Deferred Compilation: The Automation of Run-Time Code Generation

Mark Leone Peter Lee

December 1993

CMU-CS-93-225

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes *deferred compilation*, an alternative and complement to compile-time program analysis and optimization. By deferring aspects of compilation to run time, exact information about programs can be exploited, leading to greater opportunities for code improvement. This is in contrast to the use of static analyses, which are inherently conservative.

Deferred compilation automates the translation of ordinary programs into native machine code that performs fast optimization and native-code generation at run time. Automation is obtained through the use of a compile-time *staging analysis*, which determines the portions of a program that may be safely and profitably compiled at run time. Fast run-time optimization is obtained by trading space for time: compile-time *specialization* yields numerous run-time code generators, each customized to optimize a small portion of the source program based on run-time information. Implementation strategies developed for a prototype compiler are discussed, and the results of preliminary experiments demonstrating significant overall speedup are presented.

The authors' electronic mail addresses are `Mark.Leone@cs.cmu.edu` and `Peter.Lee@cs.cmu.edu`.

This research was partially supported by the National Science Foundation under grant #CCR-9057567. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

Keywords: Run-time code generation, deferred compilation, partial evaluation, specialization, binding-time analysis, staging transformation.

1 Introduction

Many compiler optimization techniques depend on static analysis to determine invariants about a program's run-time behavior. As a result, a great deal of research has been invested in the development of various approaches to static program analysis, particularly in the areas of dataflow analysis, abstract interpretation, and nonstandard forms of type inference. Despite good progress, such analyses tend to be excessively conservative in practice, thus making it difficult for a compiler to achieve thorough optimization of programs. This is, of course, a fundamental problem since most aspects of program run-time behavior are undecidable. Also, as a practical matter, further compromises in precision must be made in order to cope with the complexity and inefficiency of many analysis algorithms.

An alternative approach is to defer at least some of the analysis and optimization (and therefore also code generation) to run time. While this does not avoid the fundamental problems of undecidability and inefficiency, it does make possible the use of run-time values in improving code quality. This is an old idea that has been applied in many different ways. For example, for regular expression search, Thompson describes what essentially amounts to a compiler for regular expressions. A program can invoke this compiler at run time to obtain machine code optimized for a specific regular expression [Tho68]. A similar approach has also been applied to bitblt [PLR85] and to the implementation of operating system services [Mas92, MP89]. For general programming, Keppel, Eggers, and Henry have studied several manual methods for obtaining such "application-specific" compilers, and they show that good results are possible for realistic C programs [KEH93].

There are other ways to improve program performance using run-time information. For example, the compiler can arrange for programs to collect run-time data during development and testing, and then use the collected profile information in optimizing the code for final delivery [Wal91]. Koopman and Lee obtained improvements in the performance of a lazy functional language by implementing graph reduction as self-modifying code [KLS92]. And, of course, there have been countless other applications of self-modifying code.

In this paper, we report on our experience with a new approach to generating optimized code at run time. We have implemented a prototype compiler, which we call FABIUS, that can automatically compile a general program into RISC machine code that in turn generates optimized machine code at run time. There are several notable examples of compilers for object-oriented languages that perform aspects of compilation at run time, including the Smalltalk-80 system by Deutsch and Schiffman [DS84] and the SELF compiler by Chambers and Ungar [CU89]. The approach we have taken differs in a number of crucial ways. Perhaps most fundamentally, we compile a functional programming language and hence are able to take advantage of previously developed techniques for compiling and transforming functional programs, including aspects of offline partial evaluation [JSS89, JGS93]. This also facilitates the development of an automatic staging analysis that allows code to be dynamically generated for any part of a program, rather than being restricted to particular points such as method (procedure) invocations.

Other salient characteristics of our approach, which we term *deferred compilation*, are as follows:

- It is *automatic*. No programming or programmer intervention is required. An automatic staging analysis determines those parts of the program to be subjected to run-time compilation, with or without the guidance of the programmer.

- It is *general*. Dynamic code generation is not limited to particular constructs or code templates. Furthermore, many standard compilation techniques, such as register allocation and inlining, can be employed.

- It is *fast.* No general compilation or processing of the source program occurs at run time. Rather, each part of the compiled code that performs run-time code generation is specialized to optimize and generate code for a small portion of the input program.

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 find that there is much further work to be done.

To introduce deferred compilation, we begin with a simple example that illustrates the basic points. Then in Section 3 we give an overview of some strategies and techniques for deferred compilation. Our desire to keep the cost of run-time code generation as low as possible leads to several important practical considerations. In Section 4 we describe some of the details of a prototype implementation and present the results of preliminary experiments with the system. This is followed by sections on the secondary costs of run-time code generation and the connections between deferred compilation and partial evaluation.

2 An Example

A simple example illustrates some of the techniques employed by deferred compilation. Consider a program that contains a (tail-recursive) definition of the exponentiation function:

```
power (exp, base, accum) =
    if exp = 0 then accum
    else power(exp - 1, base, accum * base);
⋮
 ... power (e, b, 1) ...
```

A conventional compilation of `power` might yield the following machine code:[1]

```
power:  beq    r1, r0, L1       ; if exp = 0 goto L1
        sub    r1, r1, 1        ; exp = exp - 1
        mul    r3, r3, r2       ; accum = accum * base
        jmp    power            ; goto power
L1:     move   r1, r3           ; result = accum
        ret                     ; return
```

Suppose the program calls `power` repeatedly, but with the first argument changing more slowly than the second argument. This would arise, for example, in a loop where each iteration computes a new base and calls `power` without varying the exponent. One can also imagine a curried version of `power` which is applied to an exponent value and then passed to a mapping function. In such situations, we say that the first argument is computed in an *early* stage and the second argument is computed in a *late* stage.

A *staging analysis* can be used to identify such computation stages and label those subexpressions in the program that depend only on the early arguments, as opposed to those that require late argument values. In the case of just two stages, this labeling of early and late computations

[1] For simplicity of presentation, we assume an idealized RISC architecture with no delay slots; see Appendix A.

corresponds precisely to a binding-time analysis and annotation [Con93, NN92], and in fact our prototype compiler incorporates a binding-time analyzer.

Early computations are compiled in the normal way, but late computations are translated into code that *emits* the corresponding instructions at run time. In this example, since the exponent is available at an early stage, the conditional test and subtraction expressions are compiled normally, but the compilation of the multiplication expression is deferred to run time. In the simplest form of deferred compilation, we might obtain the following code:[2]

```
powgen: beq    r1, r0, L1
        sub    r1, r1, 1
        emit   mul    r3, r3, r2
        jmp    powgen
L1:     emit   move   r1, r3
        ret
```

Note that the only difference between `power` and `powgen` is that the multiplication instruction is emitted (perhaps many times) instead of being executed, as is the instruction that moves the accumulator to the result register. When called with `exp` = `5`, `powgen` completely unrolls the loop and generates code with all "constants" folded and all "dead code" eliminated:

```
        mul    r3, r3, r2
        mul    r3, r3, r2
        mul    r3, r3, r2
        mul    r3, r3, r2
        mul    r3, r3, r2
        move   r1, r3
```

Deferred compilation can be fast enough to pay off quickly. On a typical RISC architecture a fixed-argument instruction can be emitted in as few as four cycles (see Appendix A). Under this assumption the costs incurred by `powgen` are recovered after only three iterations of the run-time-generated code when `exp` = `5`.

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 loop unrolling 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. Another complication stems from the fact that real-world programs often contain many more than the two stages of computation exhibited by this example, a large number of which may benefit from run-time optimization. Thus, a conventional binding-time analysis is not, in general, powerful enough for our needs.

The next section discusses these issues in more detail and proposes several strategies for addressing them. We also examine how a wider range of optimizations and code generation techniques can be adapted to deferred compilation. The effectiveness of some of these techniques is examined in the context of a more realistic example in Section 4.

---

[2] 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. In this example the arguments in the emitted instruction are fixed; the first `emit` creates "`mul r3, r3, r2`", regardless of the current values of `r2` and `r3`.

3 Strategies for Deferred Compilation

The example above, though unrealistically simple, illustrates the basic elements of deferred compilation. First, a staging analysis is used to determine the stage at which each subexpression of a program is computed. In essence, this identifies data and control dependencies in the program and reveals in which stages run-time optimization may be of benefit. Second, no general-purpose compilation occurs at run time. Instead, parts of the source program are compiled into special-purpose code generators (*e.g.*, `powgen`), each customized to optimizing and generating code based on run-time values.

3.1 Stages of Computation

Multiple stages of computation occur naturally in both functional and imperative programs. For example, when a strict curried function `f` of type $\alpha \rightarrow \beta \rightarrow \gamma$ is applied to an argument `x`, a closure representing a value of type $\beta \rightarrow \gamma$ 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. Deferred compilation can therefore be viewed as an alternative to the conventional implementation of closures.[3]

A similar phenomenon occurs in programs with nested loops. The outer loop index is always computed before executing inner loops, and substantial benefits might be obtained by specializing inner loops to its value at each iteration. More deeply nested loops lead to more stages of computation.

Computation stages arise naturally from other programming language constructs. Macros in Scheme and other languages are early computation stages that have been manually identified by the programmer; macro expansion performs these computations before compiler optimizations are applied. In Standard ML the phase-distinction property of the modules sublanguage guarantees that arguments to functors are available at an earlier stage than arguments to functions [HMM90]. Hence, deferred compilation can be used to compile functors into code that will generate optimized function code at functor-application time. Functor application is similar to the linking of object code in conventional languages, the speed of which is not a high priority, so deferring highly aggressive optimizations to this stage appears practical [HBHM93]. Link-time optimization has also been studied by Srivastava and Wall [SW93].

In practice, programmers often arrange for computations to be staged so that the costs of early computations can be amortized over many late computations [CPW93]. For example, in a Standard ML implementation of a network communications system, Biagioni *et al.* [BHL93] describe the structure of a `send` procedure with the type

```
send : connection -> message -> unit
```

The computation is staged so that `send` analyzes the `connection` and then selects one of several possible message-sending procedures (of type `message -> unit`). Since many messages are usually sent on a connection, this allows the cost of connection-specific processing to be amortized over all of the message sends. Deferred compilation can exploit such staging even if it is not explicit in the program text.

[3] Appel has made a similar observation [App87], and "all code" closures have been proposed by Feeley and Lapalme [FL92].

We have restricted our attention to two computation stages in this paper in order to simplify the presentation. In general, however, programs exhibit many more stages, and deferred compilation can in principle exploit an arbitrary number. Consider the case of a function of three arguments, `f(x,y,z)`, in which the argument `x` changes more slowly than `y` which in turn changes more slowly than `z`. In this case it may be profitable to identify three computation stages (call them "early," "middle," and "late") and generate code for an **`fgen`** function that, given the first argument, generates the code of *another* specialized code generator.

3.2 Staging Analysis

Programs can have many stages of computation, and so a key problem is how to identify those for which deferred compilation will be profitable. This is similar to the problem of deciding where to inline procedures in conventional compilation [CHT91] and the automatic determination of specialization points during partial evaluation [JGS93, BD91]. But as we have seen, syntactic features of programming languages often provide clear indications of stages that can be usefully subjected to deferred compilation. In some cases the use of programmer-supplied hints, such as the use of curried function syntax, would also be useful.

Once useful program stages are identified, each subexpression of the program can be analyzed to determine (approximately) to what stage it belongs. This is essentially a dependency analysis: a subexpression that only depends upon values computed at or before stage n computes a value that also belongs to stage n. Although this is conceptually simple, approximations must be made so that the stages of computations involving recursion can be finitely computed. Hence, this propagation of staging information is best accomplished via a dataflow analysis or abstract interpretation. Of course, since the analysis is necessarily approximate, early stages might be assigned to some expressions that are actually late, and *vice versa*. Optimization opportunities are lost in the former case, and unnecessary run-time code generation occurs in the latter case. Hence, refining the precision of staging analysis is of fundamental importance.

Further technical details of the staging analysis problem are beyond the scope of this paper, but we refer the reader to the literature on conventional binding-time analysis [JSS89, Con93], which is precisely a staging analysis for the special case of two stages. We have, for the time being, restricted our attention to two stages, and we use a conventional binding-time analyzer in our prototype compiler with good results (see Section 4). Note, however, that in programs where there are more than two useful stages, a binding-time analysis forces distinct stages to be merged, thus causing opportunities for run-time code generation to be lost. To gain maximum benefit from deferred compilation, a generalization of binding-time analysis to an arbitrary number of computation stages is required.

3.3 Limitations of Static Specialization

The examples mentioned earlier showed some of the circumstances in which computation stages can be exploited by a compiler. We have yet to explain, however, why run-time compilation is needed. To see this, consider the alternative of using a source-to-source transformation instead of deferred compilation. For the `power` example, an effect similar to deferred compilation can be obtained by transforming `power` into the following code:[4]

[4] `nth(`i`, (`x_0`, `x_1`, ..., `x_n`))` yields x_i, the i^{th} element of a tuple)

```
powgen(exp) =
    nth (exp, (lambda (base) 1,
               lambda (base) base,
               lambda (base) base * base,          } k
               lambda (base) base * base * base,
               ⋮
              ))
```

This definition of powgen can be obtained by creating a table of specialized versions of power, each of which is created by choosing a value for exp from the set $\{0, 1, \ldots, k\}$ and then applying a partial evaluator [Bon93] to power and exp. Similar transformations might also be obtained by applying staging transformation [JS86], program bifurcation [DBV91], or procedure cloning [CHK93]. In either case, highly optimized definitions of the specialized functions can be obtained, which can then be compiled into high-quality machine code. Hence, one might expect this approach to be useful in the same situations as deferred compilation.

However, there are two practical problems in performing such a transformation automatically. First of all, there is the matter of choosing the set of values on which to specialize. In powgen, for example, there is no guarantee that the set $\{0, 1, \ldots, k\}$ is a good one, since the range of exponents that will be supplied at run time usually cannot be predicted. In fact, the specialization would not in general be on simple integer values, but possibly on arbitrary data structures.

A second problem is that all of the specialized functions must appear in the transformed source program. This incurs a serious cost in space usage, and is wasteful since only a few of the functions might be used in a single program execution. In practice, 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).

Hence, a key aspect of deferred compilation is to arrange for specialization to occur "on demand" (or "just in time"). Furthermore, our desire to minimize the cost of run-time code generation leads us to specialize the compilation process itself. In other words, we wish to avoid the overhead of manipulating source programs, which one finds in a general compiler, and instead create code generators that are specialized to optimizing a fixed piece of code based on run-time values.

One can consider incorporating conventional compilation techniques into specialized run-time code generators. In fact, one of the key design issues in deferred compilation is deciding how to apportion the costs of optimization and code generation between compile time and run time. In the next section we consider the particular case of register allocation.

3.4 Register Allocation for Deferred Compilation

Conventional compilers often use graph-coloring algorithms to assign variables to a limited number of registers [Cha82, CH84]. An *interference graph* is constructed, with nodes representing the lifetime ranges of variables and edges indicating where these ranges intersect. Any K-coloring of the interference graph is therefore a valid assignment of the variables to K registers. This section describes how such techniques can be applied when compilation is deferred.

3.4.1 Compile-Time Register Allocation

We first consider a strategy for performing all register allocation at compile time. The significant complication is that different stages in a program can use the same set of registers because their execution is not interleaved. For example, the powgen function presented in Section 2 can exploit the fact that computations involving the exponent and base belong to different program stages by assigning those variables to the same register:

```
powgen:  beq     r1, r0, L1
         sub     r1, r1, 1
         emit    mul     r2, r2, r1
         jmp     powgen
L1:      emit    move    r1, r2
         ret
```

The usual notion of lifetime ranges does not capture this distinction, since the staging being exploited is not explicit in the source program. For example, computations involving **exp** and **base** are textually adjacent but belong to different computation stages. Conventional register allocation algorithms may nonetheless be used for deferred compilation by simply modifying the construction of the interference graph. A standard lifetime analysis can be conducted without regard to the staging of the program, followed by an analysis that determines the program stage to which each variable belongs. During construction of the interference graph, edges are only added between overlapping lifetime ranges of variables from the same program stage.

3.4.2 Run-Time Register Allocation

Although compile-time register allocation leads to fast run-time code generation, it suffers several limitations. Inlining and loop unrolling may occur at run time, so an exact interference graph cannot be constructed at compile time. Also, fixing the register assignment of a function at compile time makes it difficult to inline in some contexts. For example, registers must be shuffled if the formal and actual parameters are assigned to different registers, and so forth. If the number of contexts in which a function will be inlined is small, compile-time code duplication combined with fixed register assignments can be effective, but in general the space required will be prohibitive.

It is therefore desirable to perform run-time register allocation in some cases. Although register allocation can be performed on a run-time intermediate representation of code, the cost of processing such a representation is likely to pay off only when the generated code is executed many times. A more efficient strategy is to perform register *allocation* at compile time but defer register *assignment* until run time. A static approximation of the interference graph can be constructed as described in the previous section, and the run-time code generators can be parameterized by some representation of the desired register mapping. For example, **powgen** can perform run-time register assignment as follows:

```
powgen:  beq     r1, r0, L1
         sub     r1, r1, 1
         emit    mul     r[r3], r[r3], r[r2]
         jmp     powgen
L1:      emit    move    r[r4], r[r3]
         ret
```

This function takes four arguments: the value of **exp** (in **r1**), the numbers of the registers assigned to **base** and **accum** (in **r2** and **r3**), and the number of the destination register (in **r4**). The **emit** pseudo-instruction used here determines the operands of the emitted instruction from the contents of the specified registers. This takes more time 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.

The representation of register mappings has a significant impact on the cost of run-time register assignment. In the above example, register mappings are maintained in registers throughout early stages of computation; instructions can be emitted quickly because no memory access is required to determine their arguments. It remains to be seen whether this savings will in general justify the increased register pressure suffered by early computations.

3.5 Specialized Run-Time Code Generators

Performing most of the work of register allocation at compile time can greatly improve the speed of run-time code generation. Many other conventional optimization and code generation techniques can be similarly adapted to deferred compilation. This section gives a brief overview of our work in this area.

We have generalized destination-driven code generation [DHB90] to produce specialized run-time code generators (henceforth simply called *generators*) that do not manipulate any representation of the source program at run time. The algorithm is surprisingly straightforward because it obeys staging annotations rather blindly. As an expression is traversed, "early" operations are converted to machine code that performs the appropriate computation, while "late" operations are compiled into code that emits the machine instructions that will eventually perform the computation.

As the example in Section 2 demonstrates, this simple technique produces highly effective run-time optimizations. These optimizations are more powerful that those found in many template compilers [KEH91], and eliminating the need for run-time processing of an intermediate representation or template can yield much faster code generation. Many conventional peephole optimizations, such as strength reduction and instruction selection, can easily be incorporated. For example, a generator can avoid emitting a multiplication involving a value x from an earlier stage if it takes the time to determine whether $x = 1$. The increased cost of such run-time optimizations must be weighed against their benefit; a staging analysis that determines where to aggressively optimize would facilitate such decisions.

A generator that emits native machine code in a single pass will be faster than one that builds an intermediate representation, performs analysis and optimization, and then generates machine instructions. However, it can be difficult to produce good quality native code in a single pass. Branches and procedure calls are problematic because the destination may be code that has not yet been compiled. Due to run-time inlining and loop unrolling the generator may not be able to predict where the target code will eventually be located, so run-time backpatching is necessary. Span dependent instructions are challenging for similar reasons. Good instruction scheduling is also difficult to achieve in a single pass. Although a schedule can be "hard-wired" into generators for straight-line blocks, scheduling across basic-block boundaries requires more general techniques.

We are also investigating the adaptation of inlining and loop unrolling algorithms to deferred compilation. In conventional compilers such techniques yield increased opportunities for optimization and improve the amortization of various computations, such as range checks. Our preliminary work suggests that similar benefits can be obtained by run-time inlining and loop unrolling. It can be difficult to statically determine where to inline or how far to unroll a loop. The use of run-time information to guide such decisions may prove to be of significant benefit. We have augmented the compile-time code generator described above with the partial evaluation technique of unfolding [BD91, JGS93], a form of inlining.

In some contexts it is impractical to inline a function yet still desirable to optimize it based upon the results of earlier computations. For example if a function is called at many 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. This strategy is commonly called specialization [JSS89]. Specialization also permits run-time-optimized code to be reused rather than regenerated, which saves both space and time. The memoization of run-time code generators is a simple way to achieve such reuse. Run-time memoization can be quite expensive, particularly when memoizing on structured data [Mal93]. Nevertheless, preliminary experiments indicate that it is worthwhile in some applications. The development of static and dynamic strategies for controlling memoization is an interesting open problem.

4 Implementation

We have implemented a prototype compiler called FABIUS[5] that incorporates many of the deferred compilation strategies described in the previous section, as described below. 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 compilers that perform more aggressive run-time optimizations.

The FABIUS source language is a rudimentary, strict, first-order functional language. Integers and pointers to heap-allocated structures are the only run-time values; FABIUS does not support arrays or assignment. We have currently limited our attention to two-stage programs, so that the problem of staging analysis becomes one of binding-time analysis [NN92]. The staging analysis also determines how function calls should be treated by the code generator. An aggressive heuristic is used to determine which function applications should be inlined: function calls in the branches of "late" conditionals are specialized, but all other calls are inlined [BD91]. All analysis is automatic, requiring no programmer intervention.

All register allocation and assignment occurs at compile-time; registers are assigned independently to variables in early and late computations. In keeping with the focus on fast run-time code generation, very few optimizations are applied at run time. The primary optimizations are "constant" propagation, "constant" folding, dead-code elimination, and function inlining. Loops are expressed as tail-recursive functions, so inlining effectively yields loop unrolling. We have ignored the issue of instruction scheduling for the moment; we assume an idealized RISC machine with no delay slots (see Appendix A). Run-time code generation occurs in a single pass; no intermediate representation is constructed and no analysis or optimization is performed on code after it is generated.

Our preliminary results are encouraging. As an example we consider vector-matrix multiplication, which is often used to implement matrix-matrix multiplication and is common in scientific computing applications. Berlin and Weise have investigated improvements to similar scientific code through compile-time application of partial evaluation [BW90]. Vector-matrix multiplication is a prime candidate for run-time code generation because the vector is fixed throughout the computation, and the loop that computes the inner product of the vector with a row or column from the

[5] 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.

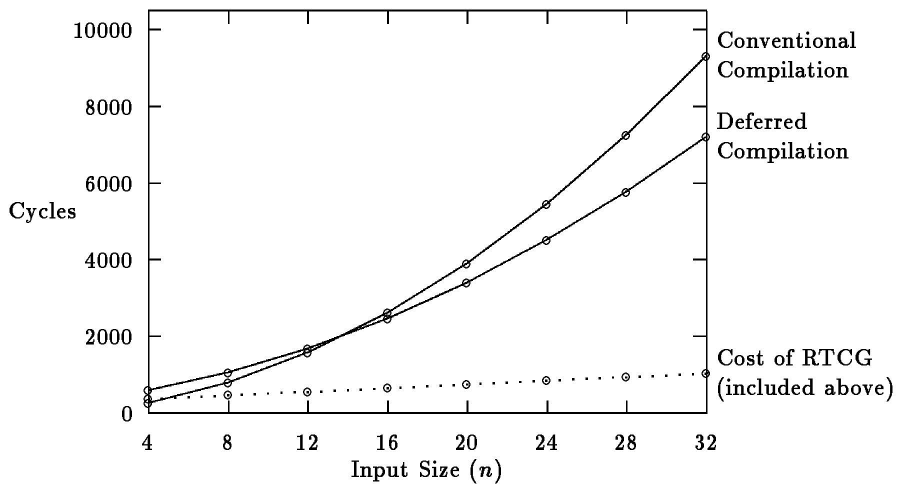

Figure 1: Time to multiply an n-vector with an $n \times n$ matrix

matrix can be completely unrolled.[6] Such optimizations cannot usually be performed at compile time, however, because the sizes and contents of the vector and matrix are generally not statically apparent. The source code for the example is given in Appendix B, along with one of the run-time code-generators produced by FABIUS.

Figure 1 compares the total execution time of vector-matrix multiplication for varying input sizes under conventional and deferred compilation. The inputs were vectors of length n and square matrices of dimension n containing pseudo-random integers, and the execution times are given in machine cycles (see Appendix A). The "conventionally compiled" code was produced by disabling the FABIUS staging analysis and is of high quality. The dotted line represents the portion of time spent generating code at run time; this time is included in the total execution time of the code produced by deferred compilation. As the figure demonstrates, deferred compilation can yield significant improvement in overall execution time even for small problem sizes. In this case the cost of run-time code generation was recouped when multiplying a 16 element vector with a 16$\times$16 matrix. The speedup increases linearly with larger input sizes (ignoring the secondary costs detailed in Section 5), yielding a speedup of greater than 20% when $n = 32$.

The amount of dynamically allocated data space was roughly the same under conventional and deferred compilation. However, as expected, we observed a significant increase in code size. The conventionally compiled code occupied just over 50 words; under deferred compilation the size of the static code rose to nearly 275 words and the size of the run-time-generated code rose linearly from 250 words to approximately 800 words as n ranged from 4 to 32. Increases of this magnitude are to be expected when aggressively inlining, since we are trading space for time, but it remains to be seen whether such increases are manageable in larger applications. More extensive experimentation is currently underway.

[6] The arithmetic operations can also be optimized based on the contents of the vector, which will likely yield substantial speedups for computations involving sparse data. The results presented here do not reflect such improvements, since we have focused on fast run-time code generation at the expense of some run-time optimizations.

5 Costs of Deferred Compilation

The time required to optimize and generate code at run time has been our primary focus, but the time/space tradeoff exploited by deferred compilation has some secondary costs that must be considered in practice.

Code-space reclamation can be a significant cost in programs that pursue aggressive run-time code generation. Conventional garbage collection techniques will likely suffice, although some new strategies may prove profitable because dynamically allocated code objects differ from data objects in both size and lifetime. Garbage collection might be complicated by the fact that run-time-generated code may contain embedded pointers to other data and code objects; this can occur if pointers are inlined like other values during optimization.[7]

Run-time code generation and modification can interact poorly with modern memory hierarchies [KLS92]. Most modern architectures prefetch instructions into an instruction cache and many do not automatically invalidate cache entries when memory writes occur. Cache flushing may therefore be required when dynamically generating or modifying code [Kep91]. The regularity of code-space allocation and initialization may simplify amortizing the cost of such operations. For example, the instruction cache could be flushed after code-space reclamation, and each newly allocated code object could be aligned to a boundary that the instruction prefetch mechanism is guaranteed not to have crossed while executing previously generated code, thus avoiding the invalidation of cached instructions. An architecture with a write buffer or a write-back data cache may require additional work to ensure that recently written instructions are fetched properly.

Another open question is how run-time code generation affects locality. Memory hierarchies offer substantial rewards to programs with highly localized data and instruction access patterns. Deferred compilation reduces locality by creating numerous optimized code blocks instead of executing a more general code block multiple times. Run-time inlining can increase code size significantly, thus decreasing locality. However, run-time dead-code elimination and other optimizations can also reduce code size. Techniques adapted from partial evaluation [Mog88] may also improve data locality by reorganizing data structures based on the staging of a program.

6 Deferred Compilation vs. Partial Evaluation

There are strong similarities between deferred compilation and offline partial evaluation [JGS93, BD91], but some significant differences deserve mention. A partial evaluator can be viewed as a generalized interpreter that, given a program and a portion of its input, produces a specialized *residual program* that accepts the remaining input and produces the desired result.

The correctness of a partial evaluator, called *mix* for historical reasons [JSS89], is described by the following equation, which specifies that the result produced by the residual program must be the same as the result of the original program p when applied to the same inputs ($[\![p]\!]$ denotes evaluation of a program p, yielding a function):

$$[\![[\![mix]\!](p, d_1)]\!] d_2 = [\![p]\!](d_1, d_2)$$

Perhaps the most intriguing aspect of partial evaluation is *self-application.* If *mix* is implemented in the language that it interprets, it can specialize *itself* to a particular source program p,

[7]These 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.

yielding a program that generates a residual program when executed:

$$[\![[\![mix]\!](mix, p)]\!]d_1 = [\![mix]\!](p, d_1)$$

This is the essence of our techniques for fast run-time code generation. $[\![mix]\!](mix, p)$ is a *generating extension* that when given the first input value will produce an optimized residual program. A generating extension is simply a specialized code generator, and it can be constructed at compile time because it does not depend on any run-time data. A further self-application of *mix* yields the stand-alone program, commonly called *cogen*, that performs the construction:

$$[\![[\![mix]\!](mix, mix)]\!]p = [\![mix]\!](mix, p)$$

In practice this approach has not been used to implement automatic run-time code generation. Typical partial evaluators [Bon93, Con88] are intended for source-to-source program transformation (see Section 3.3) and produce residual programs in Scheme or a similar high-level language. The generating extensions produced by self-application are therefore implemented in Scheme, and more importantly, they generate Scheme code when executed. The use of such systems for run-time code generation would therefore require general-purpose run-time compilation, which is too costly to be widely applicable.

Implementing a self-applicable partial evaluator that directly generates machine code would solve such problems.[8] Generating extensions would be directly executable and they would generate native code when executed. To the best of our knowledge, no such partial evaluator has been described or implemented to date. One system that comes closer than most 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.

A promising alternative to self-application is the hand-implementation of *cogen* [BW93]. In fact one can view FABIUS as a hand-implemented *cogen* whose target language is RISC machine code. This view is supported by our concentration on two-stage programs and our wholesale adoption of numerous techniques originally developed for partial evaluators, such as binding time analysis, unfolding heuristics, and memoized specialization. However, the goals and strategies of FABIUS, such as one-pass native-code generation and static register allocation, differ from those of any existing formulation of *cogen*.

7 Conclusions

We have developed a new approach to compilation. It provides an alternative to compile-time analysis and optimization by deferring aspects of optimization and code generation to run time. Automatic staging analysis is employed to detect program stages in which run-time optimization may be beneficial. Fast run-time optimization and code generation is achieved by eliminating 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.

[8] Note that such a partial evaluator need not be *implemented* in its target language.

Acknowledgements

We are grateful to Chris Stone, who provided valuable assistance in the implementation of FABIUS, and also to Ali-Reza Adl-Tabatabai, Chris Colby, Olivier Danvy, Greg Morrisett, Chris Okasaki, Amr Sabry, Chris Stone, and David Tarditi for their time and effort in productive discussions. We are indebted to Harry Bovik, who suggested that we use the term *deferred* instead of *retarded*.

References

[App87] Andrew W. Appel. Re-opening closures. Technical Report CS-TR-079-87, Department of Computer Science, Princeton University, 1987.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, September 1991.

[BHL93] Edoardo Biagioni, Robert Harper, and Peter Lee. Standard ML signatures for a protocol stack. Technical Report CMU-CS-93-170, Computer Science Department, Carnegie Mellon University, October 1993.

[Bon93] Anders Bondorf. Similix manual, system version 5.0. Technical report, DIKU, University of Copenhagen, Denmark, 1993.

[BW90] Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993.

[CH84] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232. SIGPLAN Notices, June 1984.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982.

[CHK93] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, April 1993.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software — Practice and Experience*, 21(6):581–601, June 1991.

[Con88] Charles Consel. New insights into partial evaluation: The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Springer-Verlag, March 1988.

[Con93] Charles Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, June 1993.

[CPW93] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46. Association for Computing Machinery, June 1993.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland*, pages 146–160, June 1989.

[DBV91] Anne De Niel, Eddy Bevers, and Karel De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 142–153. SIGPLAN Notices, September 1991.

[DHB90] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Computer Science Department, Indiana University, January 1990.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk–80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City*, pages 297–302, January 1984.

[FL92] Marc Feeley and Guy Lapalme. Closure generation based on viewing lambda as epsilon plus compile. *Computer Languages*, 17(4):251–267, October 1992.

[HBHM93] Nicholas Haines, Edoardo Biagioni, Robert Harper, and Brian G. Milnes. Note on conditional compilation in Standard ML. Technical Report CMU-CS-93-172, Computer Science Department, Carnegie Mellon University, June 1993.

[HMM90] Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco*, pages 341–354, January 1990.

[Hol88] N. Carsten Kehler Holst. Language triplets: The AMIX approach. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 167–185. North-Holland, October 1988.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[JS86] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, January 1986.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[Kep91] David Keppel. A portable interface for on-the-fly instruction space modification. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, April 1991.

[KLS92] Philip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[Mal93] Karoline Malmkjær. Towards efficient partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 33–43. Association for Computing Machinery, June 1993.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[Mog88] Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, October 1988.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[NN92] Flemming Nielson and Hanne Riis Nielson. Two-level functional languages. *Cambridge Tracts in Theoretical Computer Science*, 34, 1992.

[PLR85] Rob Pike, Bart Locanthi, and John Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software — Practice and Experience*, 15(2):131–151, February 1985.

[SW93] A. Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[Tho68] Ken Thompson. Regular expression search algorithm. *Communications of the Association for Computing Machinery*, 11(6):419–422, June 1968.

[Wal91] David W. Wall. Predicting program behavior using real or estimated profiles. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto*, pages 59–70, June 1991.

Appendix A Idealized RISC

The details provided in this appendix may assist interpretation of the example in Section 2 and the results presented in Section 4. FABIUS currently generates code for an idealized RISC machine that closely resembles the MIPS architecture. The primary difference is a lack of delay slots: memory access, branch, and call instructions require only one cycle to complete. The idealized RISC also supports a richer instruction set, including operations like `move`, `push`, and `call`; procedure linkage uses the stack.

The `emit` pseudo-instruction is interpreted by our RISC simulator rather than being expanded by the code generator, which facilitates the investigation of various peephole optimizations. The timings described in Section 4 attribute a cost of four cycles and a size of four words to most `emit` instructions. On the MIPS, two cycles would be required to load the 32-bit representation of a fixed-operand instruction into a register. Two additional cycles are required to store the instruction and update a code-segment pointer; the pointer update fills the delay slot of the store instruction. The cost of updating the pointer could be amortized over several `emits`, so we can reduce the average cost if another instruction is available to fill the delay slot. Fast allocation of code space is a critical requirement. We assume a garbage-collected code segment with amortized or hardware-supported overflow checking and cache flushing.

Appendix B Extended Example

This section details the vector-matrix multiplication example presented in Section 4. Although FABIUS does not yet support arrays, a realistic evaluation of the benefits of run-time code generation can be made using other data structures, so we have implemented vectors as linked lists and matrices as lists of vectors in row-major order:

```
vm-mult(v, m, a) =
   if m = nil then reverse(a, nil)
   else let prod = dotprod(v, car m, nil)
        in vm-mult(v, cdr m, cons(prod, a))

dotprod(v1, v2, a) =
   if v1 = nil then a
   else dotprod(cdr v1, cdr v2, a + car v1 * car v2)
```

The functions are implemented using tail-recursion to reduce procedure-call overhead; the accumulator can be viewed as an explicit encoding of call frames, the reversal of which corresponds to a sequence of procedure returns (The code for `reverse` has been omitted). `vm-mult` computes the dot product of the specified vector with each row of the given matrix and accumulates the results in a list. `dotprod` simply sums the products of corresponding elements of two vectors.

FABIUS creates memoized code generators for both `vm-mult` and `dotprod`; previously generated code for a particular vector is reused rather than regenerated. An inlining code generator is also created for `dotprod`; it is invoked by the memoized `dotprod` generator to generate code for its recursive tail-call (comments added):

```
L9:     beq     r1, r0, L10             ; if v1 = nil goto L10
        ld      r2, (r1)                ; x1 = car(v1)
        ld      r1, 4(r1)               ; v1 = cdr(v1)
        emit    ld      r3, (r1)        ; emit "x2 = car(v2)"
        emit    ld      r1, 4(r1)       ; emit "v2 = cdr(v2)"
        emit    move    r4, [r2]        ; emit "tmp = [x1]"
        emit    mul     r3, r4, r3      ; emit "prod = tmp * x2"
        emit    add     r2, r2, r3      ; emit "a = a + prod"
        jmp     L9                      ; goto L9
L10:    emit    move    r1, r2          ; emit "result = a"
        ret                             ; return
```

The first argument vector is supplied in `r1`. The run-time-generated code expects the second argument vector in `r1` and the accumulator in `r2`. If the first argument vector is `[1,2,3]`, the following code is generated at run time:

```
        ld      r3, (r1)                ; x2 = car(v2)
        ld      r1, 4(r1)               ; v2 = cdr(v2)
        move    r4, 1                   ; tmp = 1
        mul     r3, r4, r3              ; prod = tmp * x2
        add     r2, r2, r3              ; a = a + prod
        ld      r3, (r1)
        ld      r1, 4(r1)               ; etc.
        move    r4, 2
        mul     r3, r4, r3
        add     r2, r2, r3
        ld      r3, (r1)
        ld      r1, 4(r1)
        move    r4, 3
        mul     r3, r4, r3
        add     r2, r2, r3
        move    r1, r2                  ; result = a
```