**Lightweight Run-Time Code Generation**

Mark Leone  Peter Lee

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA
{mleone,peter}@cs.cmu.edu

**Abstract**

Run-time code generation is an alternative and complement to compile-time program analysis and optimization. Static analyses are inherently imprecise because most interesting aspects of run-time behavior are uncomputable. By deferring aspects of compilation to run time, more precise information about program behavior can be exploited, leading to greater opportunities for code improvement.

The cost of performing optimization at run time is of paramount importance, since it must be repaid by improved performance in order to obtain an overall speedup. This paper describes a lightweight approach to run-time code generation, called **deferred compilation**, in which compile-time specialization is employed to reduce the cost of optimizing and generating code at run time. Implementation strategies developed for a prototype compiler are discussed, and the results of preliminary experiments demonstrating significant overall speedup are presented.

**1 Introduction**

Many compiler optimizations depend on compile-time analysis to approximate properties of a program's run-time behavior. Static analyses are necessarily imprecise because most useful aspects of run-time behavior are uncomputable. Further compromises in precision must also be made in practice to reduce the complexity and inefficiency of analysis algorithms. Such imprecision makes it difficult for a compiler to optimize programs thoroughly.

An alternative approach is to defer some analysis and optimization (and therefore also code generation) to run time. While this does not avoid the fundamental problems of uncomputability and inefficiency, it does make possible the use of run-time information in improving code quality.

In this paper we report on our experience with a new approach to generating optimized code at run time. The salient characteristics of our approach, which we term **deferred compilation**, are as follows:

- It is lightweight. Compile-time specialization eliminates the need to process any intermediate representation of a program at run time. Each part of a compiled program that performs run-time code generation is "hard wired" to optimize and generate code for a small portion of the input program.

- It is largely automatic. Manual construction of code templates or run-time code generators is not required. Syntactic cues and programmer hints are used to determine which parts of a program should be subjected to run-time compilation.

- It is general. Many standard optimizations, such as strength reduction and function inlining, can be efficiently employed at run time.

We have implemented a prototype compiler, which we call **FABiUS**, to evaluate this approach. In preliminary experiments, we have found that the overhead of deferred compilation is often quite small when compared to the performance gain. Furthermore, we have encountered unique design tradeoffs in considering which aspects of optimization and code generation should be performed statically and which should be deferred to run time. We see some encouraging signs that deferred compilation can be practical, and we find that there is much further work to be done.

**1.1 Motivation**

Run-time code generation is beneficial for programs that exhibit multiple stages of computation because the code for late stages can be optimized based on values computed in early stages. Multiple stages of computation occur naturally in both functional and imperative programs. For example, when a strict curried function \( f \) of type \( \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \) is applied to an argument \( x \), a closure representing a value of type \( \tau_2 \rightarrow \tau_3 \) will typically be constructed before computations involving additional arguments proceed. It may be profitable to generate optimized code for \( f(x) \) if it will be applied many times. Run-time code generation can therefore be viewed as an alternative to the conventional implementation of closures; this idea has also been described by Appel [App87] and Feeley and Lapalme [FL92].
Stages of computation also arise from conventional iteration constructs such as loop nests. Values computed in an outer loop are usually fixed for the duration of inner loops, and substantial benefits might be obtained by optimizing inner loops for each iteration of an outer loop. More deeply nested loops lead to more stages of computation.

For example matrix multiplication is commonly implemented as a triply nested loop: the two outer loops select a vector from each matrix and the innermost loop computes their dot product. At run time it may be profitable to specialize the dot-product loop, since the vector selected by the outermost loop will be used to compute many dot products. The dot-product loop can be completely unrolled, eliminating a large number of bounds checks and branches. Its arithmetic operations can also be optimized based on the contents of the fixed vector, which can significantly improve performance on sparse data. Such optimizations usually cannot be performed at compile time, however, because the sizes and contents of the matrices are generally not statically apparent.

Run-time code generation may also reduce the cost of abstraction, allowing well-structured programs to be written without incurring an undue performance penalty. Functional programming languages encourage the use of small functions that are combined using higher-order constructs such as composition, map, and fold, but programs written in this manner are difficult to optimize. Intraprocedural optimizations are relatively ineffective because of the small size of basic blocks, and interprocedural optimizations are difficult to perform at compile time because control flow is uncomputable.

1.2 Compile-Time Alternatives

Program staging can also be exploited by some compile-time program transformations. To see why run-time code generation is desirable, consider the following simple program, which repeatedly calls an exponentiation function with a fixed exponent that is statically unknown:

\[
\begin{align*}
\text{fun power exp} &= \text{map (power exp) bases} \\
\text{and power exp base} &= \\
\text{if exp} &= 0 \text{ then } 1 \\
\text{else base} &= \text{power (exp - 1) base}
\end{align*}
\]

1.2.1 Staging Transformations

Various techniques such as staging transformation \cite{JS86, Han91}, program bifurcation \cite{Mog89, DBV91}, fold/unfold transformations \cite{BD77}, or loop-invariant removal \cite{ASU86} might be employed at compile time to "hoist" computations that do not depend on base out of its scope. Hoisting the conditional test, subtraction, and recursive call yields the following implementation of power:\footnote{\text{fn} \ s \Rightarrow \ e \text{ denotes the strict function } \lambda s.e.}

\[
\text{fun power exp} = \\
\text{if exp} &= 0 \text{ then } (\text{fn base} \Rightarrow \text{1}) \\
\text{else let pow = power (exp - 1) in} \\
\text{fn base} &= \text{base} * \text{pow base}
\]

This transformation reduces the dynamic frequency of the hoisted operations, which otherwise would be repeated numerous times. However, a significant amount of overhead has been introduced: calling this implementation of power will typically result in the creation of a large number of closures, each containing a code pointer and a pointer to another closure.

Run-time code generation can reduce the dynamic frequency of loop-invariant operations with less overhead. In addition, it can optimize non-loop-invariant computations using information that is not available at compile time.

1.2.2 Static Specialization

Alternatively, specialization \cite{JGS93}, driving \cite{Tur86}, or procedure cloning \cite{CKH93} might be employed at compile time to transform power into the following function:\footnote{\text{nth} \ i \ \text{[a\_0, \ldots, a\_n]} \text{ yields } a\_i, \text{ the } i^{\text{th}} \text{ element of a list. An exception called } N\text{th} \text{ is raised if } i > n, \text{ in which case power returns a closure containing the value of exp and the code of an unspecialized exponentiation function.}}

\[
\begin{align*}
\text{fun powgen exp} &= \\
\text{nth exp} &= [\text{fn base} \Rightarrow \text{1}, \\
\text{fn base} &= \text{base}, \\
\text{fn base} &= \text{base} * \text{base}, \\
\text{fn base} &= \text{base} * \text{base} * \text{base}, \\
& \vdots \\
\text{handle nth} &= \text{power exp}
\end{align*}
\]

Given an exponent value \text{powgen} simply chooses from a table of \text{power} functions, each specialized to a value of exp from 0 to \(k - 1\). The specialized functions are highly optimized and can be compiled into high-quality machine code, so one might expect this approach to be useful in the same situations as run-time code generation.

However, there are two practical problems in performing such a transformation automatically. First there is the matter of choosing the set of values on which to specialize. In general it is not possible to predict the range of values that might result from an early computation, and such values might be arbitrary data structures. A second problem is that, due to space constraints, a relatively small limit must be placed on the number of specialized functions created at compile time (represented by the constant \(k\) in the above example). Key benefits of run-time code generation are that specialization occurs "on demand" and that code space can be reused.

1.3 Run-Time Code Generation

Run-time code generation has a long history; a useful summary can be found in \cite{KEH91}. However, it has not been widely adopted, primarily because it is difficult to automate and because the cost of optimization is often recovered only for large input sizes.

The cost of run-time code generation is often reduced using templates. A template is a manually constructed, machine-dependent representation of code containing "holes" in place of some values. Run-time code generation simply requires copying a template and instantiating its holes with run-time values, which yields run-time "constant" propagation. Other simple optimizations can be achieved at low
cost; for example, loop unrolling can be implemented by concatenating templates.

Templates are employed in the Synthesis kernel [Mas92, MP90] to reduce the overhead of kernel calls and context switches. Other applications that benefit significantly from run-time template compilation include decompression and cache simulation [KEH93] and the bitblt graphics primitive [PLR86].

Although run-time template compilation is fast, the range of optimizations that may be applied at run time is limited. For example, instruction scheduling across template boundaries is difficult to achieve. Using a more general intermediate representation permits a wider range of optimizations and results in higher-quality code, but at higher cost.

Keppel, Eggers, and Henry [KEH91, KEH93] have explored the tradeoff between run-time code generation costs and code quality by implementing a template compiler and a more general intermediate-representation compiler for several applications. Engler and Proebsting [EP93] have investigated using the ldcc compiler’s intermediate representation for run-time code generation. Intermediate code constructed using ad hoc methods can be compiled by the lcc back end at run time and then directly executed.

Run-time compilation is employed in SELF [HU94, DC94, CU91], a compiler for a classless object-oriented language, using a general-purpose intermediate representation. Run-time optimizations are obtained automatically by simply deferring the bulk of compilation to run time. Similar approaches have been implemented for Smalltalk [DS84] and concurrent object-oriented languages [CKP93]. In SELF the primary run-time optimizations are inlining and a form of specialization in which methods are customized to reduce the cost of dynamic type dispatch. The cost of run-time compilation is reduced by delaying the compilation of infrequently executed methods and applying aggressive optimizations only to frequently executed methods using dynamic recompilation. Nevertheless, compilation is time consuming: the SELF/91 run-time compiler is about as fast as an optimizing C compiler [CU91].

2 Deferred Compilation

Deferred compilation employs compile-time specialization to reduce the cost of run-time code generation. No intermediate representation of a program is processed at run time; instead, portions of a program are compiled into code that is “hard-wired” to perform optimizations and generate native code at run time. Our goal is to make run-time code generation lightweight and largely automatic without greatly limiting the range of optimizations that may be applied at run time.

There are close connections between deferred compilation and partial evaluation [JGS93]. A partial evaluator, called mix for historical reasons, could be invoked at run time to specialize (the text of) a function $f$ to a particular argument value:

$$\text{mix}([[f]], x) = f_a$$

where $[[f]]$ is a closed term representing $f$, $x$ is a run-time value, and $f_a$ is the so-called residual program, representing the specialization of $f$ to $x$. A key aspect of deferred compilation is that the input $x$ is not known until run time, so some amount of specialization must be carried out at run time. The text of $f$ is known at compile time, however, so the cost of run-time specialization can be reduced by a compile-time specialization of mix to $f$:

$$\text{mix}([[\text{mix}]], [f]) = \text{mix}_f$$

If mix exhibits good “binding-time separation,” mix$_f$ will specialize $f_a$ without the overhead of processing $[f]$ at run time. Deferred compilation employs a similar form of compile-time specialization to reduce the cost of optimizing and generating code at run time.

In the remainder of this section we illustrate the basic principles of deferred compilation by describing the design of a prototype implementation. The connections between deferred compilation and partial evaluation are discussed in more detail in Section 2.4.

2.1 Implementation

We have implemented a prototype compiler called FABIUS$^4$ to investigate and evaluate the notion of deferred compilation. A key issue is how to apportion the costs of optimization and code generation between compile time and run time. For the moment, the primary goal of FABIUS is to reduce the run-time cost of code generation to a minimum, at the cost of some degradation in the quality of the generated code and an increase in the size of both the generating and the generated code. This provides a baseline for the evaluation of more aggressive run-time optimizations. Future extensions to FABIUS are discussed in Section 3.2 and Section 5.

The FABIUS source language is, for the time being, a rudimentary, strict, first-order functional language. Integers and pointers to heap-allocated structures are the only run-time values. FABIUS generates native code for the MIPS R2000. The three major phases of compilation are as follows:

- Staging analysis identifies computation stages at which it may be profitable to perform run-time code generation. In a process similar to binding-time analysis [ISS89, Con92], subexpressions of the program are annotated to indicate whether they belong to early or late stages of computation.
- Register allocation assigns registers to program variables and intermediate values. The usual notion of lifetime ranges must be modified, since textually adjacent computations may belong to different program stages, and may therefore use overlapping register sets.
- Code generation compiles “early” computations in the usual way, but “late” computations are compiled into machine code that generates optimized instruction sequences at run time.

A simple example will be used to illustrate these steps in more detail. As mentioned in Section 1.1, matrix multiplication is well suited to run-time code generation. It is usually implemented as a triply nested loop, where the two outer loops select a vector from each matrix and the innermost loop computes their dot product. Consider the following (tail-recursive) implementation of a dot-product loop:

\[ f = 3 \]

3 Chambers and Ungar [CU91] originally coined the term “deferred compilation” to describe this strategy.

4 Quintus Fabius Maximus was a Roman general best known for his defeat of Hannibal in the Second Punic War. His primary strategy was to delay confrontation; repeated small attacks eventually led to victory without a single decisive conflict.
fun dotprod (v1, v2, sum) =  
if v1 = nil then sum  
else dotprod (tl v1, tl v2,  
sum + hd v1 * hd v2)

To simplify the presentation we assume vectors are implemented as linked lists of integers. As a point of reference, consider the machine code that might be generated for the dotprod function by a conventional compiler:

dotprod:  
beq x1, r0, L1 ; if v1 = nil goto L1  
ld r4, (r1) ; x1 = hd v1  
ld r5, (r2) ; x2 = hd v2  
mul r4, r4, r5 ; prod = x1 * x2  
add r3, r3, r4 ; sum = sum + prod  
ld r1, 4(r1) ; v1 = tl v1  
ld r2, 4(r2) ; v2 = tl v2  
jmp dotprod ; goto dotprod  
L1:  
move x1, r3 ; result = sum  
ret ; return

The next section describes how staging analysis identifies where compilation can be deferred, and Section 2.3 describes how FABiUS creates a specialized code generator for dotprod.

2.2 Staging Analysis

In the implementation of matrix multiply described above, the vector selected by the outermost loop, v1, is used to compute numerous dot products. It may therefore be profitable to create an optimized dot-product function (parameterized by v2 and sum) for each iteration of the outer loop. In such situations, we say that the first argument of dotprod is available at an early stage and the remaining arguments are available at a late stage.

Deferred compilation employs a staging analysis to identify such computation stages and track data dependencies. The subexpressions of a program are annotated to indicate whether they depend only on the results of early computations or whether they might rely on late-stage results. For example, doctprod might be annotated as follows, where overlines indicate early computations and underlines indicate late computations:

fun dotprod (v\text{\texttt{T}}, v2, sum) =  
if v1 = nil then sum  
else dotprod (tl v1, tl v2,  
sum + hd v1 \texttt{T} * hd v2)

The test \(v1=nil\) can be computed at an early stage because \(v1\) is available early, and therefore the conditional branch can also be performed early, and so on. The staging analysis also labels the recursive application as an early computation, indicating that the function should be inlined at run time (see Section 3.3).

In the case of just two stages, this labeling of early and late computations is very similar to a binding-time analysis and annotation [JSS89, Con93]. There is a subtle difference between the two, however. Binding-time analysis is guided by an externally imposed division of the program inputs that specifies which are supplied at compile time and which will only be available at run time. But in the setting of deferred compilation, all of the program inputs are supplied at the same time, upon execution of the compiled program. The distinction between "early" and "late" stages is introduced by the compiler, quite artificially, in the hope of obtaining faster code.

Detecting program stages is a difficult problem. Syntactic features of programming languages often provide clear indications of stages that can be subjected to deferred compilation. Applications of curried functions are an obvious candidate for deferred compilation, as are nested loop constructs. Consel, Pu, and Walpole [CPW93] recently proposed the use of a multi-level programming language to allow lower programmers to express invariants that will become established during the execution of a large system. For example, an operating system might be structured in a way that permits specialization to be applied at compile time, link time, boot time, and run time. Such a multi-level language would facilitate staging analysis.

In addition to detecting program stages, staging analysis must also determine which stages will benefit from run-time compilation. Partial evaluators commonly adopt the aggressive strategy of performing specialization whenever possible, even when doing so does not result in significant code improvements. Deferred compilation requires a more conservative approach because the cost of optimization must be repaid by improved performance. It can be quite difficult to determine when this will be the case, especially since optimizations applied at run time may be machine specific. Online strategies that employ run-time information to guide optimization may be necessary. For example, online strategies are employed in the Cecil and SELF compilers to determine where method specialization [DCG94] and inlining [DC94] should be applied at run time.

Currently FABiUS relies on programmer hints to determine where run-time code generation will be profitable. A notation similar to currying can be used in function definitions to specify which formal parameters are "early" and which are "late." Calls to such functions require a syntax similar to curried application to introduce a "division" of the actual arguments.

In the parlance of partial evaluation, this technique is monovariant because all call sites must supply a "compatible" division of the actual arguments. But a degree of polyvariance is possible because the "signature" of a function is fixed by its definition and is not affected by the stages assigned to its actual arguments. For example, in a "curried" application of the form

\(f(e_1, \ldots, e_m)(e'_1, \ldots, e'_n)\)

a compatible division of the actual arguments is constructed as follows:

- If any \(e_i\) is late, all \(e_i\) and \(e'_i\) are "lifted" to a late stage.
- If any \(e'_i\) is late, all \(e'_i\) are "lifted" to a late stage.

As a result, the staging "analysis" implemented in FABiUS is a simple, one-pass annotation algorithm.

Currently only two-stage divisions are supported. Nevertheless, programs exhibiting more than two stages of execution can benefit from run-time code generation at each stage because "late" values may be passed as "early" arguments. In such circumstances run-time-generated code will contain calls to statically compiled code generators.

---

\(\text{hd}\) returns the head of a list and \(\text{tl}\) the tail.
2.3 Code Generation

After staging analysis, FABiUS performs register allocation, the discussion of which we postpone to Section 3.2, and then code generation. Early computations are compiled in the usual way, but late computations are compiled into code that emits optimized instruction sequences at run time. From the annotated dotprod function shown above, FABiUS generates the following code:\textsuperscript{6}

\begin{verbatim}
dotgen:
    beq r1, r0, L1 ; if v1 = nil goto L1
    ld r2, (r1) ; x1 = hd v1
    emit ld r3, (r1) ; emit x2 = hd v2
    emit mul r3, [r2], r3 ; emit prod = x1 * x2
    emit add r2, r2, r3 ; emit sum = sum + prod
    ld r1, 4(r1) ; v1 = t1 v1
    emit ld r1, 4(r1) ; emit v2 = t1 v2
    jmp dotgen ; goto dotgen

L1:
    emit move r1, r2 ; emit result = sum
    ret ; return
\end{verbatim}

There are only two differences between dotgen and the machine code for dotprod that was presented earlier. First, some instructions are emitted (perhaps many times) instead of being executed. Second, some early and late values have been assigned to the same registers, since operations involving these values belong to non-overlapping program stages. This optimization is discussed further in Section 3.2.

dotgen is a specialized code generator that does not manipulate any intermediate representation of the source program at run time. Readers familiar with partial evaluation may notice that dotgen is a generating extension for dotprod. Specialized code generators can also be viewed as executable data structures [Mas92] in which interpretational overhead is eliminated by merging code and data.

dotgen effectively performs "constant" propagation, conditional folding, and inlining at run time. For example when called with the vector \( [x_1, \ldots, x_n] \), the following code is generated at run time:

\begin{verbatim}
ld r3, (r1) ; x2 = hd v2
mul r3, x1, r3 ; prod = x1 * x2
add r2, r2, r3 ; sum = sum + prod
ld r1, 4(r1) ; v2 = t1 v2
...
ld r3, (r1) ; x2 = hd v2
mul r3, x1, r3 ; prod = x1 * x2
add r2, r2, r3 ; sum = sum + prod
ld r1, 4(r1) ; v2 = t1 v2
move r1, r2 ; result = sum
\end{verbatim}

Making deferred compilation practical for a wide variety of programs is more of a challenge than this simple example might imply. Here we see that run-time inlining can be highly profitable, but clearly there are limits; if pursued too aggressively, the run-time overhead may exceed the performance gain of the dynamically generated code. Section 3 discusses such issues in more detail and describes the strategies employed to address them. We also examine how a wider range of optimizations and code generation techniques can be adapted to deferred compilation.

2.4 Deferred Compilation via Partial Evaluation

The FABiUS code generator is structured much like a conventional code generator. It is interesting to note, however, that it could be automatically derived in a more principled way using partial evaluation.

To see how this might be accomplished, first consider how a conventional partial evaluator, mix, might be used to specialize a function \( f \) at run time:

\[ \text{mix}(f[x], z) = [f_z] \]

Here \([f]\) is the source code for \( f \), and \( z \) is a run-time value. Since conventional partial evaluators are designed for source-to-source program transformation, the specialized code that results, \([f_z]\), will be expressed in a high-level language. Hence it will require (run-time) compilation before it can be executed:

\[ (\text{comp} \circ \text{mix})(f[x], z) = [f_z] \]

The single-brackets indicate a result in the form of native code. However, note that \([f]\) is known at compile time, so we can specialize \( \text{comp} \circ \text{mix} \) (or \text{comix} for short) to reduce the cost of run-time specialization and compilation:

\[ \text{mix}([\text{comix}], [f[x]]) = [\text{comix}] \]

The result is (the text of) a program that specializes and compiles \( f \) to native code at run time. If \text{comix} exhibits good binding-time separation, no intermediate representation of \([f]\) will be processed at run time.

Unfortunately, a naive composition of \text{comp} and \text{mix} does not have good binding-time separation, so run-time compilation will not be specialized. It is well known, however, that partial evaluators need not express residual programs as source code. One could directly implement \text{comix} as a specializer that generates native machine code. A compiler similar to FABiUS could then be obtained by self-application:

\[ \text{comix}([\text{comix}], [\text{comix}]) \approx \text{FABiUS} \]

To the best of our knowledge, no existing partial evaluator directly generates native machine code. One system that comes close to this goal is AMIX, a self-applicable partial evaluator for a first-order functional language whose target is an abstract stack machine [Hol88]. AMIX's abstract machine code is a relatively high-level language, however, and the cost of compiling it to native code at run time would be substantial. The interpretational overhead present in this compilation cannot be statically eliminated.

FABiUS can be viewed as a manually derived implementation of \text{comix}([\text{comix}], [\text{comix}]). This bears strong similarity to the notion of hand-writing \textit{cogen} [BHOS76, EH80].

\textsuperscript{6} We use the pseudo-instruction "emit" to simplify the presentation. It expands into a sequence of instructions that allocates space in a dynamic code segment, builds the representation of an instruction from its opcode and arguments, and finally writes the instruction to the allocated space. The operands of the emitted instruction are usually fixed, but the value of an immediate operand may be determined by the contents of a register. For example, "emit mul r3, r2, r3" can emit a register-immediate multiply instruction if the value in \( r3 \) is small enough to permit such an encoding.

For clarity, the order in which arguments are computed has been changed, which eliminates some register shuffling, and code the emits procedure linkage code has been omitted.
How then does deferred compilation differ from partial evaluation? Perhaps the most fundamental difference, discussed in Section 2.2, is that deferred compilation is not driven by an externally imposed division of program inputs, but rather by the staging inherent a program. In addition, deferred compilation incorporates low-level optimizations that are not considered by conventional specialized (such as strength reduction and register allocation) and requires different heuristics to determine where specialization and unfolding should be applied. Section 3 discusses these issues in more detail.

3 Run-Time Optimizations

The dot-product example presented in Section 2.3 demonstrated that simply deferring code generation to run time allows us to achieve run-time "constant" folding, loop unrolling, and "dead code" elimination. Other conventional optimization and code generation techniques can be adapted to deferred compilation, but the lack of any run-time intermediate representation poses a challenge. Optimizations must be stag ed in a way that separates the manipulation of the source program from the use of run-time information. In partial evaluation terminology, optimizations must exhibit good "binding-time separation." Fortunately this seems to be the case for many conventional optimizations.

3.1 Local Optimizations

Local optimizations, such as strength reduction and instruction selection, are easy to adapt to deferred compilation. For example, when emitting \texttt{x*y}, where \texttt{x} is the result of an early computation, a code generator can first check whether \texttt{x} is zero and if so emit a cheaper instruction:

\begin{verbatim}
  beq r1, r0, L1 ; if x = 0 goto L1
  emit mul r2, [r1], r1 ; emit z = x * y
  jmp L2 ; goto L2
L1: emit move r2, 0 ; emit z = 0
L2:
\end{verbatim}

\texttt{Fabius} incorporates a number of local optimizations of this form. The increased cost of such run-time optimizations must be weighed against their benefit. In this case, the cost is only a few cycles per multiplication and the benefits can be significant, as we demonstrate in Section 4 where we consider a computation involving sparse data. The choice of run-time optimizations need not be made globally: specialized code generators can be individually tailored to perform a variety of optimizations.

3.2 Register Allocation

Deferred compilation reduces register pressure but complicates register allocation. Fewer registers are required because program stages are not interleaved and can therefore use overlapping register sets. For example in \texttt{dotprod}, computations involving the two vectors are textually adjacent, but because they belong to different stages the same register can be assigned to both vectors.

Existing register allocation algorithms based on graph coloring [Cha92, CH84] can be adapted to deferred compilation by simply modifying the construction of the interference graph. An interference graph contains nodes representing the lifetime ranges of variables and edges indicating where these ranges intersect. Any \(K\)-coloring of this graph is therefore a valid assignment of the variables to \(K\) registers. Deferred compilation simply requires a revised notion of variable lifetime: during construction of the interference graph, edges should only be added between overlapping lifetime ranges of variables from the same program stage.

\texttt{Fabius} uses a similar technique to perform all register allocation at compile time, which significantly reduces the cost of run-time code generation. This approach has drawbacks, however. Inlining and loop unrolling may occur at run time, so an exact interference graph cannot be constructed at compile time. Also, fixing the register assignments of functions at compile time makes run-time inlining less effective. For example, code must be generated to shuffle registers if the formal and actual parameters of a function are assigned to different registers, and so forth. Eliminating this kind of overhead is one of the primary motivations for inlining, so it seems desirable to perform run-time register allocation in some cases.

Constructing an exact interference graph at run time is likely to be prohibitively expensive, so we are investigating an alternative technique in which register allocation is performed at compile time but register assignment for late-stage computations is deferred to run time. Compile-time register allocation can conservatively determine where register spilling is necessary, based on a static approximation of the interference graph as described above, and run-time code generators can be parameterized by register mappings. For example, \texttt{dotgen} can perform run-time register assignment as follows:

\begin{verbatim}
  dotgen:
  beq r1, r0, L1 ; if v1 = nil goto L1
  ld r2, (r1) ; r1 = hd v1
  emit ld R_4, (R_3) ; emit x2 = hd v2
  emit mul R_4, [R_3], R_3 ; emit prod = x1 * x2
  emit add R_3, R_3, R_4 ; emit sum = sum + prod
  ld r1, 4(r1) ; v1 = t1 v1
  emit ld R_2, 4(R_2) ; emit v2 = t1 v2
  jmp dotgen ; goto dotgen
L1:
  emit move R,v5, R_3 ; emit result = sum
  ret ; return
\end{verbatim}

This function takes five arguments: the value of \texttt{v1} (in \texttt{r1}), the numbers of the registers assigned to \texttt{v2} and \texttt{sum} (in \texttt{r2} and \texttt{r3}), and the numbers of an available temporary register and the destination register (in \texttt{r4} and \texttt{r5}). Emit pseudo-instruction used here determines the operands of the emitted instruction from the contents of the specified registers. This takes a few more cycles than emitting instructions with fixed operands, but the generated code will be more efficient in contexts that would otherwise require the register shuffling described above.

3.3 Inlining and Loop Unrolling

Inlining and loop unrolling are valuable optimizations in conventional compilers because they yield increased opportunities for optimization, eliminate the overhead of function calls, and improve the amortization of computations such as range checks [DH88]. The extent to which these optimizations may be performed at compile time is rather limited, however. Loop bounds are usually known, and a loop can only be unrolled a fixed number of times. Special-case code containing numerous branches is required when unrolling a
loop because the actual number of iterations might not be a multiple of the unrolling depth. Inlining is often impossible when compiling languages with higher-order functions because calls to unknown functions are common. Run-time inlining and loop unrolling can solve these problems.

FABIUS decides at compile time where run-time inlining should occur, based on the results of the staging analysis. Loops are expressed as tail-recursive functions, so inlining yields loop unrolling. An aggressive heuristic currently guides inlining: a call to a function with both "early" and "late" formal parameters is marked for run-time inlining if it does not appear in a branch of a conditional controlled by a late-stage value [BD91]. Each function that might be inlined at run time is compiled into a specialized code generator that does not emit procedure linkage code, but instead emits optimized code directly into the code being generated for the calling context.

Although the inlining strategy implemented by FABIUS preserves the termination behavior of programs [BD91], it remains to be seen whether the increased time and space requirements of such aggressive run-time inlining are manageable in large applications.

3.4 Specialization

In some contexts it is impractical to inline a function yet still desirable to specialize it based upon the results of early computations. For example if a function is called at several different program points with the same value from an early computation, it may be preferable to generate a single optimized version of the function rather than inlining its body at each call site. Specialization also permits run-time optimized code to be reused rather than regenerated, which saves both space and time.

Determining where specialization will be beneficial is a difficult problem. FABIUS currently implements the aggressive heuristic employed in [BD91]: all functions with both "early" and "late" arguments that are not inlined are specialized. Such functions are compiled into specialized code generators, parameterized by the values of the early arguments, which generate optimized functions at run time. These code generators are memoized, so that previously optimized code is reused whenever possible. Run-time memoization on structured data can be quite expensive [Mal93], so FABIUS uses pointer equality.

Although this form of specialization yields significant speedups for some examples, it does not always terminate [BD91]. Preliminary experiments also indicate that the strategy is too aggressive in practice, since some functions do not benefit significantly from specialization.

4 Results

Preliminary experiments with FABIUS are encouraging. As an example we return to the matrix multiplication algorithm that was introduced in Section 2. We measured the number of instructions executed and the overall execution time of matrix multiply for both dense and sparse inputs. Execution times were measured on an unloaded DECstation 5000/200. The inputs were square matrices of dimension $n$ containing pseudo-random untagged 32-bit integers. Since FABIUS does not yet support vectors or arrays, we implemented matrices as lists of lists. Sparse input was 90% zero, and non-zero values were randomly located. The same run-time optimizations were applied for both dense and sparse inputs. For the purposes of comparison we also measured the performance of statically optimized code, which was obtained by simply disabling the FABIUS staging analysis. The quality of this code was good, although compile-time inlining was not performed.

Since the FABIUS prototype does not yet have a garbage collector, our measurements do not reflect the cost of reclaiming data and code space, although they do account for the cost of allocation. Also, since code space is not yet reused at run time, we have not yet evaluated the cost of instruction-cache flushing. This issue is discussed further in Section 4.3.

4.1 Instruction Counts

Figure 1 compares the number of instructions executed when multiplying dense and sparse matrices of varying size. The top curve gives the performance of the statically optimized
code, and the middle two curves demonstrate the improvement obtained by deferred compilation, after taking into account the cost of generating code at run time. The lowest curve represents the cost of run-time code generation for dense input. Sparse input resulted in faster run-time code generation, since fewer instructions were emitted.

As the figure demonstrates, deferred compilation can significantly improve performance even for small problem sizes. In this case the cost of optimizing and generating code at run time was recovered for dense matrices larger than $20 \times 20$ and sparse matrices larger than $2 \times 2$. For larger input sizes the improvement obtained for dense data asymptotically approaches 20%, and the improvement on sparse data increases linearly. At $n = 200$ deferred compilation reduced the total number of instructions executed by 18% for dense data and 72% for sparse data. The cost of run-time optimization and code generation was very low, accounting for 3% of the total number of instructions executed when $n = 200$. For dense input, the average cost was 4.8 instructions for each instruction generated at run time; the average was 6.2 for sparse input.

### 4.2 Execution Times

As predicted by the instruction counts, the time spent optimizing and generating code at run time was brief, totalling 170 milliseconds for dense input and 120 milliseconds for sparse input when $n = 200$. However, overall execution times were greater than we expected. As shown in Figure 2, deferred compilation did not yield an overall speedup for dense matrices, and yielded a speedup of only 50% for sparse matrices.

This curious behavior is caused by an instruction pipeline stall. On the MIPS an integer-multiply instruction requires several cycles to complete, and the instruction pipeline will stall if an attempt is made to access the result before it is ready. The statically optimized code is large enough that several unrelated instructions can be scheduled after the multiplication in the dot-product function, so a pipeline stall does not occur. Run-time code generation happens to eliminate precisely these instructions.

We verified this explanation by measuring the execution time of a variant of the benchmark code in which the result of the multiplication was discarded. Overall execution time improved significantly, and was closely predicted by instruction counts, indicating that no other pipeline stalls were introduced by deferred compilation.

Better instruction scheduling can reduce or eliminate such pipeline stalls. For example, instructions from iteration $i+1$ of the dot product loop can be scheduled after the multiplication in the $i^{th}$ iteration. However, since the dot-product loop is not unrolled at compile time, it may be necessary to perform this instruction scheduling at run time. We are currently investigating the feasibility of such optimizations.

### 4.3 Caveats

Although encouraging, the results of this experiment are not conclusive. A more realistic implementation of matrix multiply would likely include special-purpose code for sparse inputs, and aggressive compile-time loop optimizations might be able to improve the performance of our benchmark code without resorting to run-time code generation. Such performance improvements are not incompatible with deferred compilation, however.

Additional concerns that must be addressed in practice include space usage and instruction caching. Compile-time specialization of run-time code generators essentially trades space for time. For example, enabling run-time code generation more than doubled the static size of the matrix multiply code, from 70 words to 150 words. Aggressive run-time inlining and loop unrolling can also dramatically increase space requirements. For example when $n = 200$ the code generated at run time for a single dot product function occupied approximately 1800 words; if code space is not reclaimed the total space required exceeds one megabyte.

Run-time code generation can also interact poorly with instruction caching. Most modern architectures prefetch instructions into an instruction cache, and many do not automatically invalidate cache entries when memory writes occur. Portions of the instruction cache may therefore need
to be flushed whenever new code is generated at run time [KEH91]. Fortunately the regularity of code-space allocation and initialization simplifies amortizing the cost of such operations. FABUS aligns each newly allocated code object to a boundary that the MIPS instruction prefetcher is guaranteed not to have crossed while executing previously generated code, thus avoiding the invalidation of cached instructions. Cache flushing is therefore required only when garbage collection occurs.1

Although we have not yet implemented code-space reallocation, we believe that the cost of cache flushing will not be significant. For example Keppel reports that flushing the instruction cache on a DECstation 5000/200 requires a kernel trap plus approximately 0.8 nanoseconds per byte flushed [Kep91]. This would add approximately ten milliseconds to the total cost of run-time code generation when multiplying 200 x 200 matrices.

5 Conclusions and Future Research

Deferred compilation is an alternative and complement to compile-time analysis and optimization in which aspects of optimization and code generation are deferred to run time. Fast run-time optimization and code generation is achieved by eliminating, through compile-time specialization, the overhead of processing intermediate representations of source programs at run time. Preliminary experiments with a prototype compiler are promising, but we find that further experimentation is required for a full assessment.

Currently, we are extending the FABUS prototype to compile programs with higher-order functions and mutable data structures. We are also exploring the use of a wider range of optimization and code generation techniques at run time. Of particular interest is the usefulness of performing register assignment at run time. Also, we plan a detailed study of the effectiveness of different strategies for controlling run-time inlining, specialization, and memoization.

Finally, the matter of staging analysis is, at this point, poorly understood. We currently rely on programmer hints to determine where run-time code generation will be profitable, but clearly a more automatic approach is desirable. The development of both the theoretical foundations and a practical implementation of automatic staging analysis will be the subject of future research.

Acknowledgements

We are grateful to Chris Stone, who provided valuable assistance in the implementation of FABUS, and also to Ali-Reza Adl-Tabatabai, Chris Colby, Charles Consel, Olivier Danvy, David Keppel, Mark Lillibridge, Greg Morrisett, Chris Okasaki, Amir Sabry, Chris Stone, and David Tadrito for their time and effort in productive discussions.

1A vexing problem arises if pointers are propagated like other values during optimization, since they may become embedded in run-time-generated code. The garbage collector must then update code and flush the instruction cache whenever & moves data. Such embedded pointers may be difficult to locate and update; for example on the MIPS a constant 32-bit pointer might be embedded into two instructions that contain 16-bit immediate values. Instruction reordering during run-time code generation can make the locations of these instructions unpredictable.

References


