Optimizing ML with Run-Time Code Generation*

Peter Lee Mark Leone

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

petel@cs.cmu.edu mleone@cs.cmu.edu

Abstract

We describe the design and implementation of a compiler that automatically translates ordinary programs written in a subset of ML into code that generates native code at run time. Run-time code generation can make use of values and invariants that cannot be exploited at compile time, yielding code that is often superior to statically optimal code. But the cost of optimizing and generating code at run time can be prohibitive. We demonstrate how compile-time specialization can reduce the cost of run-time code generation by an order of magnitude without greatly affecting code quality. Several benchmark programs are examined, which exhibit an average cost of only six cycles per instruction generated at run time.

1 Introduction

In this paper, we describe our experience with a prototype system for run-time code generation. Our system, called FABIUS, is a compiler that takes ordinary programs written in a subset of ML and automatically compiles them into native code that generates native code at run time. The dynamically generated code is often much more efficient than statically generated code because it is optimized using run-time values. Furthermore, FABIUS optimizes the code that dynamically generates code by using partial evaluation techniques, completely eliminating the need to manipulate any intermediate representation of code at run time. This results in extremely low overhead: the average cost of run-time code generation is about six instructions executed per generated instruction.

Although not every program benefits from run-time code generation, we have had little trouble finding realistic programs that run significantly faster—sometimes by more than a factor of four—when run-time code generation is used. In some cases, the ML programs compiled by FABIUS even outperform corresponding C programs. For example, the BSD packet filter interpreter, which the BSD operating-system kernel uses for fast selection of network packets on behalf of user processes [29], runs nearly 40% faster on typical inputs when translated from C to ML and compiled by FABIUS.

This paper describes our early experience with the FABIUS compiler, with a particular emphasis on low-level optimization and code-generation issues. We begin with a brief discussion of previous approaches to run-time code generation. Then we describe, using a running example, some of the compilation strategies implemented in FABIUS. Of course, it will not be possible to catalog all of our design decisions in this paper, but the example should illuminate some key problems, insights, and implementation techniques. Next, we present the results (both good and bad) of evaluating FABIUS using several benchmark programs. We focus initially on two programs: matrix multiplication and network packet filtering. Besides showing good results, these programs vividly illustrate the potential of this technology. We then discuss briefly the performance of several other benchmarks and mention the particular lessons learned from each. Finally, we describe current related work in detail, and conclude with a number of directions for further research.

2 Previous Approaches

Like the informal principle of the time-space tradeoff, there is also a tradeoff between general-purpose and special-purpose programs. General-purpose programs are desirable and even necessary for many applications. As a simple example, consider the choice between a general numerical-analysis routine and one that has been written specially for sparse inputs. The special version can be much faster, but might not work well in cases where the inputs are dense. Since generality usually incurs a penalty in run-time performance, programmers still take the trouble to write efficient specialized programs and use them whenever possible. Unfortunately, it is not always possible to predict ahead of time whether the specialized versions can be used.

Of course, these tradeoffs are as old as computer programming, and so is the idea of attacking them with general-purpose programs that generate special-purpose code at run time. For example, in 1968 Ken Thompson implemented a search algorithm that compiled a user-supplied regular expression into an executable finite-state machine in the form of native code for the IBM 7094 [35]. This program was both

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. 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 Advanced Research Projects Agency or the U.S. Government.

To appear in the ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, May 1996.

general (because it accepted any regular expression) and fast (because it generated special-purpose code quickly).

Run-time code generation also led to notable performance improvements in the areas of operating systems [4, 8, 26, 33, 36], method dispatch in object-oriented systems [1, 9, 13, 20], instruction-set simulation [10], graphics [14, 31], and many other applications. With the emergence of highly distributed and Web computing, more applications demand software that is general-purpose, safe, and highly composable. This trend has prompted increased interest in run-time (and link-time) optimization and code generation as a way to regain the performance advantage enjoyed by special-purpose, monolithic systems [5]. Additional arguments for run-time code generation have been well-summarized by Keppel and colleagues [23, 24]

2.1 General-Purpose Dynamic Compilation

Despite the increasing attention, researchers have done relatively little to automate and optimize the process of run-time optimization and compilation. Indeed, existing approaches, although quite effective for some applications, are themselves rather specialized and require significant effort on the part of the programmer.

A standard approach is to provide programming support for constructing representations of programs at run time, and for invoking an optimizing compiler to transform such representations into native code. This approach is relatively easy to implement, and systems such as DCG [17] based on this approach have been shown to be useful in interesting applications. Using DCG, a C programmer can write a program that constructs intermediate code trees and then invokes a compiler back-end to translate them into optimized native code that can be dynamically linked and executed.

Lisp systems have long provided programmer support for this style of programming, using backquoted S-expressions and `eval`. The advantage of S-expressions is that the programmer can specify the construction of Lisp terms at the source level, rather than as intermediate code trees. A similar features has also been implemented as an extension of Standard ML [3]. Recently, Engler and colleagues added support for this style of run-time code generation on top of DCG, in a system called 'C [16], with good results. An even better approach is to design a programming language so that run-time code generation can be performed without programmer assistance. For example, in an implementation of SELF [9], the system provides run-time compilation of type-specialized versions of methods by simply postponing most aspects of optimization and compilation until run time.

The problem with this basic approach is that the cost of code generation at run time can be high, to the point that the improvements gained by delaying compilation to run time are eliminated by the cost of run-time compilation. For example, DCG's reported overhead of generating an instruction at run time is about 350 instructions per instruction generated [17].

2.2 Templates

It is possible to reduce the cost of run-time code generation by "pre-compiling" (as much as possible) the code that will be generated at run time. Previous work has focused on the use of *templates*, which are sequences of machine instructions containing "holes" in place of some values. Code is generated simply by copying templates and instantiating holes with values computed at run time; templates may also be concatenated to effect loop unrolling and function inlining. Several systems have used this approach with great success, such as Pike, Locanthi, and Reiser's `bitblt` compiler [31] and in Massalin and Pu's Synthesis kernel [27].

Until very recently, the use of templates has imposed a significant burden on programmers. Templates and the code that instantiates them are typically constructed manually, which is non-portable and error prone. Recent work has explored the automatic derivation of templates from programs written in higher-level languages. For example, the Tempo system [11, 36] uses `gcc` to create machine code templates corresponding to portions of ordinary C programs that are classified as dynamic by a binding-time analysis. Chambers and colleagues adopt a similar approach, using the Multiflow compiler to generate templates for programmer-delimited dynamic regions in Modula-3 programs [4, 8].

A major drawback of templates is that they severely limit the range of optimizations that may be applied at run time. A template fixes a particular instruction schedule and chooses fixed encodings for instructions, which precludes optimizations such as run-time code motion and instruction selection. It is possible to pre-compile several alternative templates for the same code sequence and choose between them at run time, but to our knowledge this approach has never been attempted in practice. Furthermore, instantiating templates involves a certain amount of overhead, because a template is essentially a low-level intermediate representation of a code sequence. Instructions must be copied from the template during code generation, and table lookup is required to locate and instantiate holes. As we shall demonstrate, this interpretive overhead can be largely eliminated.

2.3 Specialized Code Generators

We advocate a more general approach to reducing the cost of run-time code generation that is based on ideas from the literature on partial evaluation [21]. A partial evaluator is a function (called *mix* for historical reasons) that takes the code of a two-argument function, f, and the value of its first argument, x, and returns optimized code for the result of *partially* applying f to x:

$$mix(f, x) \ = \ f_x$$

The result of this optimization is code for a *specialized* one-argument function, f_x, that when given an argument, y, computes the same result as $f(x, y)$, but which does so without repeating computations that depend only on x.

In a similar way, partial evaluation can be used to reduce the cost of performing specialization and code generation at run time. If the text of a function f is known at compile time, the (run-time) computation of f_x can be optimized by specializing *mix* to f at compile time:

$$mix(mix, f) \ = \ mix_f$$

The result is the code for a specialized code generator, mix_f, called a *generating extension*, that can be used to compute f_x without the overhead of processing the text of f, but retaining all of the optimizations. In essence, the code for f is "pre-compiled" away, so that it is not needed when the argument x is later supplied.

The FABIUS compiler is not a partial evaluator, but it does create specialized run-time code generators (as generating extensions) that do not manipulate templates nor any other intermediate representation of code at run time. A similar approach has also been employed recently by the `tcc` compiler for 'C [15, 32]. In contrast to `tcc`, Fabius achieves run-time code generation from completely ordinary ML programs (rather than depending on language extensions), and is more systematic in its use of well-known techniques and heuristics from partial evaluation. These include memoization of specialization, unfolding of static conditionals, and the residualization of static values in dynamic contexts [7].

3 The Fabius Compiler

We have previously given a high-level overview of our approach, which we call *deferred compilation* [25]. The main goals of deferred compilation are to minimize the costs of run-time code generation while allowing a wide range of optimizations in both the statically and dynamically generated code. The goal of the Fabius[1] compiler is to determine the feasibility of this approach, and to provide a way to test various design choices.

Fabius compiles a pure, first-order subset of ML with integers, reals, vectors, and user-defined datatypes. We chose this language primarily for two reasons. First, we are involved in a more general project to investigate the practicality of ML for systems programming (specifically, network protocols) [6]. Thus, we have a special interest in the optimization of ML programs. Second, ML provides good support for expressing staged computations in a way that might be usefully exploited by run-time code generation. When a function f of type $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ is applied to an argument $x : \tau_1$, it may be profitable to generate optimized code for the result of $f(x)$ rather than simply building a closure. Even though our current language is first-order, we allow function definitions to use currying to express staging, and then compile such functions into run-time code generators. The restriction to a pure subset of ML is not fundamental to our approach; it merely eliminates the need to analyze evaluation-order dependencies, which simplifies implementation. Also, the lack of mutable data structures avoids any need for an analysis of sharing or aliasing, and furthermore allows run-time optimizations to copy values freely.

3.1 A Simple Example

We shall use a simple example, an integer dot-product function, to demonstrate how Fabius constructs specialized run-time code generators. The dot product of two vectors is simply the sum of the products of their corresponding elements. In ML, it is typically implemented using a tail-recursive auxiliary function, as follows:

```
fun dotprod (v1, v2) =
    loop (v1, v2, 0, length v1, 0)

and loop (v1, v2, i, n, sum) =
    if i = n then sum
    else loop (v1, v2, i + 1, n,
               sum + (v1 sub i) * (v2 sub i))
```

[1]The compiler is named after Quintus Fabius Maximus, who 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.

The auxiliary function, `loop`, is parameterized by the vectors `v1` and `v2`, an index `i`, the length of the vectors `n`, and an accumulating parameter `sum` that tallies the result.

The dot-product operation is an especially attractive candidate for run-time code generation because it is frequently the innermost loop of long-running numerical computations. Matrix multiplication, for example, is often implemented as a triply nested loop: the outer two loops select a vector from each matrix and the inner loop computes their dot product. The vector selected by the outermost loop will be used to compute many dot products, so it may be profitable to create a specialized function for each such vector.

To see more clearly the potential for a code improvement, suppose the value of the vector `v1` is fixed. Because the loop index `i` depends only on the length of `v1`, which is fixed, the loop can be completely unrolled. Then, constant propagation and constant folding can eliminate the computation of `(v1 sub i)`. Because subscripting in ML requires bounds checking, these optimizations can yield a substantial benefit. Finally, if `v1` is sparse, a simple reduction of strength eliminates the multiplication, addition, and subscripting of `v2` whenever `(v1 sub i)` is zero. All these optimizations can be performed only at run time, once the values of `v1`, `i`, and `n` have been computed.

The first step taken by Fabius is to identify the "early" computations that can be performed by the statically generated code and the "late" computations that must appear in the dynamically generated code. This is a difficult task to automate because it requires global knowledge of the implicit staging of computations in a program.[2] For the moment, Fabius relies on the programmer to provide simple hints that allow a local analysis to uncover the required information. Syntactically, these hints are given by declaring functions so that they take their arguments in curried form.

When a curried function is applied to its first argument—referred to as the "early" argument—it invokes a code generator to create a specialized function that is parameterized by the remaining "late" arguments. Fabius employs a dependency analysis to extend this classification of function parameters to the body of the function, thereby producing an annotation of each subexpression as being either early or late. For example, the dot-product function is annotated as follows, where an overline indicates an **early** annotation and an underline indicates a **late** annotation:

$$
\begin{array}{l}
\texttt{fun dotprod } \overline{\texttt{v1}}\ \underline{\texttt{v2}}\texttt{ =} \\
\qquad \overline{\texttt{loop}}\texttt{ (}\overline{\texttt{v1}}\texttt{, }\overline{\texttt{0}}\texttt{, }\overline{\texttt{length v1}}\texttt{) (}\underline{\texttt{v2}}\texttt{, }\underline{\texttt{0}}\texttt{)} \\
\\
\texttt{and loop (}\overline{\texttt{v1}}\texttt{, }\overline{\texttt{i}}\texttt{, }\overline{\texttt{n}}\texttt{) (}\underline{\texttt{v2}}\texttt{, }\underline{\texttt{sum}}\texttt{) =} \\
\qquad \overline{\texttt{if}}\ \overline{\texttt{i = n}}\texttt{ then }\underline{\texttt{sum}} \\
\qquad \texttt{else }\overline{\texttt{loop}}\texttt{ (}\overline{\texttt{v1}}\texttt{, }\overline{\texttt{i + 1}}\texttt{, }\overline{\texttt{n}}\texttt{)} \\
\qquad\qquad\qquad \texttt{(}\underline{\texttt{v2}}\texttt{, }\underline{\texttt{sum + }\overline{\texttt{(v1 sub i)}}\texttt{ * (v2 sub i)}}\texttt{)}
\end{array}
$$

Note the use of a curried definition to express the staging of computation in the `loop` function. The variables `i` and `n` are specified as early parameters, because they depend only on the length of `v1`, so the conditional test is also early, and so on. Function applications have also been given **early** annotations, which is taken as an indication that they should be inlined at run time. Determining the treatment of function calls is actually a rather subtle problem, which we do not discuss here. (See [25] for details.)

[2]In particular, this *staging analysis* is more difficult than binding-time analysis because an initial division of program inputs is not supplied by the programmer [25].

After annotating the program, FABIUS compiles it into a register-transfer language, also containing **early** and **late** annotations, which in turn is translated into a representation of annotated MIPS assembly code. The annotated MIPS code for the dot-product loop is shown below. (In the interests of clarity, we have substituted symbolic register names and omitted the **early** annotations.[3])

```
loop: beq   $i, $n, L1
      sll   $i1, $i, 2         ; Shift to scale index
      addu  $p1, $v1, $i1      ; Compute address
      lw    $x1, ($p1)         ; Fetch number
      sll   $i2, $i, 2         ; Shift to scale index
      lw    $x2, $i2($p2)      ; Fetch, immed. offset
      mult  $prod, $x1, $x2
      add   $sum, $sum, $prod
      addi  $i, $i, 1
      j     loop
L1:   move  $result, $sum
      j     $ra                ; Return
```

This code is actually a specialized run-time code generator. The early instructions are simply executed, but the late instructions are *emitted* into a dynamic code segment, as explained below. For example, when supplied with the vector `v1 = [1, 2, 3]`, the following dot-product function is dynamically created:

```
lw    $x2, 0($v2)
mult  $prod, 1, $x2
add   $sum, $sum, $prod
lw    $x2, 4($v2)
mult  $prod, 2, $x2
add   $sum, $sum, $prod
lw    $x2, 8($v2)
mult  $prod, 3, $x2
add   $sum, $sum, $prod
move  $result, $sum
```

Even though this code is generated in a single pass by a relatively simple code generator, it is extremely efficient. All the loop operations are early, so the generated code is completely unrolled. All the operations involving the vector `v1` are performed by the code generator, and its elements appear as immediates in the emitted code. And because the index (`i`) is early, the byte offsets used to access `v2` are hard-wired into the generated code.

3.2 Emitting Native Code

Emitting instructions is quite simple. FABIUS adds initialization code to the compiled program to allocate a dynamic code segment; a dedicated register, called the code pointer (`$cp`), tracks the address of the next available word. Each late instruction is translated into code that constructs the native encoding of the instruction, writes it to the dynamic code segment, and advances the code pointer.[4] For example,

```
add   $sum, $sum, $prod
```

is emitted as follows:

```
li    $t0, x852020     ; Instruction encoding
sw    $t0, ($cp)       ; Write to code segment
addiu $cp, $cp, 4      ; Advance code pointer
```

Notice that no intermediate representation is required at run time. Of course, emitting code and then immediately executing it involves some subtle interactions with the cache-memory system and the instruction prefetching mechanism, which we discuss in Section 3.4. Overall, the cost of code emission is about six instructions per instruction generated, as additional overhead is incurred by run-time instruction selection (described below) and a memoization mechanism (discussed in Section 3.5).

Rather than loading a 32-bit immediate value (which usually requires two cycles on the MIPS), the instruction encoding could instead be copied from a template. Doing so is slightly more complicated, because a pointer to the template must be maintained in a register. Also, not all emitted instructions have a fixed encoding. For example, the following load instruction

```
lw    $x2, $i2($p2)
```

has an immediate offset that is specified by the value in `$i2`. It is emitted as follows:

```
lui   $t0, x8c66       ; Upper half is opcode
or    $t0, $t0, $i2    ; Load half is offset
sw    $t0, ($cp)
addiu $cp, $cp, 4
```

However, because the MIPS restricts immediate values to 16 bits, this encoding is valid only when the value in `$i2` is relatively small. The next section describes how FABIUS employs run-time instruction selection to determine when such instructions can be used.

3.3 Run-Time Instruction Selection

Much of the benefit of run-time code generation can be viewed as a kind of run-time loop invariant removal: the execution frequency of early computations is reduced in a way that cannot be achieved by compile-time optimizations. Another benefit of run-time code generation arises when early values are used in late computations. In such cases, data-dependent optimizations can yield code that is superior to statically optimized code.

In the preceding dot-product example, the use of an immediate-offset load instruction to load a value from `v2` is correct only when the byte offset of the element can be specified by a 16-bit immediate value. Otherwise, code must be generated to load the offset into a register (which requires two instructions on the MIPS), followed by a register-register addition and a load. FABIUS therefore translates

```
lw    $x2, $i2($p2)
```

into the code shown in Figure 1.

FABIUS implements a number of similar forms of run-time instruction selection. Of course, the benefits of such optimizations must be weighed against their cost, but in our experience their cost is quite low (because no intermediate representation is manipulated) and their benefit is often very significant. For example, run-time strength reduction greatly improves the performance of dot product on

[3] There are other simplifications as well: we omitted the code for subscript checking and run-time instruction selection, and used a single pseudo-instruction for multiplication that actually requires multiple instructions to implement on the MIPS. Also, untagged integers are used in this and all other examples (see Section 5). The astute reader may notice that this code is not optimal: it could be improved slightly by common subexpression elimination and better pointer arithmetic.

[4] For efficiency, multiple updates of the code pointer within a basic block are coalesced.

```
        li    $t0, x8000       ;  Scale i2 for
        addu  $t0, $i2, $t0    ;   unsigned comparison
        li    $t1, xffff
        sltu  $t0, $t1, $t0    ;  Does i2 fit in 16 bits?

        beq   $t0, $0, L1
        srl   $t0, $i2, 16     ;  Get upper half of i2
        lui   $t0, $t0         ;  Load upper half
        andi  $t0, $i2, xffff  ;  Get lower half of i2
        or    $t0, $t0         ;  Load lower half
        addu  $p2, $v2, $t0
        lw    $x2, ($p2)
        j     L2
L1:
        lw    $x2, $i2($v2)
L2:
```

Figure 1: Specialized run-time instruction selection

sparse input by completely eliminating the computation of `sum + (v1 sub i) * (v2 sub i)` whenever `(v1 sub i)` is zero. As we demonstrate in Section 4.1, this optimization allows the performance of a general-purpose ML implementation of matrix multiply to compete favorably with C code designed specifically for sparse matrices.

3.4 Instruction Caching and Prefetching

Unfortunately, modern cache architectures are not designed with run-time code generation in mind. Most memory systems employ separate data and instruction caches, and many do not invalidate cached instructions when memory writes occur, because it is assumed that self-modifying code is rare. The run-time code generators created by Fabius do not modify existing code, but it is crucial that they be able to reuse code space. Thus, invalidating or updating the instruction cache is necessary to ensure that it remains coherent with memory.

Coherency mechanisms vary, but portability does not appear to be a major concern [22]. The cost of instruction-cache invalidation varies widely, however. On the DECstation 5000/200, flushing the instruction cache requires a kernel trap plus approximately 0.8 nanoseconds per byte flushed [22]. Fortunately, it is relatively easy to amortize this cost. Because Fabius-generated code generators do not modify existing code, it is not necessary to flush individual words from the instruction cache as new instructions are written. Instead, when code is garbage collected the freed space can be invalidated in a single operation.

We have encountered several other technical challenges related to instruction caching. For example, as we have shown in earlier sections, run-time "constants" are commonly propagated into run-time generated code, where they appear as immediate values. Pointers to heap-allocated structures may also be propagated, so a garbage collector must update or invalidate code if the referenced data is copied to a different location. This might be difficult, however; on the MIPS a 32-bit pointer is split into two 16-bit immediate values whose location in the dynamically generated code can be unpredictable.

Instruction prefetching also complicates matters, because programs compiled by Fabius interleave code generation with the execution of generated code. Because instructions are usually cached on a line-by-line basis, several locations immediately preceding and following a run-time-generated function may be cached when it is executed. But the instructions that will appear in these locations may not yet have been written, so the cache will become incoherent. Fabius solves this problem by aligning each newly generated function to the start of an instruction-cache line.

3.5 Other Issues

In this section we mention some additional complexities that we have encountered in the implementation of Fabius. Due to space constraints, we can do little more than hint at the solutions we have adopted and possible alternatives.

The dot-product example shown previously is actually quite misleading in its simplicity. For example, the treatment of function calls was simplified by the use of inlining: the tail-recursive call in the `loop` function was merely compiled into a jump to the start of the code generator. Curried applications that are not inlined are instead compiled into two calls, the first to a *memoized* code generator and the second to the address returned by the first call. Memoization is currently implemented using a per-procedure log that records the entry points of previously optimized code and the values that were used to optimize it. For efficiency, values are compared using pointer equality to determine whether code can be reused. However, this strategy causes unacceptable duplication of effort in some benchmarks because structurally equivalent values fail to match.

One-pass run-time code generation is fundamental to our approach. As we have demonstrated, Fabius is able to completely eliminate the overhead of processing intermediate code at run time by "hard wiring" optimizations and instruction encodings into run-time code generators. However, this approach compromises our ability to generate high-quality code. For example, it is surprisingly difficult to avoid creating jumps to jumps when generating code for conditionals at run time. Other desirable optimizations, such as instruction scheduling, are difficult to accomplish in a single pass because the structure of the code is often determined by dynamic inlining.

Improved register usage is a benefit of run-time code generation that bears mention. The use of symbolic register names obscures this effect in the preceding examples, but it can be observed by considering their live ranges. Although early and late instructions are textually adjacent, they occupy different live ranges and may therefore use overlapping register assignments. For example in the dot-product loop, Fabius assigns the variables `v1` and `v2` to the same register. Currently all register assignments are determined at compile time, which makes instruction encodings easy to construct at run time. However, this practice defeats some of the advantages of run-time inlining: the registers that are free at one call site may be unavailable at another. Fabius is forced to insert spill code around inlined code in several of the benchmarks we have examined. We propose an efficient solution to this problem in [25].

4 Preliminary Results

To evaluate the performance of Fabius, we have experimented with a number of small and medium-sized benchmark programs. Because the current Fabius prototype does

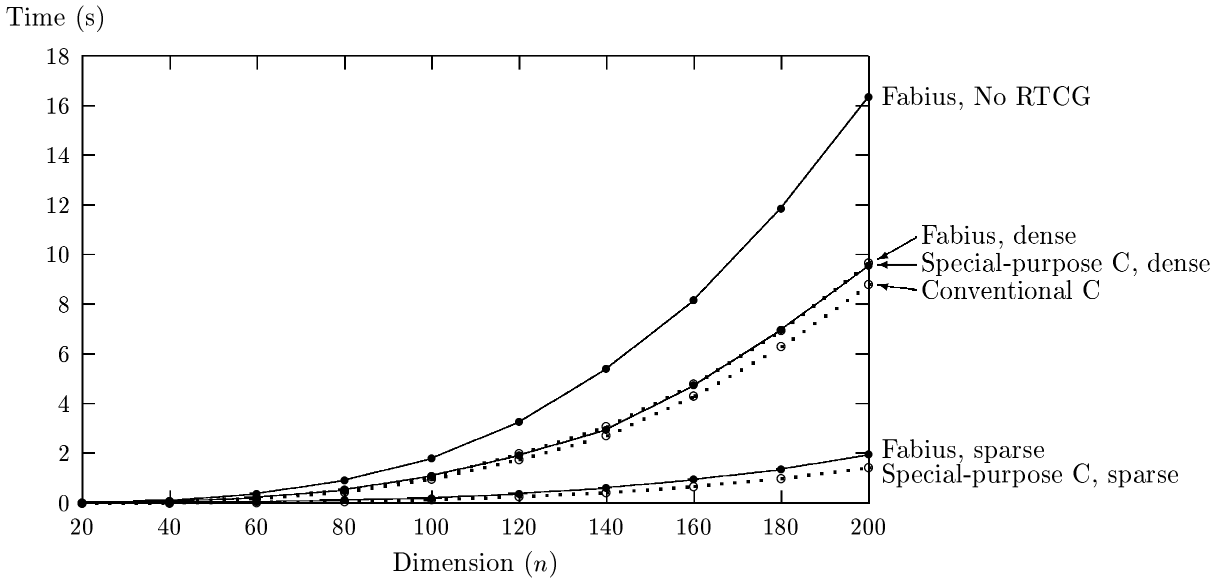

Figure 2: Time to multiply two $n \times n$ matrices (dense and sparse)

not yet have a garbage collector, the measurements we report here do not reflect the cost of reclaiming data and code space. Also, for reasons discussed in Section 5, we have not yet implemented a tagging scheme for integers and pointers. We note, however, that these issues probably have only a small effect on the measurements. For example, garbage collection of the code heap is unnecessary in the packet-filter benchmark, and thus its absence is of little consequence. These issues are discussed further in Section 5.

We executed the benchmark programs on an unloaded DECstation 5000/200 that was equipped with a timing board providing an accurate 12.5 MHz memory-mapped clock. This machine was not run in single-user mode, and so transient loads caused occasional variation of execution times. To minimize this effect, we selected the minimum elapsed time observed during five iterations of each benchmark trial.

4.1 Matrix Multiplication

We implemented a simple integer matrix multiplication routine in ML using the dot product function described in Section 3.1 and evaluated its performance, with and without run-time code generation, on both dense and sparse matrices. For comparison purposes, we also evaluated the performance of two matrix multiplication algorithms implemented in C. One was a conventional triply nested loop (in row-major order) and the other was a special-purpose algorithm based on indirection vectors [17], which is well-suited to sparse matrices that lack predictable characteristics.

The ML implementation was purely functional and used dynamically dimensioned arrays, which were implemented using immutable one-dimensional vectors; each subscript operation included two bounds checks. Both C implementations used statically allocated two-dimensional arrays (without bounds checking), and were compiled by `gcc` (version 2.3.2) with `-O2` optimization. The input matrices contained untagged pseudo-random 16-bit integers; sparse matrices were 90% zero, and the non-zero elements were randomly located.

Figure 2 compares the overall execution times, including time spent generating code, of the ML code compiled by FABIUSwith the C code, on both dense and sparse matrices. Without run-time code generation the ML code was nearly twice as slow as the conventional C code, mainly because of array-bounds checking. Run-time code generation significantly improved its performance on both dense and sparse inputs, even for small inputs. On dense input the FABIUS-generated code matched the performance of the special-purpose C code and was 10% slower than the conventional C code on 200×200 matrices. On sparse input the FABIUS-generated code was a factor of 4.5 faster than the conventional C code on 200×200 matrices, and only 39% slower than the special-purpose C code. Similar improvements were also observed for floating-point matrix multiply.

The time spent generating and managing code at run time was insignificant, representing less than one percent of the overall execution time. The cost of run-time code generation was recovered even for small matrices: the break-even point for $n \times n$ dense matrices was $n = 20$, and the break-even point for sparse matrices was $n = 2$. An average of 4.7 instructions were required to generate an instruction at run time. Space usage was significant, however. At $n = 200$, each dot-product function created at run time required 6.25 kilobytes (although better scheduling could reduce this by 25%). Efficient reuse of code space is clearly a requirement: without it, the total memory footprint is over a megabyte for large matrices.

4.2 Packet Filtering

A packet filter is a procedure invoked by an operating system kernel to select network packets for delivery to a user-level process. To avoid the overhead of context switching on every packet, a packet filter must be kernel resident. But kernel residence has a distinct disadvantage: it can be difficult for user-level processes to specify precisely the types of packets they wish to receive, because packet selection criteria can be quite complicated. Many useless packets may be delivered as a result, with a consequent degradation of performance.

A commonly adopted solution to this problem is to parameterize a packet filter by a selection predicate that is dynamically constructed by a user-level process [30, 29]. A selection predicate is expressed in the abstract syntax of a "safe" or easily verified programming language, so that it can be trusted by the kernel. But this approach has substantial overhead: the selection predicate is reinterpreted every time a packet is received.

Run-time code generation can eliminate the overhead of interpretation by compiling a selection predicate into trusted native code. More generally, run-time code generation can allow a kernel to efficiently execute "agents" supplied by user-level processes while avoiding context switches. Such an approach has also been investigated by others [5, 18].

To investigate the feasibility of this idea, we implemented the BSD packet filter language [29] using FABIUS and compared its performance to BPF, a kernel-resident interpreter implemented in C [28]. The interpreter shown in Figure 3 is a simple ML function, called `eval`, that is parameterized by the filter program, a network packet, and variables that encode the machine state. Note that `eval` is curried: when applied to a filter program and a program counter, the result is a *function* that is parameterized by the machine state and the packet. As with the matrix-multiply example, `eval` is written as an ordinary ML function, with no explicit indication of run-time code generation (in fact, it is still a legal ML program) and no specification of staging of computations beyond the currying of its arguments.

FABIUS compiles `eval` into a run-time code generator that evaluates all of the operations involving the filter program (such as instruction decoding) and generates code for all operations involving the packet data and the machine state. For example, consider the following filter program, which selects packets whose Ethernet type field specifies an IP packet:

```
      LD    4            ;  Accum. gets 5th pkt word.
      RSH   16           ;  Shift, yielding type field.
      JEQ   ETH_IP, L1   ;  Is this an IP packet?
      RET   1            ;  If so, accept.
L1:   RET   0
```

The following MIPS code is generated for this packet filter at run time:[5]

```
      move $t, 4
      slt  $t, $t, $n    ;  Is offset < pkt size?
      beq  $t, $0, L1
      lw   $a, 16($p)    ;  Accum. gets 5th pkt word.
      srl  $a, $a, 16    ;  Shift, yielding type field.
      move $t, ETH_IP
      beq  $a, $t, L2    ;  Is this an IP packet?
      move $r, 0         ;  Reject if not.
      j    L3
L2:   move $r, 1         ;  Else accept.
L3:   j    L4
L1:   li   $r, -1        ;  Indexing error.
L4:   jr   $ra
```

This code is not quite optimal because the run-time code generator has failed to eliminate two jumps whose targets are jumps. But it does demonstrate several interesting run-time optimizations:

- The most beneficial optimization is a kind of run-time loop invariant removal: all operations involving the packet filter and the program counter, such as instruction fetching and decoding, have been performed by the code generator.

[5] Symbolic register names have been added for clarity: \$a contains the accumulator, \$p is a pointer to the packet, \$n is the number of words in the packet, \$r is the result register, and \$t is a temporary. Delay slots are omitted; all are filled with nop instructions.

- Run-time "constant" propagation has embedded values from the filter program (such as the load offset, the shift amount, and the `ETH_IP` constant) as immediates in the run-time generated code.

- Run-time "constant" folding has also occurred: because the load offset is a run-time "constant," the code generator has folded the index scaling implicit in "`pkt sub k`." The resulting byte offset is used in a load-immediate instruction.

- Run-time inlining has eliminated all of the tail-recursive calls in `eval`. These jumps are performed by the code generator, not the generated code.

Larger packet filter programs show similar improvements. Figure 4 compares the overall execution times of the FABIUS packet filter implementation (including time spent generating code) to the BPF implementation in C, using a packet filter that selects non-fragmentary TCP/IP packets destined for a Telnet port. This packet filter must parse the IP header, the length of which may vary, to determine the location of the TCP destination port. To reliably compare execution times, we obtained five sample packet traces by eavesdropping on a busy CMU network, and we averaged execution times over these traces as a precaution against abnormal packet mixes. We modified the BPF interpreter to read packets from these trace files, compiled it with -O2 optimization, and executed it in user mode. The timings reported here exclude the time required to read packets from the trace files.

As the figure demonstrates, the time spent generating code at run time was quickly repaid: the FABIUS implementation broke even with BPF after approximately 250 packets. After only 1000 packets, execution time was reduced by 30.3%. The time spent generating code was brief, totalling 1.3 ms. Instrumentation revealed that the run-time code generator executed an average of only 5.6 instructions per instruction generated. The run-time generated code required an average of 8.3 μs to process a packet, whereas BPF averaged 13.7 μs per packet.

Space usage was insignificant. The abstract syntax of the packet filter program occupied 34 words in both implementations, and the FABIUS implementation generated 85 instructions at run time. The run-time code generator performed a small amount of heap allocation (43 words), but none was required by the run-time generated code. Stack usage was also insignificant, because tail-recursive calls were compiled into jumps.

4.3 Additional Benchmarks

We now briefly summarize our experience with seven additional benchmark programs written in ML. Some of the benchmarks are trivial in size, such as a three-line function for determining set membership, but are significant because of their widespread use in ML programs. Others represent real-world applications, such as a program that determines the three-dimensional structure of RNA molecules.

Figure 5 shows the overall execution times of six of the seven benchmarks for varying input sizes (the input size of the seventh benchmark cannot be varied). In the interest of brevity, we simply compare the performance of FABIUS-generated code with and without run-time code generation (using curried and uncurried functions, respectively), and we do not report details of space usage, which was generally

```
val eval : int vector * int -> int * int * int vector * int vector -> int

fun eval (filter, pc) (a, x, mem, pkt) =
 let val pc = pc + 2
 in
    if pc >= length filter then ~1 else
    let val instr = filter sub pc
        val opcode = instr >> 16
    in
        (*  Load packet word at immediate offset into accumulator. *)
        if opcode = LD_ABS then
            let val k = filter sub (pc+1)
            in
                if k >= length pkt then ~1
                else eval (filter, pc) (pkt sub k, x, mem, pkt)
            end
        ...
        (*  Jump if accumulator equals immediate. *)
        else if opcode = JEQ_K then
            if a = filter sub (pc+1) then
                eval (filter, pc + ((instr >> 8) andb 255)) (a, x, mem, pkt)
            else eval (filter, pc + (instr andb 255)) (a, x, mem, pkt)
        ...
    end
 end
```

Figure 3: Packet filter interpreter

The BSD packet filter language is comprised of RISC-like instructions for a simple abstract machine with an accumulator, an index register, and a small scratch memory. The abstract machine is made "safe" by confining memory references to the packet data and scratch memory and by forbidding backwards jumps. Instructions are encoded as pairs of 32-bit words. The first word specifies a 16-bit opcode and, optionally, a pair of 8-bit branch offsets. The second word specifies an immediate value that can be used for ALU operations and memory indexing.

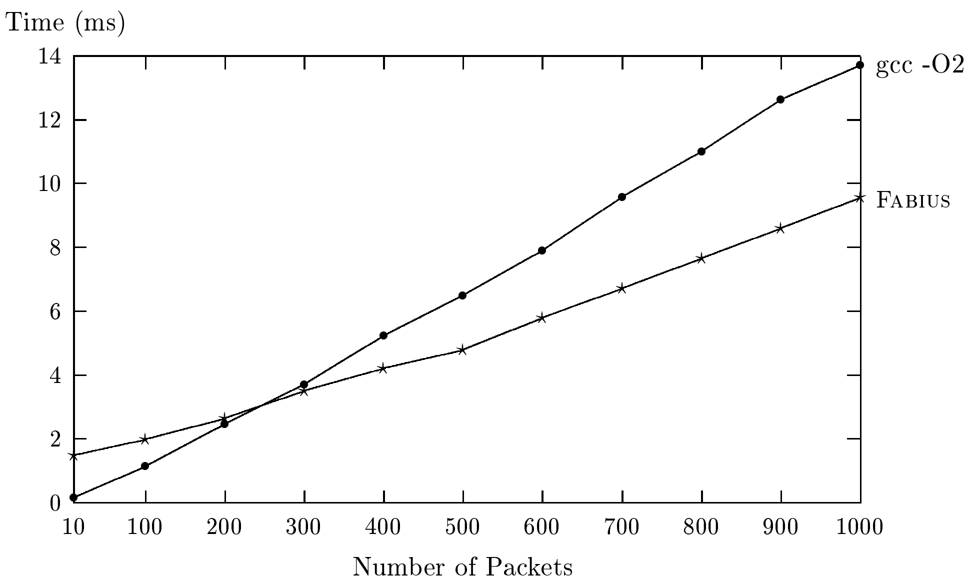

Figure 4: Run-time code generation for a packet filter

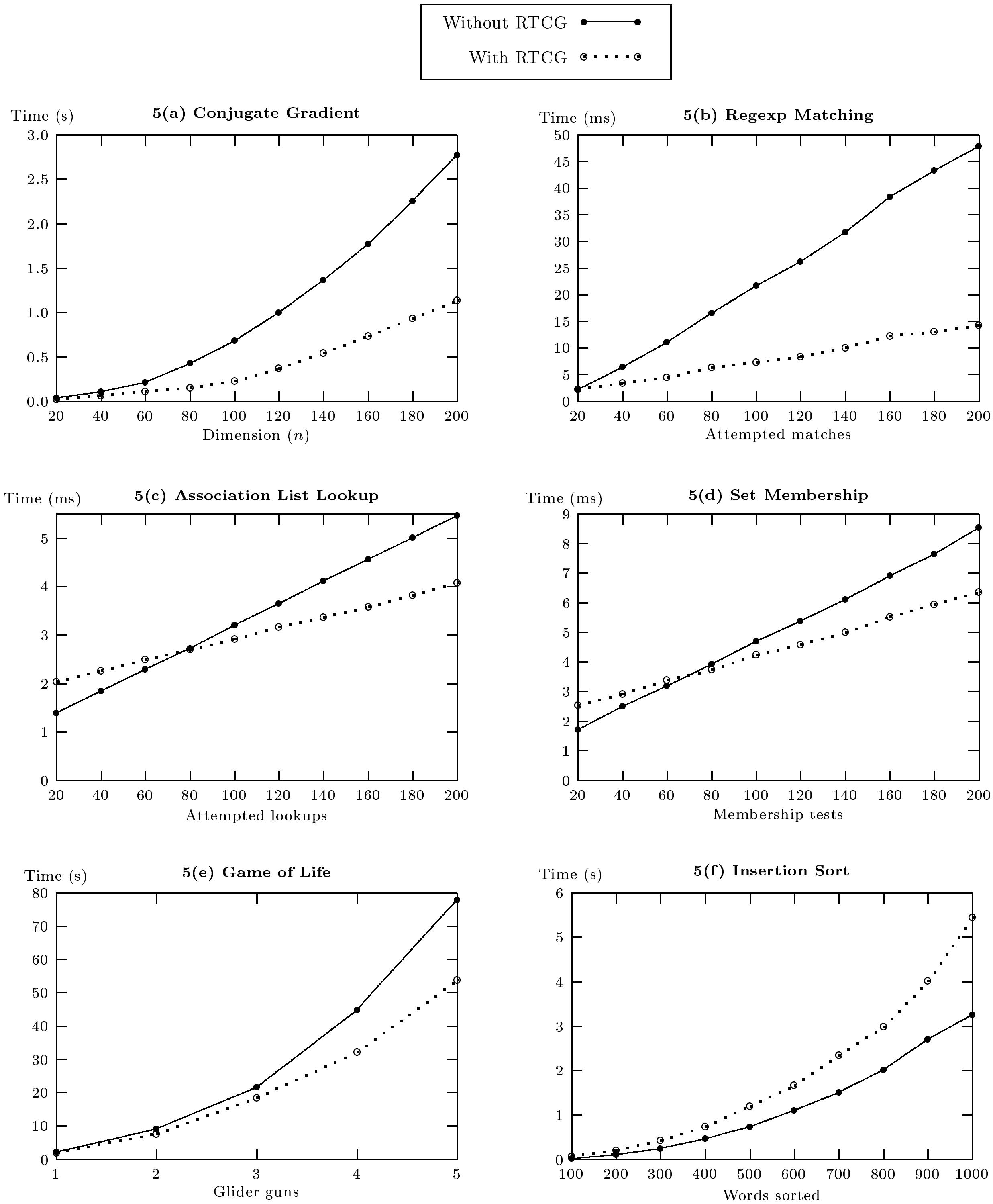

Figure 5: Additional Benchmarks

```
        li    $tmp, 3
        beq   $arg, $tmp, L5
        li    $tmp, 2
        beq   $arg, $tmp, L3
        li    $tmp, 1
        beq   $arg, $tmp, L1
        ...   ; Else signal an error.
L1:     li    $result, 1
L2:     j     L4
L3:     li    $result, 2
L4:     j     L6
L5:     li    $result, 3
L6:     jr    $ra
```

Figure 6: An executable association list

low. The total cost of run-time code generation varied from one benchmark to the next, but its relative cost was low, averaging roughly six cycles per generated instruction.

The first benchmark (Figure 5a) is an iterative solver for sparse linear systems of equations, adapted from [37]. It is based on the conjugate gradient method, which finds solutions to systems of equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square, symmetric, positive-definite matrix of double-floating-point values. This program is amenable to run-time code generation because the coefficient matrix, $\mathbf{A}$, is multiplied with the gradient direction vector on each iteration, but never varies. The system of equations used as input was tri-diagonal, and as the graph demonstrates, run-time code generation was able to exploit this sparsity, yielding a 2.4-fold speedup for a system of 200 equations. Because the cost of run-time code generation was low, performance on smaller systems of equations was also superior to the statically optimized code.

Figure 5(b) demonstrates the improvement obtained in a regular-expression matching algorithm using run-time code generation. Unlike Thompson's compiler for regular expressions [35], our program is a simple backtracking interpreter. FABIUS compiled it into a code generator that, given a regular expression, creates a finite state machine (in native MIPS code). The input was a regular expression describing words containing the five vowels in order (e.g. "facetious"), matched against the first n lines of `/usr/dict/words`. A significant speedup was observed even for small input sizes: the cost of run-time code generation was recovered after only 20 matches, and a 3.4-fold speedup was attained for 200 matches.

Association lists are a common data structure in ML programs; they are often used as symbol tables, mapping keys of one type to values of another. Looking up a value in an association list can be time consuming: two memory accesses are required to fetch the key of the entry at the head of the list, and if it does not match an additional memory access is required to locate the next entry. Figure 5(c) demonstrates that run-time code generation can amortize this overhead when multiple lookups are performed on the same association list. FABIUS automatically compiles a curried lookup function into a code generator that, in essence, creates executable data structures [26] that require no memory accesses; a sample of the lookup code generated at run time is contained in Figure 6.

The task of determining set membership is also well suited to run-time code generation, because repeated membership tests on the same set require duplicate effort that can be amortized. Figure 5(d) demonstrates the effect of run-time code generation on a simple curried membership function written in ML, and Figure 5(e) shows the performance improvement observed in a commonly used ML benchmark, Conway's game of life (adapted from [2]), which uses a set to record the locations of living cells.

Run-time code generation does not always improve performance, of course. One promising application we considered was an insertion sort, which repeatedly chooses a string from the head of a list and compares it to the rest of the strings in the list until a lexically greater one is found. If the lexical comparison function is curried, FABIUS compiles it into a code generator that performs all of the subscripting operations on the first string and unrolls the comparison loop, yielding a nest of comparisons involving constant characters. As Figure 5(f) demonstrates, however, optimizing the lexical comparison function at run time does not improve performance, despite using input that requires many repeated comparisons to sort (in this case, a reverse-sorted prefix of `/usr/dict/words`). In practice, most lexical comparisons do not examine more than two or three characters of each string, so the time spent generating code for the remaining characters is wasted.

We also experimented with the Pseudoknot benchmark [19], which is a floating-point intensive program that attempts to determine the three-dimensional structure of portions of RNA molecules using constraint satisfaction and backtracking search. Each step of the search investigates possible placements of a nucleotide by checking whether these placements violate certain constraints, which are specified as input to the program. The constraint-checking step is actually unnecessary for most nucleotides, but the general-purpose nature of the program does not allow this property to be exploited by compile-time optimizations. We used FABIUS to create a specialized search function that performs constraint checking only when truly necessary. However, doing so did not have the desired effect: overall execution time was reduced by approximately five percent (the problem size was fixed). This appears to be due to a peculiar property of the problem domain: nucleotides that do not require constraint checking have very few possible placements, so the overhead eliminated by run-time code generation was not large enough to repay the cost of run-time optimization.

Even when no significant improvements are to be had, the low cost of run-time code generation means that comparable performance is still possible. This greatly reduces the risk of automating the staging decisions in the compiler.

5 Conclusions and Future Work

We have implemented an approach to run-time code generation that is both principled and practical. The use of ML allows the compiler to perform run-time optimizations with little effort on the part of the programmer. It also facilitates a substantial use of partial evaluation techniques to optimize the optimization process. Indeed, keeping the cost of run-time code generation low appears to be critical for languages such as ML, because typical programs do not manipulate large data sets and cannot easily amortize the cost incurred by a general-purpose run-time compiler.

Further experimentation will be required to fully evaluate the progress that has been made thus far, and we have identified numerous areas for further research. We are currently extending the prototype FABIUS compiler to support a richer source language, including mutable data structures and higher-order functions. We also plan to investigate the feasibility of run-time register assignment, scheduling, and other run-time optimizations.

We have postponed implementing a major component of the FABIUS system, a tagging scheme for integers and pointers, because of recent advances in type-directed compilation. Existing ML compilers commonly use tags to distinguish between integers and pointers to support efficient garbage collection, but this approach has undesirable side effects: integers are restricted to 31 bits and numerous tagging and untagging operations are involved in arithmetic operations. The TIL compiler project [34] has shown that a principled use of types at compile time and run time permits most ML code to be compiled into tag-free, monomorphic code. Hence, we have focused on efficiently compiling such code and have ignored the problem of tagging.

Our experience with benchmark programs has led us to the conclusion that this technology is usable but not yet foolproof. The use of a high-level programming language greatly reduces the programmer effort required to make use of run-time code generation, but it can be difficult for the programmer to predict program behavior. Memoization can be expensive and is sometimes ineffective, and the heuristic we employ to control run-time inlining occasionally leads to over-specialization or under-specialization. Recent advances in type theory have suggested a mechanism for providing better feedback to programmers [12].

Ultimately, the feasibility of deferred compilation will have to be demonstrated on larger, more realistic programs. It is encouraging to see that a prototype such as FABIUS can already achieve good results.

References

[1] AGESEN, O., AND HÖLZLE, U. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA'95 Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 1995).

[2] APPEL, A. W. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[3] APPEL, A. W., AND MACQUEEN, D. B. Separate compilation for Standard ML. In *PLDI'94 Conference on Programming Language Design and Implementation* (June 1994), pp. 13–23.

[4] AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. Fast, effective dynamic compilation. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996).

[5] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (December 1995), pp. 267–284.

[6] BIAGIONI, E., HARPER, R., AND LEE, P. Signatures for a protocol stack: A systems application of Standard ML. In *Conference on Lisp and Functional Programming* (June 1994).

[7] BONDORF, A., AND DANVY, O. Automatic autoprojection of recursive equations with global variables and abstract data types. *Sci. Comput. Programming 16*, 2 (September 1991), 151–195.

[8] CHAMBERS, C., EGGERS, S. J., AUSLANDER, J., PHILIPOSE, M., MOCK, M., AND PARDYAK, P. Automatic dynamic compilation support for event dispatching in extensible systems. In *WCSSS'96 Workshop on Compiler Support for System Software* (February 1996).

[9] CHAMBERS, C., AND UNGAR, D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI'89 Conference on Programming Language Design and Implementation* (June 1989), pp. 146–160.

[10] CMELIK, R. F., AND KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. Tech. Rep. 93-06-06, Department of Computer Science and Engineering, University of Washington, June 1993.

[11] CONSEL, C., AND NOËL, F. A general approach to run-time specialization and its application to C. In *POPL '96 Symposium on Principles of Programming Languages* (January 1996), pp. 145–156.

[12] DAVIES, R., AND PFENNING, F. A modal analysis of staged computation. In *POPL '96 Symposium on Principles of Programming Languages* (January 1996), pp. 258–270.

[13] DEUTSCH, L., AND SCHIFFMAN, A. M. Efficient implementation of the Smalltalk–80 system. In *POPL'84 Symposium on Principles of Programming Languages, Salt Lake City* (January 1984), pp. 297–302.

[14] DRAVES, S. Compiler generation for interactive graphics. In *Dagstuhl Seminar on Partial Evaluation* (February 1996).

[15] ENGLER, D. R. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996).

[16] ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL '96 Symposium on Principles of Programming Languages* (January 1996), pp. 131–144.

[17] ENGLER, D. R., AND PROEBSTING, T. A. DCG: An efficient, retargetable dynamic code generation system. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)* (October 1994), ACM Press, pp. 263–272.

[18] ENGLER, D. R., WALLACH, D., AND KAASHOEK, M. F. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

[19] Feeley, M., Turcotte, M., and Lapalme, G. Using Multilisp for solving constraint satisfaction problems: An application to nucleic acid 3D structure determination. *Lisp and Symbolic Computation 7* (1994), 231–247.

[20] Hölzle, U., and Ungar, D. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI'94 Conference on Programming Language Design and Implementation* (June 1994).

[21] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[22] Keppel, D. A portable interface for on-the-fly instruction space modification. In *Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991), pp. 86–95.

[23] Keppel, D., Eggers, S. J., and Henry, R. R. A case for runtime code generation. Tech. Rep. 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[24] Keppel, D., Eggers, S. J., and Henry, R. R. Evaluating runtime-compiled value-specific optimizations. Tech. Rep. 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[25] Leone, M., and Lee, P. Lightweight run-time code generation. In *PEPM 94 Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (June 1994), Technical Report 94/9, Department of Computer Science, University of Melbourne, pp. 97–106.

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

[27] Massalin, H., and Pu, C. Threads and input/output in the Synthesis kernel. In *ACM Symposium on Operating Systems Principles* (1989), pp. 191–201.

[28] McCanne, S. The Berkeley Packet Filter man page. BPF distribution available at `ftp://ftp.ee.lbl.gov`.

[29] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Winter 1993 USENIX Conference* (January 1993), USENIX Association, pp. 259–269.

[30] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles* (November 1987), ACM Press, pp. 39–51. An updated version is available as DEC WRL Research Report 87/2.

[31] Pike, R., Locanthi, B., and Reiser, J. Hardware/software trade-offs for bitmap graphics on the Blit. *Software — Practice and Experience 15*, 2 (February 1985), 131–151.

[32] Poletto, M., Engler, D. R., and Kaashoek, M. F. `tcc`: A template-based compiler for 'C. In *WCSSS'96 Workshop on Compiler Support for System Software* (February 1996), pp. 1–7.

[33] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *Symposium on Operating Systems Principles* (December 1995).

[34] Tarditi, D., Morrisett, J. G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996).

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

[36] Volanschi, E.-N., Muller, G., and Consel, C. Safe operating system specialization: the RPC case study. In *WCSSS'96 Workshop on Compiler Support for System Software* (February 1996).

[37] Wainwright, R. L., and Sexton, M. E. A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming 2*, 1 (January 1992), 61–72.