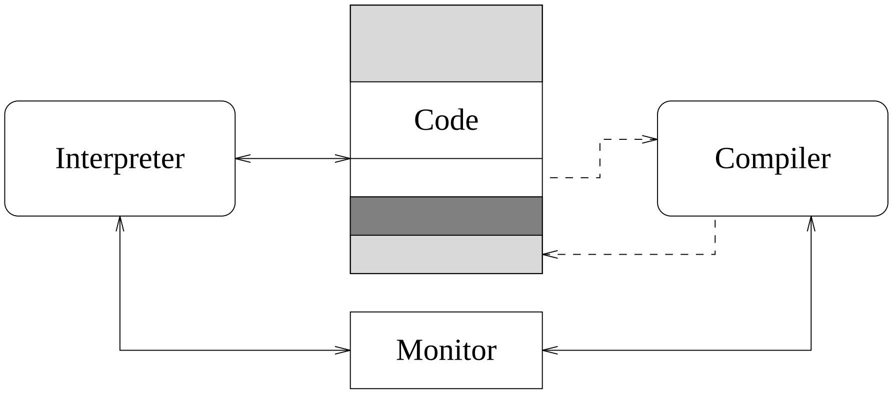

Figure 3: The Continuous Compiler

procedures. Not until a procedure has been fully translated to native code does the native-code version become available to the Interpreter.

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

Interpreter: is responsible for the actual execution of the program. It starts by interpreting the transmitted code, making jumps to the native-code version of procedures as they become available. Our system can accommodate two replacement strategies:

replace-at-call jumps to the native code version of a function only at the time that function is called;

replace-preemptive jumps to the native code as soon as it becomes available, even while the function is active.

Replace-preemptive is more effective, though implementation of this strategy is more difficult. [12] For the purposes of our study here, we adopt the replace-preemptive strategy.

Monitor: The Compiler and Interpreter modules communicate through the Monitor shared data structure, which contains information needed by both.

The compilation thread of our continuous compiler is able to translate compilation units into native-code form one at a time. Our experiments show that a program's execution time under continuous compilation is strongly affected by the order in which the routines are translated into native code. [12] Here, we focus only on the strategy we call "longest-so-far", in which the compilation unit chosen by the Compiler module contains the methods that have collectively consumed the greatest amount of interpreted execution time so far.

Compilation takes place on a file-based level of granularity, not on the routine level. 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, our longest-so-far strategy picks the file with the most accumulated execution time, considering *all* of the routines defined in the file, which is not necessarily the file that contains the single most time-consumptive routine.

Using the same 5 runs of the application programs discussed in Section 2, we obtained the data shown in Figure 4 for the continuous compilation system. The compile times shown in Figure 4 comprise only those compilation units chosen by the Compiler module for translation. Unlike JIT, the continuous compiler can translate a compilation unit that is never actually needed by a program run. While this seems inefficient, recall that the continuous compiler attempts to have native-form code available by the time it is needed; we therefore cannot wait until the point of first reference, as JIT does, if we are to mask the compilation time completely. Figure 4 also shows how program execution time is divided among interpreted and native-code forms for user files, as well as the time spent in the (native-code) library files. In this paper, interpretation time is assumed to be 10 times slower than native-code execution.

In analyzing the performance of the continuous compiler, it may seem unfair to compare its times with those of the other systems we study, since the continuous compiler overlaps compile time and execution time using two processors while the other systems we study use only one. On the other hand, dual-processor worksta-

Program	Compile Time	User Interp.	User Exec.	Lib. Exec.
ghostview1	52500	17178.6	36257.8	1.1
ghostview2	75100	37911.3	87816.7	1.5
pico1	54900	133.3	61.8	63439.9
pico2	45000	17725.0	1663.6	30205.6
render	69000	17228.2	129236.4	0.9

(times shown in milliseconds)

Figure 4: Results using continuous compilation, where interpretation is assumed to be 10 times slower than native-code execution.

Program	Speedup		
	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$
ghostview1	2.12	1.41	1.06
ghostview2	1.33	1.03	0.66
pico1	2.04	1.52	1.02
pico2	2.01	1.35	1.00
render	1.36	1.13	0.68

Situations:

$\boxed{1}$ The other system has only one processor, of the same speed as one of the processors used by the continuous system.

$\boxed{2}$ The other system has the same dual-processor configuration used by the continuous system. The other system can use both processors to accomplish the compiles; execution then proceeds with just a single processor.

$\boxed{3}$ The other system has a single processor whose speed is twice that of the processor used in the continuous system. Because the twice-as-fast processor could emulate the dual-processor system, this configuration is interesting in that it compares the two methods on "equal" hardware.

Figure 5: Comparison of continuous with traditional systems.

tions are becoming increasingly more affordable, and continuous compilation puts the second processor to good use. In Figure 5, we examine performance separately for a variety of situations concerning the resources associated with the other systems.

If compile time is excluded, JIT will always outperform the continuous compiler, because the former never interprets while the latter mixes interpreted and native-code forms. However, the continuous compiler never suffers the "pauses" associated with JIT compilation; moreover, interpretation of some compilation units can be faster than compiling and executing them in native-code form. For example, Figure 6 shows the interpretation penalty for which it would be optimal to translate render's compilation units to native code prior to execution. Of the 35 compilation units, only 7 of them (shown above the first horizontal line) merited compilation to native code prior to execution, given an interpretation penalty of 10. If the penalty is raised to 100, then only 2 more join the "should compile" set. The JIT system spent time compiling all of the files shown in Figure 6, yet performance would be better if most of these were interpreted. Clearly, if compiled and interpreted forms can mix, a more clever approach can improve on JIT, even if only a single processor is used. Section 4 examines such a system.

4 Smart Just-in-time

In this section we use the results of the previous sections to fashion a better approach to just-in-time, based on the insight that, given a single thread of control, it is best either to

1. translate a compilation unit to native code and avoid interpretation altogether, or
2. avoid compilation altogether and interpret the compilation unit throughout program execution.

raytrace.o	1	Xgraf.o	1	trace.o	1
render.o	1	rayshade.o	1	readgif.o	2
read_geom.o	10				
bview.o	56	planes.o	84		
rdata.o	106	parsecmd.o	692	uedges.o	570
hlist.o	1330	color.o	2704	texture.o	3598
shades.o	4899	rotpln.o	4387	vrp.o	4762
scale.o	5086	p4dm4d.o	6732	background.o	7844
light.o	7880	screen.o	8990	m4dinv.o	7657
vup.o	11689	zplane.o	12001	eye.o	12330
reflective.o	11393	lookat.o	13847	name.o	20969
field.o	14287	cm4d.o	11112	m4dm4d.o	48001
shade.o	128570	p3dm4d.o	140000		

Figure 6: Interpretation penalty needed to justify translation to native code for render. The groups show routines that should be compiled if the penalty is 10, 100, 1000, 10000, and greater than 10000.

In other words, if a file is going to be compiled, it might as well happen prior to any interpretation, so that the benefit of native code performance can be enjoyed throughout program execution. The difficulty lies in determining which of the above approaches to take with a given compilation unit.

Like the JIT system of Section 2, the smart JIT system uses only a single thread of control, but it is able to do interpretation as well as compilation and native-code execution.

Based on the results we presented in Sections 2 and 3, it is clearly not in the best interest of program performance to translate *every* compilation unit to native code. For a given compilation unit, consideration of compilation time as well as its relative execution time should be taken into account to decide whether compilation to native code is warranted. Figure 7 contains a point for each compilation unit studied in this paper. The plot's x-axis indicates compilation time. The y-axis indicates the *crossover point*: the interpretation penalty for which compilation to native code yields better performance than interpretation. This is computed as follows: Let T be the native-code execution time spent in a compilation unit in a traditional (no interpretation) setting, and let C be the time to compile the unit. Then the crossover point P is computed as the value for which

$$T \star P > C + T \qquad (1)$$

In other words, we compute the value of P for which compilation followed by native-code execution takes less time than interpretation.

For those points plotted near the (0,0) origin of Figure 7, compilation is clearly preferable: such routines take little compilation time, and the interpretation penalty justifying compilation is also low. For a modest interpretation penalty of 10, Figure 7 demonstrates that most compilation units are easily differentiated between the "should-compile" and "should-interpret" categories. However, the data of Figure 7—in particular, the crossover point—requires first running the program to obtain execution time profiles for the compilation units. At issue is whether one can predict *on-the-fly* when a given compilation unit should be compiled, if at all.

In this section, we compare the following strategies for deciding compilation, with the results shown in Figure 8:

Optimal: For comparison purposes we assume perfect information concerning the time spent in each compilation unit.

JIT and **Contcom:** We reprise data from Sections 2 and 3 for comparison.

Threshold: We demand compilation if a compilation unit has exceeded 0.1 seconds of interpretation time. Figure 8 shows this approach to be poor, primarily because it is *too eager* to compile: most of the routines should have remained interpreted, because the relatively long compile times did not justify translation.

Crossover: We demand compilation if the interpretation time accumulated so far places the compilation unit above its crossover point shown in Figure 7. Specifically, we hold P fixed and rearrange Equation (1) to obtain the interpretation time threshold I beyond which compilation should take place: $I > \dfrac{P \star C}{P - 1}$. For our experiments, P was valued as 10.

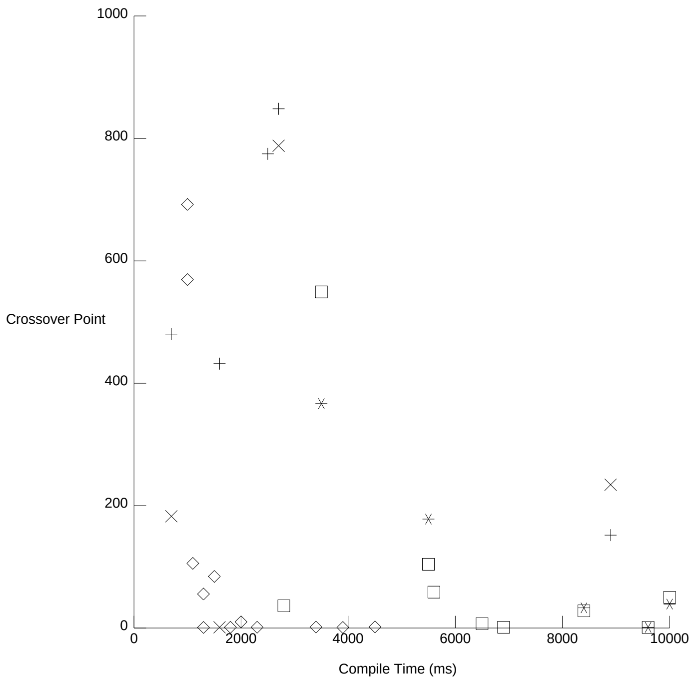

Figure 7: Plot of compilation time vs. penalty factor suggesting compilation prior to execution. Legend: + is pico1, X is pico2, ⋆ is ghostview1, □ is ghostview2, ⋄ is render

Balance: We demand compilation whenever a compilation unit's accumulated interpretation time has exceeded its predicted compile time. With this approach we can bound the amount of overhead to the compile time.

Figure 8 shows results for the above. These results show that JIT performs up to 50% worse than Optimal, while Contcom is almost always better than Optimal. To be fair, Contcom is using two processors; however, Contcom lacks the *a priori* knowledge concerning program behavior given to Optimal. For the three comparisons shown for smart JIT, Threshold performs very poorly, which can be attributed to poor compilation choices. This is unfortunate, since Threshold is probably the easiest scheme to implement. However, Crossover and Balance beat standard JIT in every case, approaching Optimal in two cases. Balance works slightly better than Crossover, primarily because the same compilation units are chosen earlier for compilation.

4.1 Estimation of compilation time

In the Web setting, a compilation unit's compile time on a given platform may well be unknown, yet the method we present in this section requires such information to function effectively. Where compile time is unknown, this section describes an approach to *predict* compilation time given other easily observable features of a compilation unit. For example, character-, word-, and line-counts for a compilation unit could easily be computed as intermediate-text programs are received for interpretation.

Figure 9 plots compilation time as a function of file size, where the file size is measured with respect to the *output* of the C preprocessor. This is important, since some source files draw heavily on large header files, which can impact compile time significantly. Even after accounting for preprocessing, each application contained files with aberrant compile times. Closer scrutiny revealed that those had unusually long procedures. The curve shown in Figure 9 is the closest-fit quadratic for all data shown:

$$y = 0.000066x^2 + 1531 \tag{2}$$

A much better fit can be obtained by considering each application separately. However, the Crossover and Balance schemes do not require precision to function effectively.

5 Methodology

For the purpose of obtaining empirical data on the possible benefits of the systems we describe in this paper,

Strategy	ghostview1	ghostview2	pico1	pico2	render
Optimal	57159	126541	64191	35693	150652
JIT	89477	164909	94315	66942	179360
Contcom	53437	125729	63635	49594	146466
Threshold	82326	158097	72650	44430	155027
Crossover	66759	149541	64191	37293	167853
Balance	65799	147241	64191	37133	167977

Figure 8: Strategies

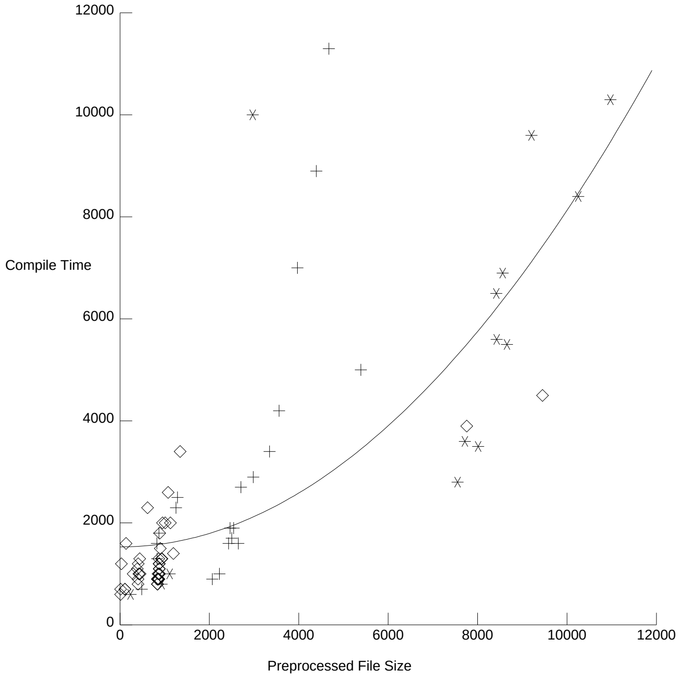

Figure 9: Compilation time (ms) vs. preprocessed file size (lines). Legend: + is pico, ⋆ is ghostview, ⋄ is render.

we constructed a simulator. This simulator accepts as input static information about a particular program along with a trace of its execution. It then *simulates* execution of the program using the JIT, continuous, or smart JIT systems described in this paper.

As shown in Figure 10, the simulator is designed to accept information about either real or synthetic programs. In this paper, we report on our experiments with real programs. Experiments on synthetic programs can be found in [12]. 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 below.

5.1 Simulator Input

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.

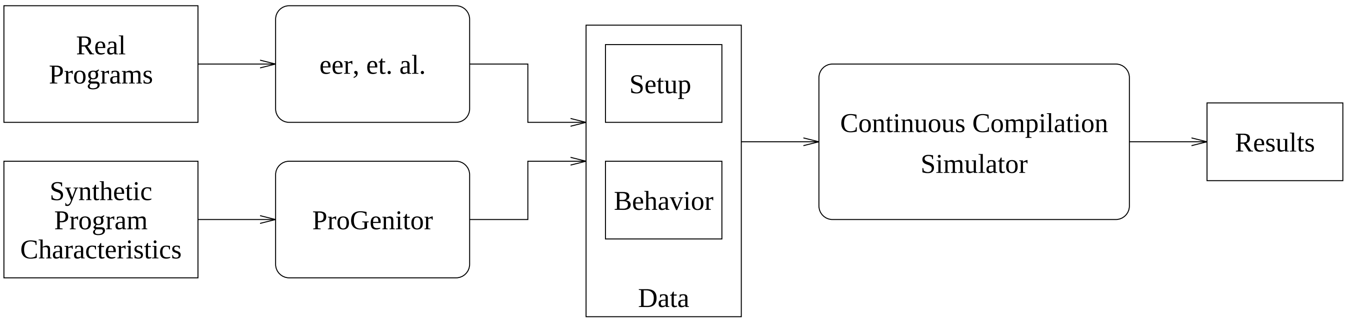

Figure 10: Overview of the Simulator

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 the continuous system described in Section 3. 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 11: Sample Setup File

Figure 11 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.[4]

Figure 12 shows a (very) small portion of a 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.

```
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
```

Figure 12: Sample Behavior File

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

5.2 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. The Interpreter and Compiler modules are modeled as separate objects, 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 shared data structure (the "Monitor" in Figure 3) contains the compilation status of each function, *i.e.*, whether the function is available in native code or not. Performing a Compiler event consists simply of updating this data structure by changing the status of the appropriate functions to "compiled."

Performing an Interpreter event consists of updat-

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

ing 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.

5.3 Real Programs

To simulate the performance of real programs it is necessary to obtain the required setup and behavior files. The setup information for real programs was obtained using the `make` utility to generate file dependencies [4] and the `ctags` utility to determine which functions were contained in each source file. File sizes were obtained using `wc` after expansion by the preprocessor. We gathered data on compilation times by compiling the programs with the `gcc` compiler and using the `time` utility to report the elapsed time needed for compiling 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's excellent tool `EEL` [9] for instrumenting the executables to generate trace information. `EEL` was then used 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.

5.4 Profiling gcc

Throughout this paper we have used programs written in C, and the GNU `gcc` compiler, to generate our test data. However, current JIT systems are aimed at the Java runtime environment. An obvious question to ask is if data obtained using C programs is applicable to Java.

In order to answer this question it is necessary to compare the steps taken by the C compiler and a Java JIT compiler. Some operations, namely code generation and optimization, are likely to be similar in complexity. Parsing and semantic analysis, however, are likely to be much simpler for Java bytecodes than for C source code.

To examine this question in more detail, we attempted to profile the `gcc` compiler. Figure 13 shows the percentage of time spent in each category by the C-compilation portion of `gcc` on our study programs. This data was obtained by recompiling `gcc` so that it produced the monitor files used by the `prof` utility. The data was then further processed to eliminate the effect of the profiling itself from the data.

Due to the nature of the profiling process, the values given in Figure 13 reflect the percentage of CPU time, *not* real time, spent on the various steps of the compilation process. In addition, profiling was only performed on the C-compilation step, not on the preprocessor or the assembler. However, the values will serve as a general guide.

As can be seen from Figure 13, the parsing and semantic analysis steps account for between 14 and 25 percent of the compilation time. While this is a significant factor, it does not dominate the process. Even if we make the simplifying assumption that the parsing and semantic analysis required of Java bytecodes is negligible, this will not reduce the compilation time of equivalently sized programs by more than 25 percent since the other steps in the compilation process are not likely to change much.

How does this possible reduction in compilation time affect the studies presented in this paper? A reduced compilation time will result in a lower crossover point for any given compilation unit; however, the actual compilation time or crossover point of any given compilation unit has no effect on the general strategies presented for either smart JIT compilation or continuous compilation.

6 Previous work

The idea of just-in-time compilation is relatively new, though apparently nearly every Java vendor is eager to produce environments that include the just-in-time compilation approach [14, 3, 7].

For languages such as C, which have long been implemented primarily by compilation to native code (an exception is the si system [5]), interpreters are experiencing somewhat of a comeback: Proebsting and others are making interpreters much more efficient [13] by combining compiled with interpreted text; moreover, a strong market force is at play in the World-Wide Web, where Sun has created the HotJava system for distributing applications in a machine-independent fashion [11].

Java is likely not to be the only language available for safe execution on the Web. Recent work [1] from CMU and Colusa (now Microsoft) is likely to lead to efficient and safe implementations of inherently unsafe languages such as C.

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 [6]. Also, Kastens has considered how to generate interpreters automatically

Phase	ghostview	pico	render	combined
Scanning	5.29	3.33	4.02	4.28
Parsing	22.20	11.98	23.48	17.45
Semantic	2.25	1.84	1.90	2.03
Code Gen.	34.43	37.69	32.36	35.83
Optimize	14.90	22.07	13.92	18.23
Library	13.78	15.21	17.16	14.68
Other	7.15	7.88	7.15	7.50

Figure 13: Breakdown of CPU time spent by the C-compilation part of gcc on our study programs.

from compiler specifications [8]. The continuous compiler approach described in Section 3 differs from the above, in that an attempt is made to overlap compilation with interpretation, so that native-form code is available when needed.

7 Conclusions

We have shown that while JIT systems do perform well when compared with traditional compilation, there is room to do better. The continuous compiler performs as well or better than an optimal extant JIT approach, but uses two threads to do so. Combining ideas to obtain a smart JIT system can yield performance close to optimal JIT while using only a single thread.

The continuous compilation described in this paper focuses on the initial transition from interpreted to native code. However, continuous compilation need not end when all compilation units have been translated to native form. The continuous system could continue to perform optimization while the program is executing, replacing a compilation unit's native-code form with increasingly optimized versions.

It is important to note that JIT compilation and continuous compilation are not limited in effectiveness to Java, or C, or any language in particular. They are applicable to any field in which both performance and quick response to change are important. Visual languages, for example, or any interactive development environment could benefit from such an approach.

We anticipate that the ideas presented in this paper may prove useful in the area of run-time code generation [10, 2], where the profitability of spending time to generate more optimal code must be assessed.

Acknowledgements

We are grateful to Jim Larus for making his EEL system available to us. We thank Noah Treuhaft for his work in Summer '95, using EEL to instrument the C programs we studied. We thank Marc Snir, Susan Eggers, and Fred Schneider for conversations and correspondence on the problem we studied. We thank Mark Franklin for suggesting that we compare the two-processor continuous system with a twice-as-fast single-processor JIT system. We thank Bill Pugh for his insightful comments on our paper and for suggesting that we measure how gcc was spending its time.

References

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 127–136, 1996.

[2] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. pages 149–159, 1996.

[3] Symantec Corporation. Just-in-time compiler for windows 95/nt, 1996. http://cafe.symantec.com-/cafe/fs_jit.html.

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

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

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

[7] David Intersimone. Interview with régis crelier. Technical report, Borland International, 1996. http://www.borland.com/internet/java-/interviews/regis2.html.

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

[9] 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.

[10] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 137–148, 1996.

[11] Sun Microsystems. Sun offers a cupful of... hot-java. *Network world*, 12(21), May 1995.

[12] Michael P. Plezbert. Continuous compilation for software development and mobile computing. Master's thesis, Washington University, 1996. http://www.cs.wustl.edu/~plezbert-/contcom/thesis/thesis-main.html.

[13] 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.

[14] Evan Quinn. Sun's java workshop revolutionizes software development. Technical report, Sun Microsystems, 1996. http://www.eu.sun.com/sunsoft-/Developer-products/java/literature/idc.html.