Optimizing ML with Run-Time Code Generation

`detex paper | wc` = 5246 words, excluding figures

Peter Lee Mark Leone

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213-3891

{petel,mleone}@cs.cmu.edu
(412) 268-3049

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 cycles 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 kernel uses for fast selection of network packets on behalf of user processes, 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 extended abstract, 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 illustrate in an especially vivid way the potential for this technology. We then briefly discuss the performance of several other benchmarks and mention the particular lessons learned from each. Finally, we conclude with a number of directions for further research.

2 Previous Approaches to Run-Time Code Generation

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. However, generality usually incurs a penalty in run-time performance. So, in practice, programmers often write unwieldy but efficient specialized programs instead. Similarly, modularity and abstraction are desirable design goals for most software systems. But, in practice, the costs of modularity often conflict with the requirement for good performance. So once again programmers must sacrifice good system structure for practical reasons.

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 a executable finite state machine in the form of native code for the IBM 7094 [1]. This program was both general (because it accepted any regular expression) and fast (because it quickly generated special-purpose code).

Run-time code generation also led to notable performance improvements in the areas of graphics (in `bitblt` code [2]), operating systems (in the Synthesis kernel [3]), method dispatch in object-oriented systems (in Smalltalk [4] and SELF [5]), instruction-set simulation (in Shade [6]), and many others. Indeed, 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 a corresponding increase of 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. [7, 8]. Additional arguments for run-time code generation can be found in [9].

2.1 Run-Time Compilers

Despite the increasing attention, researchers have done relatively little to automate and optimize the process of run-time optimization and compilation itself. Indeed, existing approaches, while 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 [10] 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, in the form of compiled macros. The advantage of macros is that the programmer can specify the construction of terms at the source level, rather than as intermediate code trees. Recently, Engler and colleagues added support for this style of run-time code generation on top of DCG, in a system called ‘C [10], with good results. An even better approach is to design the programming language so that run-time code generation can be smoothly integrated into the semantics of the language. For example, in an implementation of SELF [5], the system provides run-time compilation of type-specialized versions of methods by invoking the compiler at run time automatically, without any need for the programmer to specify anything in the program.

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 [10].

Fundamentally, the use of a general tool such as a compiler—even if it is just the back-end of a compiler—is at odds with the goal of eliminating the cost of generality in favor of the efficiency of specialization.

2.2 Templates

An effective but crude way to eliminate the overhead of a general compiler is to use native-code templates that can be quickly instantiated at run time and then immediately executed. Basically, several templates can be constructed ahead of time, and then the best one chosen and instantiated at run time. Several systems have used this approach with great success, such as Pike, Locanthi, and Reiser's **bitblt** compiler [2] and in Massalin and Pu's **Synthesis** kernel [11]. Templates have the advantage of eliminating the need for a general compiler, but the disadvantage of requiring the programmer to manually construct the templates (in machine or assembly language!) along with a procedure for selecting and instantiating them. Typically, this is an ad-hoc process, with no opportunity for re-using the templates or instantiation mechanisms. Furthermore, since the programmer must write the templates as low-level code, porting becomes problematic. These difficulties effectively limit the range of potential optimizations. As we shall see in this paper, optimizations such as instruction selection based on run-time values lead to significant improvements in running time. Performing such optimizations in an ad-hoc manner would be difficult and error-prone.

The best approach, we believe, views run-time code generation as a compiler optimization technique. This is in contrast to systems like DCG and 'C, which provide programming extensions, but similar to SELF in that it is an automatic part of the language implementation. On the other hand, the SELF approach does not take the cost of run-time compilation to be a main concern.

3 The Fabius Compiler

We have previously given a high-level overview of our approach, which we call *deferred compilation* [12]. The main goals of deferred compilation are to minimize the costs of run-time code generation while at the same time allowing a full 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, booleans, and vectors. We chose this language for two main reasons. First, we are involved in a more general project to investigate the practicality of ML for systems programming (specifically, network protocols) [13]. 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 curried function $f : \tau_1 -> \tau_2 -> \tau_3$ is applied to an argument $x : \tau_1$, it may be profitable to generate optimized code for $(f x)$ 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 eliminates the need for analyzing evaluation-order dependencies. Also, the lack of

[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.

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 will use a simple example, the 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, this 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))
```

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 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:

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

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

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, since 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 [12] for details about this.)

After annotating the program, FABIUS compiles it into an RTL-like intermediate representation, 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 is shown below.

[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 [12].

(In the interests of clarity, we have substituted symbolic register names and elided the **early** annotations.[3])

```
loop: beq  $i, $n, L1
      sll  $i1, $i, 2          ; Scale index by shifting
      addu $p1, $v1, $i1       ; Compute address
      lw   $x1, ($p1)          ; Fetch number
      sll  $i2, $i, 2          ; Scale index by shifting
      lw   $x2, $i2($p2)       ; Fetch with immediate 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 of 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 since 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 then translated into code that constructs

[3]There are other simplifications as well: we elided 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.

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       $tmp, 0x852020
sw       $tmp, ($cp)
addiu    $cp, $cp, 4
```

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 pre-fetching mechanism. We discuss these in Section 3.4. Overall, the cost of code emission is about six instructions per instruction generated, as additional overhead is incurred by a memoization mechanism, which is 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, since 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` and emitted as follows:

```
lui      $tmp, 0x8c66              ; Load opcode into upper half
or       $tmp, $tmp, $i2           ; Load offset into lower half
sw       $tmp, ($cp)
addiu    $cp, $cp, 4
```

However, since 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

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:

---

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

```
        li      $tmp, 0x8000
        addu    $tmp, $i2, $tmp          ;  Scale value of i2 for unsigned comparison
        li      $tmp2, 0xffff
        sltu    $tmp, $tmp2, $tmp        ;  Does i2 fit in 16 bits?

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

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 and their benefit is often very significant. For example, run-time strength reduction greatly improves the performance of dot product on 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 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, since 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.

Instruction prefetching also complicates matters, since programs compiled by Fabius interleave ordinary computations with code generation. *[Due to the length restriction, a complete discussion of caching and prefetching issues has been omitted from this extended abstract, but appears in the full paper.]*

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 above 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 causes unacceptable duplication of effort in some benchmarks because structurally equivalent values fail to match.

[Discussion of additional code-generation issues appears in the full paper.]

4 Preliminary Results

To evaluate the performance of FABIUS, we have experimented with a number of small and medium-sized benchmark programs.

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 more complicated specialized algorithm from [10], based on indirection vectors, that 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 a bounds check. Both C implementations used statically sized 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 integers; sparse matrices were 90% zero, and the non-zero elements were randomly located.

Figure 1 compares the overall execution times of the ML code, as compiled by FABIUS, with 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. 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 specialized 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 39% slower than the specialized 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

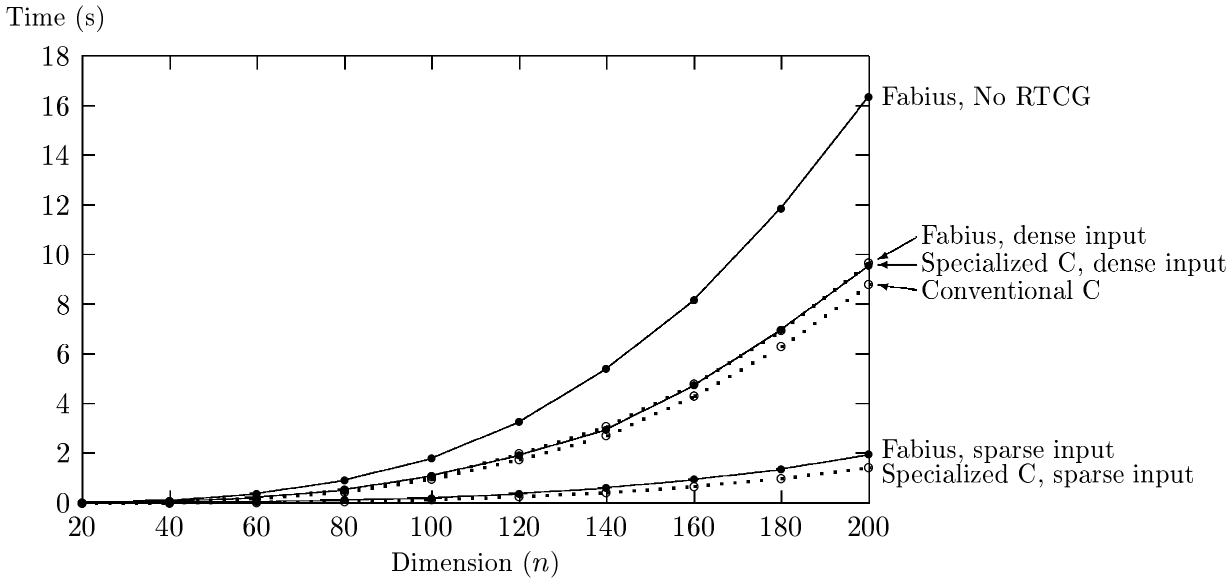

Figure 1: Time to multiply two $n \times n$ matrices (dense and sparse)

by 25%). Efficient re-use 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 this has a distinct disadvantage: it can be difficult for user-level processes to specify precisely the types of packets they wish to receive, since 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 [14, 15]. 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 significant overhead: the selection predicate is re-interpreted 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 mentioned by others [7].

To investigate the feasibility of this idea, we implemented the BSD packet filter language [15] using Fabius and compared its performance to BPF, a kernel-resident interpreter implemented in C [16]. The BSD packet filter language consists 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.

The Fabius implementation of the BPF interpreter is a simple tail-recursive function called `eval`, summarized in Figure 2, whose parameters encode the machine state. Note that the `eval`

```
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            (* Instructions are pairs of words *)
 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 2: Packet filter interpreter

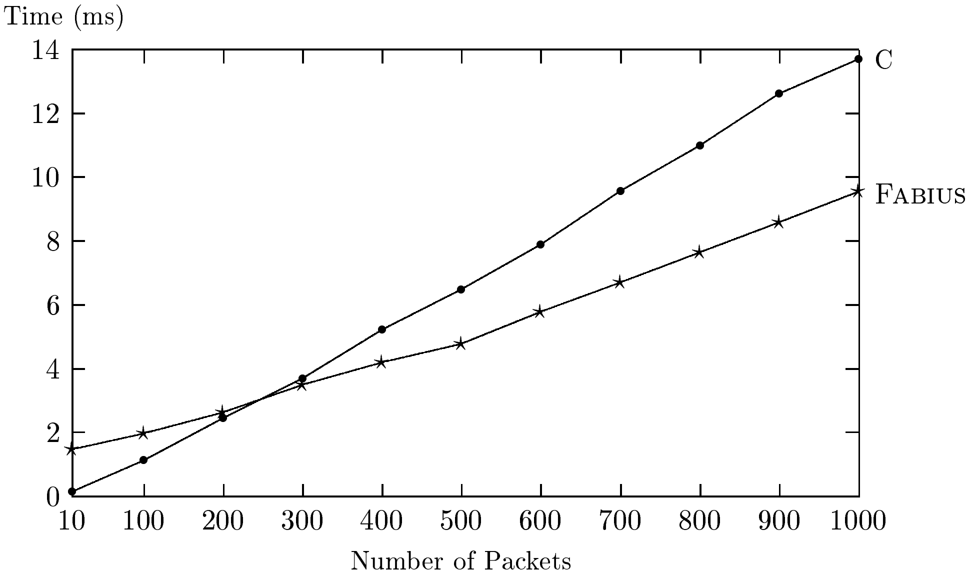

Figure 3: Run-time code generation for a packet filter

function 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, this `eval` function 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 machine state. For example, consider the following filter program, which selects packets whose Ethernet type field specifies an IP packet:

```
     LD   4                ; Load 5th packet word into accumulator.
     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       ; Compare load offset to packet size.
     beq  $t, $0, L1
     lw   $a, 16($p)       ; Load 5th packet word into accumulator.
     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:  lui  $r, 65535        ; Return -1 if indexing error.
     ori  $r, $r, 65535
L4:  jr   $ra
```

Figure 3 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, in order 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

[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.

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 ms to process a packet, whereas BPF averaged 13.7 ms 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, since 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 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 [17]. 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. 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.

Figure 5(b) demonstrates the improvement obtained in a regular-expression matching algorithm using run-time code generation. 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. 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 [3]; a sample of the lookup code generated at run time is contained in Figure 4.

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 how this improves the performance of a commonly used ML benchmark, Conway's game of life, which uses a set to record the locations of living cells [18].

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

```
      li   $tmp, 3
      beq  $arg, $tmp, L5
      li   $tmp, 2
      beq  $arg, $tmp, L3
      li   $tmp, 1
      beq  $arg, $tmp, L1
      ... Otherwise 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 4: An executable association list

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, this does not improve performance.

We also experimented with the Pseudoknot benchmark [19], which a floating-point intensive program that attempts to determine the three-dimensional structure of portions of RNA molecules using constraint satisfaction and backtracking search. We used FABIUS to create a specialized search function that performs constraint checking only when truly necessary. However, this did not have the desired effect: overall execution time was reduced by approximately five percent (the problem size was fixed).

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 like ML, because typical programs do not manipulate large data sets and cannot easily amortize the cost incurred by a general-purpose run-time compiler.

Futher 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 feasiblity 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

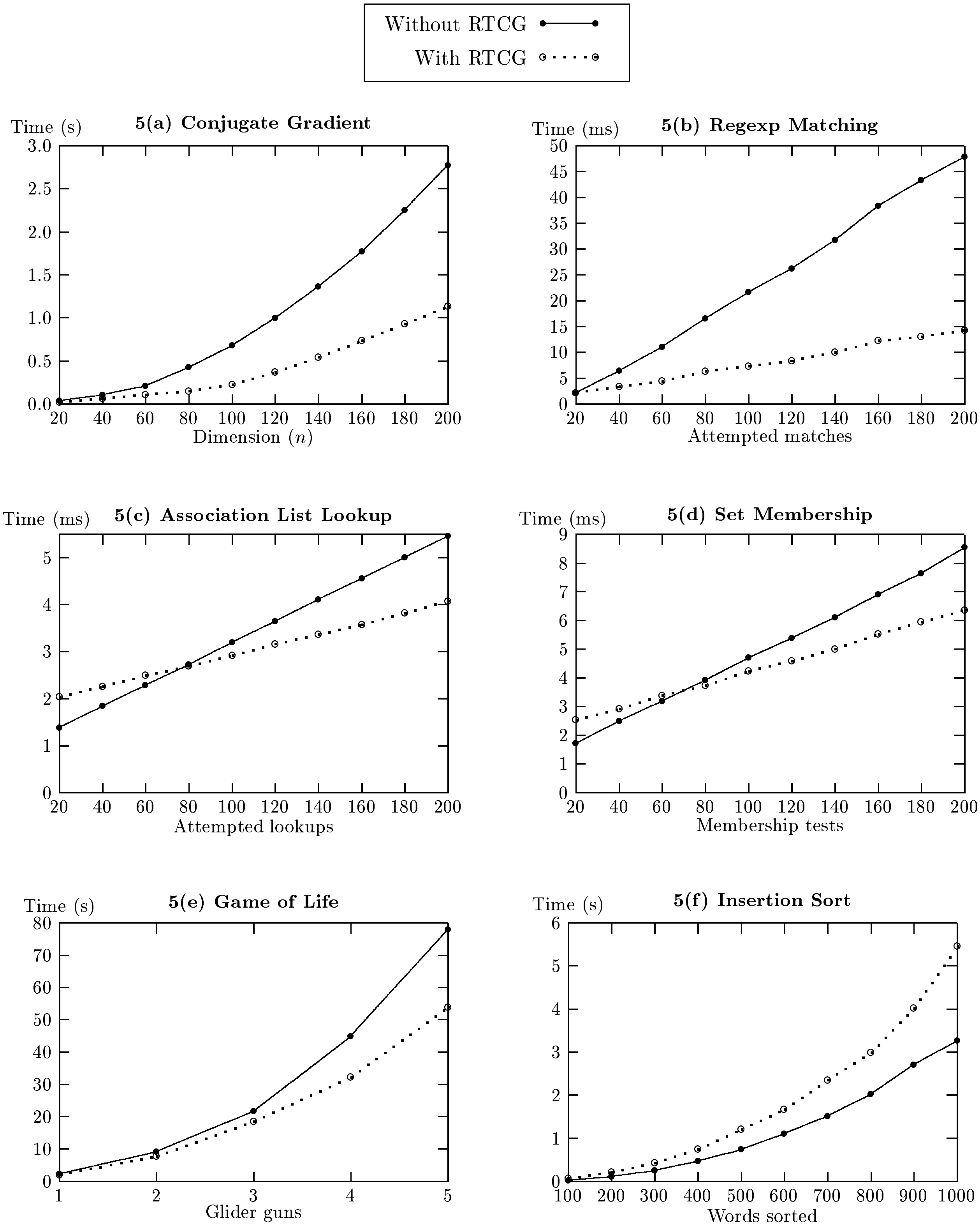

Figure 5: Additional Benchmarks

for integers and pointers, because of recent advances in type-directed compilation. Existing ML compilers commonly use tags to distinguish between integers and pointers in order to support efficient garbage collection, but this 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 [20] 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 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 the 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 [21].

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

[2] 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.

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

[4] 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.

[5] 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*, pages 146–160, June 1989.

[6] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report 93-06-06, Department of Computer Science and Engineering, University of Washington, June 1993.

[7] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *To appear in the 1995 ACM Symposium on Operating Systems Principles*, 1995.

[8] Steve Lucco. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1994.

[9] 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.

[10] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–272. ACM Press, October 1994.

[11] 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, 1989.

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

[13] Edoardo Biagioni, Robert Harper, and Peter Lee. Signatures for a protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, Orlando*, June 1994.

[14] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51. ACM Press, November 1987. An updated version is available as DEC WRL Research Report 87/2.

[15] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

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

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

[18] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[19] Marc Feeley, Marcel Turcotte, and Guy Lapalme. Using multilisp for solving constraint satisfaction problems: An application to nucleic acid 3D structure determination. *Lisp and Symbolic Computation*, 7:231–247, 1994.

[20] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Submitted to PLDI'96*, 1995.

[21] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *ACM Symposium on Principles of Programming Languages*, January 1996. To appear. An earlier version is available as Technical Report CMU–CS–95–145, School of Computer Science, Carnegie Mellon University.